UNIVERSIDAD AUTÓNOMA DE MADRID
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

Universidad Autónoma
de Madrid

DOCTORAL THESIS

Engineering Recommender Systems for Modelling Languages: An Automated End-to-end Approach

THIS DISSERTATION IS SUBMITTED IN THE FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor of Philosophy in Computer and Telecommunication Engineering

AUTHOR
Lissette Almonte García

SUPERVISORS
Dr. Esther Guerra Sánchez
Dr. Iván Cantador Gutiérrez

ABSTRACT

Recommender systems (RSs) are ubiquitous in all sorts of online applications, such as shopping, media broadcasting, travel and tourism, among many others. They are also employed in software engineering tasks, such as software modelling, where we are recently witnessing proposals to enrich modelling languages and environments with the provision of personalised recommendations to the user. Specifically, recommenders for modelling assist in building models by suggesting items based on previous solutions to similar problems in the same domain.

Building a RS for a modelling language, however, demands significant effort and specialised knowledge. Furthermore, while there are numerous RSs for modelling tasks available today, they are typically tailored to particular modelling languages, and are created as independent programs that subsequently need to be integrated into a modelling tool, incurring high development effort. In general, it is not possible to reuse a RS created for a modelling language with a different notation, even if the languages share similarities.

To address these problems, this thesis proposes an automated end-to-end approach to create RSs for modelling languages. It is based on the use of a domain-specific language called Droid to configure every aspect of a RS: the type of the modelling elements to be recommended, the gathering and preprocessing of training data to build the RS, the recommendation method to use by the RS, and the metrics to evaluate the RS. Once configured, the system is deployed as a service to facilitate its integration with third-party tools. This service adheres to a reference recommendation API that enables indexing RSs, investigating their properties, and obtaining recommendations likely coming from various sources. Moreover, Droid is extensible with new data sources and recommendation methods, hence facilitating its evolution.

As a second contribution, this thesis presents a methodology that enables the reuse of RSs created for one modelling language with other notations via a structural mapping, and provides automation for their integration into modelling tools. In addition, it facilitates the aggregation of recommendations coming from multiple sources. This methodology is supported by IronMan, an Eclipse plugin that automates the integration of existing RSs within Sirius and EMF tree editors, bridging recommenders created for one modelling language for their reuse with a different one.

The developed tools have been evaluated through offline experiments, a user study, and two tool validations. Overall, these evalua-

tions have served to measure the precision, completeness, diversity, and perceived quality of the generated recommendations, as well as to validate the tools though various use cases.

Keywords: Recommender Systems, Modelling Languages, Model-driven Engineering, Domain-specific Languages, Modelling Tools

RESUMEN

Los sistemas de recomendación (SR) son ubicuos en todo tipo de aplicaciones en línea, como compras, retransmisión de contenidos, viajes y turismo, entre muchas otras. También se emplean en tareas de ingeniería del software, como el modelado de software, donde encontramos propuestas recientes para enriquecer los lenguajes y entornos de modelado con la provisión de recomendaciones personalizadas para el usuario. En concreto, los recomendadores para modelado ayudan en la construcción de modelos sugiriendo nuevos elementos en base a soluciones previas para problemas similares en el mismo dominio.

Sin embargo, construir un SR para un lenguaje de modelado requiere un esfuerzo significativo y conocimientos especializados. Además, aunque hoy en día existen numerosos SR para tareas de modelado, suelen estar diseñados para un lenguaje de modelado específico, y se crean como programas independientes que posteriormente deben integrarse en una herramienta de modelado, lo que conlleva un alto esfuerzo de desarrollo. Por lo general, no es posible reutilizar un SR creado para un lenguaje de modelado con una notación diferente, incluso si ambos comparten similitudes.

Para abordar estos problemas, esta tesis propone un enfoque integral automatizado para crear SR para lenguajes de modelado. Se basa en el uso de un lenguaje de dominio específico llamado Droid para configurar todos los aspectos del SR: el tipo de elementos de modelado que se recomendarán, la recopilación y preprocesamiento de datos de entrenamiento para construir el SR, el método de recomendación a utilizar por el SR, y las métricas para evaluar el SR. Una vez configurado, el sistema se despliega como un servicio para facilitar su integración con herramientas de terceros. Este servicio es conforme a una API de recomendación de referencia que permite indexar SR, investigar sus propiedades, y obtener recomendaciones que podrían provenir de varias fuentes. Además, Droid puede extenderse con nuevas fuentes de datos y métodos de recomendación, facilitando así su evolución.

Como segunda contribución, esta tesis presenta una metodología que permite la reutilización de SR creados para un lenguaje de modelado con otras notaciones a través de un mapeo estructural, y proporciona automatización para su integración en herramientas de modelado. Además, facilita la agregación de recomendaciones provenientes de múltiples fuentes. Esta metodología está respaldada por IronMan, un plugin de Eclipse que automatiza la integración de SR existentes en editores Sirius y EMF, donde recomendadores creados para un lenguaje de modelado pueden reutilizarse con un lenguaje diferente.

Las herramientas desarrolladas se han evaluado mediante experimentos offline, un estudio con usuarios, y dos validaciones de las herramientas. En conjunto, estas evaluaciones han permitido medir la precisión, completitud, diversidad y calidad percibida de las recomendaciones generadas, así como validar las herramientas a través de diversos casos de uso.

Palabras clave: Sistemas de Recomendación, Lenguajes de Modelado, Ingeniería Dirigida por Modelos, Lenguajes de Dominio Específico, Herramientas de Modelado

ACKNOWLEDGEMENT

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

ATL Atlas Transformation Language. xiii, 33, 36, 41, 43, 45, 48, 56
BC Borda Count. xiii, 25, 94, 107, 114
BPMN Business Process Model and Notation. xiii, 25
CACF Context-Aware Collaborative Filtering. xiii, 80, 81, 101, 113, 120
CARS Context-Aware Recommender Systems. xiii, 19
CBIB Item-Based Collaborative Filtering with Content-Based Similarity. xiii, 81, 101, 113, 120
CBUB User-Based Collaborative Filtering with Content-Based Similarity. xiii, 81, 101, 113, 120–125, 127, 129, 131, 133, 135–138, 140
CB Content-Based. xiii, 13, 15, 17–20
CF Collaborative Filtering. xiii, 15–20
CosineCB Content-Based Cosine Similarity. xiii, 80, 101, 113, 120, 121, 123, 127, 129, 131, 133, 135–138, 140
DCG Discounted Cumulative Gain. xiii, 23
DSL Domain-Specific Language. xiii, 4, 25, 26, 30, 33, 34, 36, 43, 54, 65, 71, 73–76, 78–81, 87, 95, 97, 102, 113, 152
EMF Eclipse Modelling Framework. i, xiii, 5, 32, 33, 36, 43, 48, 64, 89, 91, 111, 143, 146, 153, 157
ER Entity-Relationship. xiii, 145, 146, 148, 153, 157
FN False Negatives. xiii, 23
FP False Positives. xiii, 23
GNN Graph Neural Network. xiii, 42
GPL General-Purpose Language. xiii, 25, 30, 34
IBCF Item-Based Collaborative Filtering. xiii, 80, 101, 113, 120
IDCG Ideal Discounted Cumulative Gain. xiii, 23
IDF Inverse Document Frequency. xiii, 14
IFML Interaction Flow Modelling Language. xiii, 145, 146, 148, 153, 157
IR Information Retrieval. xiii, 11, 14
ISC Item Space Coverage. xiii, 24, 81, 95, 120, 121, 125
ISD Information System Designer. xiii, 146
ItemPop Item Popularity. xiii, 80, 101, 113, 120, 121, 123, 127, 129, 131, 133, 135, 136, 138, 140
KB Knowledge-Based. xiii, 17, 18
LOO Leave-One-Out. xiii, 21
LoC Lines of Code. xiii, 66, 144–146, 148, 153
M2M Model-to-Model. xiii, 31–34, 36
M2T Model-to-Text. xiii, 31–34, 36
MAE Mean Absolute Error. xiii, 22, 36, 58
MAP Mean Average Precision. xiii, 82, 95, 120, 121, 125

MDE Model-Driven Engineering. xiii, 3, 4, 11, 25, 26, 32, 34, 36–38, 40, 46–48, 50, 58, 59, 62–67, 71, 73, 95, 151, 155
MF Matrix Factorization. xiii, 16
MOF Meta Object Facility. xiii, 29, 30, 40
MRA Median Rank Aggregation. xiii, 25, 94, 95, 107, 114
MRR Mean Reciprocal Rank. xiii, 58, 62, 67
MSE Mean Squared Error. xiii, 16
NLP Natural Language Processing. xiii, 42, 43
NL Natural Language. xiii, 141–143
NN Neural Network. xiii, 16, 17, 42
OCL Object Constraint Language. xiii, 29, 36, 45, 48, 55, 56, 78, 93, 148, 154, 158
OMG Object Management Group. xiii, 29, 59, 145
OSGi Open Service Gateway Initiative. xiii, 33
RMSE Root Mean Squared Error. xiii, 22, 36, 58
RRF Reciprocal Rank Fusion. xiii, 25
RS Recommender Systems. i, xiii, 3–6, 11–15, 17–21, 23, 24, 35–38, 40–42, 44, 47–52, 54–67, 71, 73–78, 80–95, 97, 98, 101, 102, 104–109, 112–114, 117, 118, 120, 121, 123, 125, 138, 141, 142, 144–146, 148, 151–154
SGD Stochastic Gradient Descent. xiii, 17
SPL Software Product Line. xiii, 34–36
T2M Text-to-Model. xiii, 31–33
TF Term Frequency. xiii, 14
TP True Positives. xiii, 23
UBCF User-Based Collaborative Filtering. xiii, 80, 101, 113, 120
UML Unified Modelling Language. xiii, 4, 25, 40, 42, 43, 46, 53, 54, 56, 61, 71, 77, 78, 84, 85, 87, 91–93, 101, 103, 108, 109, 112, 117, 118, 120, 133, 145, 146, 148, 152, 153, 156, 157
URI Uniform Resource Identifier. xiii, 77
USC User Space Coverage. xiii, 24, 82, 95, 120–122, 124, 125
VSM Vector Space Model. xiii, 13, 14
XMI XML Metadata Interchange. xiii, 33, 36, 101
XML Extensible Markup Language. xiii, 33, 44
nDCG normalised Discounted Cumulative Gain. xiii, 23, 36, 58, 81, 95, 120, 121, 125

CONTENTS

1 INTRODUCTION

This chapter presents a comprehensive overview of this thesis. Section 1.1 delves into the motivation and proposals of the thesis. Section 1.2 reports on the publications that have originated from the conducted work. Section 1.3 provides an account of a research stay conducted during the course of the thesis. Lastly, Section 1.5 describes the structure of this document.

1.1 MOTIVATION

Recommender Systems (RS) are information filtering systems that assist users in choosing from a potentially large collection of items. Their goal is to predict and exploit the preferences of the user to offer a personalised list of items [5]. RSs are present within all sorts of platforms, providing suggestions on a variety of items, such as music and video on streaming platforms (e.g., Spotify and Netflix), or products to buy on e-commerce sites (e.g., Amazon).

RSs are also used to assist developers in software engineering activities [42, 135]. These include recommenders of code refactorings [117], API code examples [2], meaningful method names [95], or development team members [163], to name a few.

Following this trend, proposals of RSs for software modelling and design tasks have been increasingly appearing in recent years [41]. Modelling is central to most engineering disciplines and essential for software engineering. Indeed, some software development paradigms like Model-Driven Engineering (MDE) have models as the main assets of the development process [25]. Therefore, mechanisms that help designers to find, create, complete, repair and reuse models based on existing knowledge are of great importance. For this reason, researchers have proposed RSs for modelling tasks, such as model completion [101, 105], model finding [36, 100], and model repair [74, 112].

With respect to the creation of RSs, building a RS by hand is costly, as it requires selecting and configuring the most convenient recommendation algorithm for the problem at hand – which demands specialised knowledge – and then integrating it into a tool. This is especially challenging when building a RS for modelling, as modelling languages can be of different nature and target diverse domains.

Building a RS for a modelling language involves several steps, most importantly selecting the data for training and testing the RS, preprocessing these data (e.g., to fix ambiguities or unify item names), configuring the recommendation algorithm, evaluating the RS with

suitable metrics, and deploying the RS within a modelling tool. These tasks require specialised knowledge and high implementation effort. Moreover, software paradigms like MDE or low-code development [45] often need to create new modelling languages and Domain-Specific Languages (DSLs) to express solutions in the targeted domain. Hence, given the potential variety of modelling languages and DSLs, mechanisms to facilitate the construction of RSs for them are required.

To tackle these challenges, this dissertation proposes an MDE solution to automate the creation of RSs for modelling languages. It is supported by a tool, called DROID, which provides:

(i) A DSL to configure the type of items that the RS will recommend (e.g., attributes and methods for class diagrams, activities for process models).

(ii) An engine that automates the preprocessing of models for training and testing, and the evaluation of candidate recommendation algorithms against configurable metrics.

(iii) Facilities to empirically compare distinct RS configurations and identify the most appropriate one.

(iv) A generator that deploys a RS as a service, which heterogeneous modelling clients can integrate.

This solution is extensible, allowing for a lightweight integration of new sources of models, model encodings, and recommendation methods.

Moreover, in light of the growing assortment of recommenders designed by the community for modelling purposes, we harbor a keen interest in exploring the feasibility of reusing these RSs for alternative modelling notations and automatically integrating them into established modelling tools. However, the reuse and integration of RSs pose a number of practical challenges. First, RSs could have been developed for a different (albeit perhaps similar) modelling language, such as certain RS for Ecore models that one may like to reuse for Unified Modelling Language (UML) class diagrams. Second, it can be useful to combine several RSs because they suggest different types of items (e.g., attributes, operations) for distinct target elements (e.g., classes, interfaces). Even if they suggest the same type of items, combining RSs might be useful to retain their best recommendations. Finally, from a technical point of view, RSs may be deployed in numerous ways (e.g., a stand-alone program, a service, within a modelling tool), and have to be integrated within heterogeneous modelling tools (e.g., graphical, textual, tree-based).

In order to tackle these challenges, the approach presented in this thesis facilitates the reutilisation of RSs specifically tailored to a modelling language for alternate notations through structural mapping.

The combination of recommenders for the same type of items (e.g., attributes) relies on the aggregation of their recommendation lists [118], employing a method that is flexible to accommodate various aggregation techniques. The tool IronMan provides comprehensive support for this process, encompassing:

(v) Assistance throughout all phases of the reutilisation and tool integration tasks, including discovery and selection of RSs for modelling, adaptation of the RS to the modelling language (if needed), configuration of the aggregation method, and integration of the assembled recommender within Eclipse modelling editors based on Sirius, and EMF tree editors.

1.2 CONTRIBUTIONS

This section presents the contributions of the thesis. Specifically, Subsection 1.2.1 lists the resulting publications, and Subsection 1.2.2 details the technical achievements.

1.2.1 *Publications*

The publications resulting from this work have been organised into four groups: journals, conferences, workshops and journals under review.

1.2.1.1 *Journals*

1. Lissette Almonte, Esther Guerra, Iván Cantador, and Juan de Lara. "Recommender systems in model-driven engineering: A systematic mapping review". Software and Systems Modeling (Springer), 2022. Volume 21. pp.: 249—280, DOI: 10.1007/s10270-021-00905-x, pp.: 249–280. (JCR Impact Factor in 2022: 2.0, Q3 in Software Engineering).

1.2.1.2 *Conferences*

1. Lissette Almonte, Antonio Garmendia, Esther Guerra, and Juan de Lara. "Reuse and automated integration of recommenders for modelling languages." In Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering, 2023, pages 97–110, ACM, DOI: 10.1145/3623476.3623523 (Conference CORE B in 2023).

2. Lissette Almonte, Esther Guerra, Iván Cantador, and Juan de Lara. "Building recommenders for modelling languages with Droid." In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022, pages

155:1–155:4, ACM, DOI: 10.1145/3551349.3559521 (Conference CORE A* in 2022).

3. Lissette Almonte, Sara Pérez-Soler, Esther Guerra, Iván Cantador, and Juan de Lara. "Automating the synthesis of recommender systems for modelling languages." In Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering, 2021, pages 22–35, ACM, DOI: 10.1145/3486608.3486905 (Conference CORE B in 2021).

1.2.1.3 *Workshops*

1. Lissette Almonte, Iván Cantador, Esther Guerra, and Juan de Lara. "Towards automating the construction of recommender systems for low-code development platforms." In the 1st Low-code Workshop (satellite event of ACM/IEEE MODELS'20), 2020, pages 1–10, ACM, DOI: 10.1145/3417990.3420200.

1.2.1.4 *Journals under review*

1. Lissette Almonte, Esther Guerra, Iván Cantador, and Juan de Lara. "Engineering recommender systems for modelling languages: Concept, tool and evaluation." Empirical Software Engineering, 2023 (JCR Impact Factor in 2022: 4.1, Q2 in Software Engineering). Under review (2nd round).

2. Arsene Indamutsa, Juri Di Rocco, Lissette Almonte, Davide Di Ruscio, and Alfonso Pierantonio. "Advanced discovery mechanisms in model repositories". Software: Practice and Experience, 2023 (JCR Impact Factor in 2022: 3.5, Q2 in Software Engineering). Under review (2nd round).

1.2.2 *Technical contributions*

In addition to scientific publications, this thesis also encompasses the following technical contributions:

1. The framework Droid to automate the configuration, evaluation, synthesis and deployment of RSs for modelling languages. Droid has an extensible architecture that allows incorporating new model sources, data encoding techniques, and recommendation algorithms. The Droid page is available in `https://droid-dsl.github.io/`

2. The framework IronMan to reuse existing modelling recommenders with distinct modelling notations, aggregating them, and integrating them into existing modelling editors. IronMan

has an extensible architecture that allows incorporating additional recommendation aggregation methods and modelling editors. The IronMan page is available in https://github.com/lissetteag/integrate-recommenders-ironman.

1.3 RESEARCH VISITS

Throughout the course of my doctoral studies, I conducted an external research stay in the Software Engineering Department at the University of L'Aquila, Italy, under the supervision of Dr. Davide Di Ruscio. The stay lasted for a period of three months, from January 15 to April 15, 2022. As a result of this collaboration, two extension points were added to the Droid framework. The first extension point was designed for the incorporation of new data encodings, while the second facilitated the inclusion of recommendation methods. The first one enabled the integration of the MDEForge repository, while the second one facilitated the incorporation of the recommender MemoRec within Droid. These developments will be presented in Subsection 5.1.1.

1.4 SUPPORT

The realisation of this PhD was supported by the EU Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 813884 - Lowcomote (https://www.lowcomote.eu/). The main objective of this project was to train a generation of professionals in the design, development and operation of new Large-Scale Collaborative Data Platforms that overcome its limitations by being scalable, open and heterogeneous.

1.5 STRUCTURE OF THE DOCUMENT

The remainder of this dissertation is structured as follows:

1. Chapter 2 presents the background for the thesis. It introduces essential topics of RSs and explores key concepts of MDE.
2. Chapter 3 provides the state-of-the-art on RSs within MDE. It explores RSs developed to assist in modelling tasks. Additionally, it delves into approaches for the automatic generation of RSs.
3. Chapter 4 describes the proposed approach to facilitate the construction of RSs for specific modelling languages, and the integration and reuse of RSs for those languages.
4. Chapter 5 describes the tools developed in the thesis, which are Droid and IronMan. Droid facilitates the construction of RSs

for modelling languages of interest by allowing for the configuration, evaluation, synthesis and deployment of the RSs. IronMan enables adapting and aggregating existing RSs for their use with distinct notations, and integrating them within existing Sirius and tree-based modelling editors.

5. Chapter 6 outlines an offline experiment aimed at assessing the accuracy of the recommendations generated by recommenders created with Droid.

6. Chapter 7 reports on a user study assessing people's perception of the recommendations generated by Droid recommenders.

7. Chapter 8 validates both tools developed in the thesis. It includes the manual integration of RSs generated with Droid into third-party modelling technologies, and the automatic integration of RSs into existing Sirius and tree-based modelling editors.

8. Chapter 9 discusses the conclusions of the thesis and proposes lines of future work.

Part I

CONTEXT

2

BACKGROUND

This chapter presents the fundamental knowledge required to fully understand the thesis work. Specifically, in Section 2.1, we provide an introduction to RSs, a general overview of major recommendation techniques (Subsection 2.1.1), and a description of popular evaluation protocols and metrics (Subsection 2.1.2). Next, in Section 2.2, we provide a broad explanation of MDE and its key concepts. Finally, in Section 2.3, we conclude with a summary of the chapter.

2.1 RECOMMENDER SYSTEMS

RSs are software tools that suggest items that may be of interest to a particular user according to her preferences (e.g., her tastes and interests), supporting decision making tasks in situations of information overload [133]. As a research field, RSs emerged in the mid-90s with roots in areas such as cognitive science and Information Retrieval (IR) [5]. Nowadays, they are ubiquitous and key components in many types of applications for diverse domains, such as streaming platforms of music (e.g., Spotify), video (e.g., Netflix) and media content (e.g., Twitch), e-commerce sites (e.g., Amazon), and social networks (e.g., TikTok), among others.

The term *items* is used to indicate what a RS suggests to its *users*. RSs typically focus on a specific type of item (e.g., movies), using customised graphical interfaces, and filtering and ranking algorithms to deliver useful and effective recommendations of that type of item [133]. Given the prevalent objective of assisting users in discovering relevant items based on their personal preferences, the following mathematical formulation of the recommendation problem was given in [5], one of the earliest comprehensive surveys of the state of the art on RSs.

Consider a system with a set of registered users, denoted as U, and a catalogue of items, denoted as I. Let $f : U \times I \to \mathbb{R}$ be a utility function, where $\mathbb{R}$ represents a totally ordered set. This utility function $f(u, i)$ quantifies the *usefulness* of item i for user u. The objective of a RS is to identify the items $i^*_u \in I$ which are unknown to a target user $u \in U$ and maximise the utility function for that user.

$$i^*_u = \arg\max_{i \in I} f(u, i) \tag{2.1}$$

The set $\mathbb{R}$ typically is composed of positive integer or real numbers within a specified range, serving as a measure of preference or usefulness of the items for the users. However, an essential aspect of this formulation is that the utility function f is not defined for

the entire $U \times I$ set, but rather for a limited subset of it. In other words, the system only has access to utility values for a fraction of the recorded or observed user-item interactions. The objective of a RS is thus to extend the utility function f to the entire $U \times I$ set by estimating the unknown utility values. In the RSs literature, the utility data is commonly represented as a matrix R with dimensions $|U| \times |I|$. This matrix, often referred to as the user-item matrix or rating matrix, organises users and items such that its rows correspond to the users and its columns correspond to the items. Hence, each entry $R_{ui} \in R$ represents the value of the utility function $f(u, i)$.

Rating matrix

To generate personalised item suggestions for users, RSs require data on previous item choices, consumptions and evaluations. These data serve as feedback from the users, and are necessary for accurately predicting user preferences. There are two main types of feedback: *explicit feedback* and *implicit feedback*. Explicit feedback involves users providing evaluations of items they are familiar with. These evaluations are typically in the form of ratings, frequently represented graphically as stars and stored internally as integers on a small scale, such as 1 to 5. A 1-star rating indicates strong dislike, while a 5-star rating indicates strong like. Other forms of explicit feedback include thumbs up/down or like scores. Explicit feedback requires users to intentionally provide their preferences, which can be time-consuming and effortful. Additionally, real-world applications tend to have biases towards positive feedback, as users are more likely to rate items they like. This results in skewed rating distributions, leading to inaccurate predictions by recommendation algorithms. On the other hand, implicit feedback automatically captures user-item interactions as a source of preferences. This type of feedback typically includes item click logs, browsing sessions, consumption counts, and purchase records. Rather than relying on users to actively provide information about their preferences, implicit feedback records their interactions passively [162].

Explicit feedback

Implicit feedback

2.1.1 *Recommendation techniques*

RSs are commonly classified into the following major categories, depending on how they generate personalised recommendations:

- *Content-based* systems, which recommend items that are similar to other items the target user liked in the past [97];
- *Collaborative filtering* systems, which base their suggestions on the items liked by people "similar" to the target user [114, 145];
- *Knowledge-based* systems, which exploit domain knowledge to describe and relate users and items for providing personalised item suggestions [29];

- *Others*
  - *Demographic-based* systems, which use demographic data to represent user and item profiles that are considered in the recommendation generation processes [122];
  - *Social-based* systems, which analyse and exploit social network connections of the target user to provide her with recommendations [66];
  - *Context-aware* systems, which consider the context of the target user (e.g., location, time, weather) to enrich personalised recommendations [6];
  - *Group-based* systems, which provide recommendations to a group of users considering their individual preferences [57];
  - *Sequence-aware* systems, which focus on capturing the temporal dynamics and sequential patterns of user-item behaviors to enhance recommendation accuracy [127];

  and

- *Hybrid* systems, which combine two or more of the previous types of RSs [30].

Additionally, RSs can be categorised according to the algorithmic approach they use to compute the relevance of items [5]. In this regard, we can distinguish between two types of systems:

- *Memory-based* systems, where the relevance of items is estimated through heuristic formulae [145]; and
- *Model-based* systems, which predict item relevance by using a data-based model built via machine learning techniques, e.g., matrix factorisation [88] or neural networks [68].

The following subsections explain in more detail the above categories and types of RSs.

2.1.1.1 *Content-based recommenders*

Content-based recommender systems

Content-Based (CB) recommenders rely on the utilisation of descriptive characteristics to create profiles of users and items. These profiles are then exploited to generate personalised recommendations [133]. The attributes or features of items represent both user and item profiles and determine the corresponding similarities between users and items. Typically, CB systems incorporate metadata and textual information such as keywords and social tags to construct user and item profiles [97]. As a result, according to the considered features, a CB system recommends items that are similar to those for which the target user has previously shown interest [5].

Vector Space Model

Several CB approaches are grounded in the *Vector Space Model* (VSM),

widely recognised in the IR field. In the VSM, each item is represented as a vector of feature weights, denoted as $v_i = (w_1, \ldots, w_N) \in \mathbb{R}^N$, where w_j represents the importance of the j-th feature in describing the item, and N represents the total number of considered features. There are several ways to determine the weight or importance of a feature. One of the most commonly used weighting techniques is TF-IDF, but other popular methods such as BM25 have demonstrated their effectiveness in the context of RSs [32]. One of the factors is Term Frequency (TF) and the other one is Inverse Document Frequency (IDF). In TF-IDF, the combination of two factors computes the weight of a feature f for item i (cf. Equation 2.2).

$$\text{TF-IDF}(f, i) = \text{TF}(f, i) \times \text{IDF}(f) \tag{2.2}$$

The factor TF captures the frequency of occurrence of feature f concerning item i (cf. Equation 2.3). This frequency can be represented in a binary form to indicate the presence or absence of the feature or as an integer to indicate the number of times a term appears within a text document. In this context, TF values are normalised as follows:

$$\text{TF}(f, i) = \frac{\text{count}(f, i)}{\max_{f'} \text{count}(f', i)} \tag{2.3}$$

The rationale behind TF is that the more a feature is utilised in representing an item, the more likely it is to be relevant for that item. On the other hand, the IDF factor quantifies the distinguishing power of each feature across the entire collection of items (cf. Equation 2.4, where n_f represents the number of items in the collection I that contain the feature f). When a feature is more prevalent in the item collection, with a higher nf value, it becomes less informative in characterising a specific item i. Consequently, this leads to a smaller IDF value.

$$\text{IDF}(f) = \log \frac{|I|}{n_f} \tag{2.4}$$

After representing the content of the items as a vector of features, the profile of the user $\tilde{v}_u \in \mathbb{R}^N$ is defined by aggregating the models of the items she liked. To do this, for instance, we could compute the average sum of the corresponding item vectors. The utility of item i for user u is then calculated through similarity metrics applied to the corresponding feature vectors. The cosine similarity is a widely used example of such metrics. It is computed taking the dot product of the two vectors $(\tilde{\mathbf{v}}_u \cdot \tilde{\mathbf{v}}_i)$, which is the sum of the products of their corresponding components (cf. Equation 2.5). Afterwards, the dot product is divided by the product of the magnitudes of the vectors $(\|\tilde{\mathbf{v}}_u\| \cdot \|\tilde{\mathbf{v}}_i\|)$.

$$\cos(\tilde{v}_u, \tilde{v}_i) = \frac{\tilde{v}_u \cdot \tilde{v}_i}{\|\tilde{v}_u\| \cdot \|\tilde{v}_i\|} \tag{2.5}$$

A CB recommender is able to provide accurate personalised suggestions when it has enough information about the preferences of the user, since content similarities can be easily established. Moreover, it is capable of suggesting items for which no feedback has been expressed yet (i.e., cold items), since recommendations are generated via content-based item similarities [5].

Advantages and limitations of content-based recommenders

However, this type of RSs has some disadvantages. One of them is the *overspecialisation* problem, in which the user is exposed to items that are very similar to the ones the user already knows, limiting the discovery of diverse items. In this sense, CB recommenders are not suitable for domains and applications where, at a certain point, the user has to be suggested novel, fresh or even unexpected (serendipitous) recommendations, which can be very valuable in certain domains, such as news articles. Another drawback of the CB approach is the *new user cold-start* problem, as a RS needs a considerable amount of preferences of the user before it can provide well-suited recommendations.

2.1.1.2 *Collaborative filtering recommenders*

Collaborative filtering recommender systems

Collaborative Filtering (CF) recommenders rely on analysing ratings or usage patterns to generate personalised recommendations for users. They make use of user feedback which is often represented in the form of numeric ratings [133]. Consequently, user and item similarities are usually established through explicit or implicit rating-based similarities and patterns [68, 145]. As a result, CF provides suggestions to the user based on items preferred by similar-minded individuals [5].

CF recommenders can be categorised as *memory-based* or *model-based*. Memory-based methods, also known as neighbourhood-based methods, are heuristic formulae or algorithms that estimate ratings by considering the entire set of previously rated items by users. An unknown rating $r_{c,s}$ for user c and item s is commonly calculated aggregating the ratings provided by other users, usually the N most similar ones, for the same item s (cf. Equation 2.6, where $\hat{C}$ denotes the set of N users that are the most similar to user c and have rated item s [5]):

$$r_{c,s} = \frac{\text{aggr}_{r_{c',s}}}{c' \in \hat{C}} \tag{2.6}$$

Equation 2.7 shows an example of aggregation function where the multiplier k acts as a normalising factor, which is generally chosen as $k = \frac{1}{\sum_{c' \in \hat{C}} |\text{sim}(c,c')|}$, where the user's average rating $\bar{r}_c$ is defined as in equation 2.8:

$$r_{c,s} = \bar{r}_c + k \sum_{c' \in \hat{C}} \text{sim}(c,c') \cdot (r_{c',s} - r_{c'}) \tag{2.7}$$

$$r_{c,s} = r_c + k \sum_{c_0 \in \hat{C}} \text{sim}(c, c_0) \cdot (r_{c_0,s} - r_{c_0}) \tag{2.8}$$

Distinct similarity functions can be utilised in CF algorithms. For instance, in the correlation-based method, the similarity between items or users is calculated with the Pearson's correlation coefficient (cf. Equation 2.9, where $\text{sim}(x, y)$ denotes the existing similarity between two users (or items), denoted as x and y; and S_{xy} represents the set of items (or users) that have been rated by (or have rated) both x and y [5]).

$$\text{sim}(x, y) = \frac{\sum_{s \in S_{xy}} (r_{x,s} - \bar{r}x)(ry, s - \bar{r}y)}{\sqrt{\sum s \in S_{xy} (r_{x,s} - \bar{r}x)^2} \sqrt{\sum s \in S_{xy} (r_{y,s} - \bar{r}_y)^2}} \tag{2.9}$$

In the case of model-based approaches, the ratings are used to create a predictive (machine learning) model. They commonly require a set of model parameters that influence on the capture of important characteristics of users and items. Two well-known learning techniques for model-based CF are *Matrix Factorization* (MF) and *Neural Networks* (NNs).

Matrix factorization

MF methods employ machine learning techniques to train a statistical model, focusing on *dimensionality reduction* of the sparse rating matrix by projecting it into a lower-dimensional subspace composed of latent factors. Hence, instead of attempting to fill the entire matrix with potential incomplete or unrelated information, it aims to factorise the observed ratings, forming matrices of rating latent factors. Then, unknown ratings are estimated through the dot product of latent feature vectors (cf. Equation 2.10, where $\hat{r}(u, i)$ represents the estimated rating for a user u and item i). Specifically, the dot product of the latent feature vectors associated with the user $\mathbf{p}_u^T$ and the item $\mathbf{q}_i$ is the rating estimation:

$$\hat{r}(u, i) = \mathbf{p}_u^T \cdot \mathbf{q}_i \tag{2.10}$$

Alternatively, the rating matrix R can be approximated by the product of two matrices, P and Q^T, where P is a matrix of size $|U| \times k$ with the user vectors $\mathbf{p}_u$ as rows, and Q is a matrix of size $|I| \times k$ containing the $\mathbf{q}_i$ vectors as rows (cf. Equation 2.11). The values of these matrices can be automatically estimated from the available data by minimising the *Mean Squared Error* (MSE), the average squared difference between predicted and actual values. This means that the matrices P and Q are chosen in such a way that they minimise the following loss function:

$$L(P, Q) = \sum_{(u,i) \in R} (r_{ui} - \mathbf{p}_u^T \cdot \mathbf{q}_i)^2 + \lambda \left(\sum_u \|\mathbf{p}_u\|_2^2 + \sum_i \|\mathbf{q}_i\|_2^2 \right) \tag{2.11}$$

where R is the set of observed ratings, i.e., the set of non-zero entries of the rating matrix R, and $\lambda > 0$ is a regularisation hyper-parameter used to prevent overfitting.

In [62], this function can be minimised through *Stochastic Gradient Descent* (SGD), a widely used optimisation technique that iteratively updates the parameters in the opposite direction of the gradient. When applied to Equation 2.11, this technique yields the following update rules for the parameters $\mathbf{p}_u$ and $\mathbf{q}_i$ for each rating r_{ui} in the training set:

$$\mathbf{p}_u \leftarrow \mathbf{p}_u - \eta \left(e_{ui} \mathbf{q}_i + \lambda \mathbf{p}_u \right) \tag{2.12}$$

$$\mathbf{q}_i \leftarrow \mathbf{q}_i - \eta \left(e_{ui} \mathbf{p}_u + \lambda \mathbf{q}_i \right) \tag{2.13}$$

Neural networks

On the other hand, NNs offer a great deal of expressive power to capture intricate relationships within the user-item space, surpassing the limitations of matrix factor models. At the output layer, the loss function can incorporate either a pointwise approach involving rating or binary prediction errors, or a pairwise approach considering ranking errors [34].

Advantages and limitations of collaborative filtering recommenders

Unlike CB approaches, CF systems are able to provide more novel and diverse recommendations for a user. Even in situations of rating sparsity, CF has shown better performance than CB in many real-world applications, and represents the most widely used approach for providing personalised recommendations [88].

Nonetheless, similar to CB approaches, CF recommenders suffer from the new user cold-start problem, i.e., they need to have enough ratings of a user to provide her with accurate personalised recommendations. CF also suffers the so-called *item cold-start* problem since an item can only be recommended after being rated. Moreover, CF is affected by situations of high sparsity, where the number of collected ratings is very small compared to the total number of possible ratings given by users to items.

2.1.1.3 *Knowledge-based recommenders*

Knowledge-based recommender systems

Knowledge-Based (KB) recommenders suggest items using domain- specific knowledge about how item attributes and features could meet the user preferences [29]. Many recommendation approaches can be categorised as KB. Among them, three main types of approaches have gained great interest in the literature: *rule-based*, *constraint-based* and *case-based* [133].

Rule-based recommender systems

Rule-based RSs generate recommendations based on predefined, heuristic rules. They are based on explicit if-then clauses created by domain experts or derived from historical user data. Case-based systems, on the other hand, address the recommendation task via case-based reasoning methods, which aim to solve a new problem (i.e., a

Case-based recommender systems

new case) by remembering previous similar cases and reusing knowledge about them. Finally, constraint-based systems predominantly exploit knowledge commonly expressed utilising explicit restrictions on how current user preferences have to be related to the item attributes and features [77].

Constraint-based recommender systems

Additionally, another advanced approach of knowledge-based RSs that also has reached popularity is semantic-based RSs. Broadly, semantic RSs can be split into *ontology-based* and *knowledge graph-based* systems.

Ontology-based recommender systems

On the one hand, ontology-based recommenders employ ontologies for knowledge representation. Ontologies are used to model the domain, the users and items, or the context. They capture the domian concepts, relationships and constraints that are important for building user and item profiles, and improve the effectiveness of recommendations by considering semantic similarity and relatedness between profiles [104].

Knowledge graphs recommender systems

On the other hand, graph-based recommenders model diverse data interconnected, such as user preferences, item attributes, and contextual variables. This knowledge is described as a graph consisting of linked data, where the nodes represent entities (e.g., users and items) and the edges represent relationships between entities, providing a flexible and scalable representation. Then, RSs exploit the knowledge graph to generate personalised recommendations [133].

Advantages and limitations of knowledge-based recommenders

The main advantage of KB systems is their capability of providing and explaining accurate recommendations that entail a deep understanding of the user's preferences, which cannot be achieved by CB and CF approaches. Although KB systems usually do not suffer from cold-start problems, they are affected by the so-called *knowledge acquisition bottleneck*. This problem consists of the need to learn the underlying knowledge and consult domain experts to model and build the knowledge bases. Additionally, KB recommenders are usually ad-hoc solutions to specific problems, making their generalisation to several problems or domains difficult or not possible.

2.1.1.4 *Other recommenders*

There are other types of RSs that can be considered orthogonal to CB, CF and KB systems, since they exploit particular data through CB and CF strategies. Special attention can be drawn to the following recommenders:

Demographic-based recommender systems

- *Demographic-based recommenders* make use of demographic data about the users, e.g., age, gender and address [122]. Taking this information into account, recommendation algorithms identify users or items that are demographically compatible with the target user. These systems assume that users with similar demographic attributes may rate similarly, and have been applied

to alleviate cold-start problems of traditional recommendation approaches.

- *Context-aware recommenders*, commonly abbreviated as CARS, take into consideration contextual information associated or influencing to user preferences when generating personalised recommendations [5]. Quoting Dey [40], "*context is any information that can be used to characterise the situation of an entity*." In CARS, context commonly refers to circumstances in which recommendations are generated, such as the time, the weather, and the current location of a user.

Context-aware recommender systems

- *Social-based recommenders* generate personalised recommendations in social media [66]. A widely explored approach in these systems is the exploitation of explicit relationships between users in online social networks. In this sense, many solutions are based on the so-called trust-based model, where the social influence and trust of users are established and propagated through a social network and considered for providing recommendations. In fact, research supports the theory that social trust can be used as a positive way to generate recommendation explanations [161].

  Social-based methods perform well when used together with other recommendation approaches, like CB or CF, since social network information can help in dealing with the user and item cold-start problems [133].

Social-based recommender systems

- *Group-based recommenders* generate recommendations to a group of users considering the preferences of the individual members of the group. In numerous real-world scenarios, recommendations must be tailored for groups rather than individuals. The primary objective in such scenarios is to generate recommendations that maximise the satisfaction of each group member [57].

Group-based recommender systems

- *Sequence-aware recommenders* generate recommendations by capturing the temporal dynamics and sequential patterns of user behaviors to improve recommendation effectiveness. These RSs usually utilise sequentially ordered logs of user interactions as input, and leverage extracted sequential patterns to generate item suggestions [127].

Sequence-aware recommender systems

2.1.1.5 *Hybrid recommenders*

Hybrid systems make use of two or more recommendation methods, such as CB and CF, to take advantage of their benefits and avoid some of their limitations [30]. Due to this aspect, many real-world RSs are hybrid. Without going into details, we can identify three main ways to implement a hybrid system:

Hybrid recommender systems

- Incorporating some feature of a recommendation method into another one, e.g., a CF strategy that uses CB similarities;
- Combining the recommendations generated separately by two methods, e.g., via ranking aggregation and diversification techniques; and
- Building a unifying recommendation model that incorporates characteristics of distinct methods, e.g., a matrix factorisation model with both collaborative and content-based features.

2.1.2 *Evaluation of recommender systems*

The focus of current research on RSs significantly revolves around practical applications, making the evaluation of these systems a crucial aspect of their deployment. To assess the *effectiveness* and *quality* of these systems, a variety of evaluation methodologies and metrics have been proposed. Given that evaluation plays an essential role at different phases of the RS life cycle, we next discuss different evaluation protocols that can be conducted at various development stages of RSs, as well as popular metrics used for their evaluation.

2.1.2.1 *Evaluation protocols*

Offline experiments

There are many different ways to evaluate RSs. One of the most commonly used approach for evaluating a RS is *offline experiments*. These experiments can be performed at early development stages of the RS, and are relatively simple to design and cost-effective to execute when user-item data are available. They provide a fast way to compare different recommendation methods, and can be viewed as simulations of the system interacting with users. The best offline experiment is the one that accurately represents and emulates the recommendation generation and acceptance setting [133].

Data split

To perform an offline experiment, the first step is to collect a dataset of user ratings, which serves as a source of ground truth for the evaluation. Then, the created dataset is commonly divided into training data and test data, ensuring that the two splits do not overlap. The training data are used as input for building the RS, whereas the test data are used to compute certain evaluation metrics on recommendations generated by the RS [34].

There are several alternatives for splitting the data in offline experiments. Among them, *cross-validation* is widely employed. It partitions the data into multiple subsets or folds. The RS is trained using a combination of all except one of these folds, and the remaining fold is used for testing. *Random* data splitting is commonly used in cross-validation, e.g., selecting 80% of the user-item interactions used for training and the remaining 20% for testing. In this context, the

K-fold cross-validation method involves dividing the data into K approximately equal sets, where one fold is held out as the ground truth for testing, and the remaining folds are used for training. This process is repeated K times, with each fold serving as the test set iteratively [162].

Another offline evaluation methodology is the *Leave-One-Out (LOO)* approach, where certain user preference is withheld one at a time for testing. The remaining data are used for training, and the system generates predictions for the excluded preference. This approach can be considered an extreme variant of *K-fold cross-validation*. However, it is important to note that the reliability of cross-validation, including LOO, may be limited unless the dataset is sufficiently large [75].

Advantages and limitations of offline experiments

The main advantage of offline experiments is their cost-effectiveness as they can be executed whenever is needed. Additionally, compared to other approaches, offline experiments are faster and can be conducted in a controlled environment. Their disadvantages include the limited availability of data in many domains, and potential data quality issues, such as a biased representation of the real world.

Online experiments

A RS can also be evaluated after deployment. This can be done through *online experiments*, which usually are user-centric [85]. Online experiments are particularly suitable for RSs that are already deployed, and have a substantial user base, which is often the case in industry settings. In online experiments, real users engage in actual tasks with the system, providing valuable recommendation feedback for evaluation purposes. A popular technique for conducting online experiments is A/B testing. In an *A/B testing* setting, two versions A and B of the system are deployed, and one of them implements a recommendation algorithm or functionality that is being evaluated. After a period of time, the explicit feedback and implicit behaviour of users recorded in both systems are analysed and compared according to certain metrics [133].

Advantages and limitations of online experiments

These types of evaluations allow for the direct collection of data from real users, providing insights into the overall goals of the system. However, it is important to consider the potential risk of degrading the user experience for customers in the test group. Additionally, uncontrolled external factors can influence the collected data, potentially leading to misleading evaluation results. It is worth noting that while online experiments offer valuable insights, they can be resource-intensive and expensive to conduct [133, 162].

User studies

Finally, *user studies* provide another approach for evaluation. They involve recruiting a group of test subjects, and asking them to perform various tasks on a prototype of the system. The prototype is deployed in a controlled setting, allowing an evaluation with a small set of users. User studies can also incorporate the A/B testing methodology, and include online questionnaires to gather feedback on user satisfaction and opinions regarding the system and its functionali-

ties. User interaction with the system is crucial to collect real-world insights, even when offline experiments are possible.

Advantages and limitations of user studies

Compared to offline and online experiments, user studies offer a wide range of insights. They enable the testing of user behaviour and the impact of recommendations on user actions, addressing assumptions that are typically made in offline experiments. User studies provide both qualitative data and quantitative measurements. While they are not as expensive as online experiments, user studies still involve costs and are limited to a relatively small number of subjects and tasks [133].

2.1.2.2 *Evaluation metrics*

An important factor to consider when conducting evaluations is the selection of metrics [133]. Since they offer valuable insights into various aspects of the recommendations, it is crucial to choose them based on well-defined goals.

Rating prediction metrics

Initially, early papers that focused on the rating prediction task placed significant emphasis on measuring the accuracy of their estimations. Error-based metrics were employed to quantify the level of error on the estimation of real ratings. The most popular error-based metrics are *Mean Absolute Error* (MAE) and *Root Mean Squared Error* (RMSE):

$$\text{MAE} = \frac{1}{|R_{\text{test}}|} \sum_{(u,i) \in R_{\text{test}}} |r_{ui} - \hat{r}_{ui}| \tag{2.14}$$

$$\text{RMSE} = \sqrt{\frac{1}{|R_{\text{test}}|} \sum_{(u,i) \in R_{\text{test}}} (r_{ui} - \hat{r}_{ui})^2} \tag{2.15}$$

In Equation 2.14, MAE is calculated as the average absolute difference between the predicted rating $\hat{r}_{ui}$ and the actual rating r_{ui} for each user-item pair in the test set R_{test}. The summation $\sum_{(u,i) \in R_{\text{test}}} |r_{ui} - \hat{r}_{ui}|$ calculates the sum of all absolute differences, and dividing it by the size of the test set $|R_{\text{test}}|$ gives the average absolute difference, representing the average prediction error [133].

In Equation 2.15, RMSE is calculated in a similar manner, taking the square difference between the predicted rating $\hat{r}_{ui}$ and the actual rating r_{ui} for each user-item pair in the test set R_{test}. The summation $\sum_{(u,i) \in R_{\text{test}}} (r_{ui} - \hat{r}_{ui})^2$ calculates the sum of all squared differences, and dividing it by the size of the test set $|R_{\text{test}}|$ provides the average of the squared differences. Finally, taking the square root of the average squared difference using the $\sqrt{\cdot}$ operator gives the RMSE value, representing the average prediction error with larger errors being emphasised due to the squaring operation.

Ranking-based metrics

Other metrics focus on ranking-based metrics to measure the qual-

ity of recommendation lists. One commonly used ranking-based metric is *precision*, which measures the likelihood that a suggested item is relevant. In Equation 2.16, precision is derived from the concepts of *True Positives* (TP) and *False Positives* (FP) in a binary classification task. TP represents the number of correct positive predictions, whereas FP represents the number of instances that are incorrectly predicted as positive by the model. Another metric is *recall*, which measures the proportion of relevant items in the recommendation lists. In Equation 2.17, recall is derived from the concept TP and the concept of *False Negatives* (FN). FN represents the number of instances that are incorrectly predicted as negative by the model when they are actually positive. A third metric is *F1*, which is the harmonic mean of precision and recall (cf. Equation 2.18) [133]:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2.16}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2.17}$$

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2.18}$$

Another popular metric used to measure the quality of recommendation lists is *normalised Discounted Cumulative Gain* (nDCG). nDCG considers whether the most useful items are positioned at the top of the recommendation lists. To calculate nDCG, we first compute the *Discounted Cumulative Gain* (DCG), cf. Equation 2.19). DCG@k quantifies the cumulative gain of items up to position k in the ranked list. This involves ranking the items based on their relevance scores, calculating the gain for each item using the formula $2^{rel_i} - 1$, and then applying a discount factor by dividing it by the logarithm base 2 of the rank of the item position. By summing up the discounted gains, we obtain DCG@k. Once DCG@k is computed, it can be normalised by dividing it by the *Ideal Discounted Cumulative Gain* (IDCG)@k, which represents the maximum possible DCG value for the given list. The resulting nDCG@k provides a normalised measure that allows fair comparisons of recommendation lists, ensuring that the most relevant items are appropriately ranked and weighted (cf. Equation 2.20) [133].

$$\text{DCG@k} = \sum_{i=1}^{k} \frac{2^{rel_i} - 1}{\log_2(i+1)} \tag{2.19}$$

$$\text{nDCG@k} = \frac{\text{DCG@k}}{\text{IDCG@k}} \tag{2.20}$$

Novelty and coverage metrics

In addition to the previous metrics, there are other complementary metrics that can be utilised to evaluate different aspects of RSs.

One such aspect is the *novelty* of recommendations, which entails suggesting items that are new and unfamiliar to the user. To measure novelty, various metrics can be used, based on randomisation, serendipity and diversification. These metrics focus on the potential enhancement of the user experience by introducing recommendations of fresh and unexpected items. Additionally, coverage metrics play a crucial role in evaluating RSs. For instance, one widely used metric is *User Space Coverage* (USC), which assesses the percentage of users that the RS can recommend (cf. Equation 2.21), and *Item Space Coverage* (ISC), which measures the diversity of the recommendations (cf. Equation 2.22) [33].

$$\text{USC} = \frac{\text{Number of users with at least one recommendation}}{\text{Total number of users}} \tag{2.21}$$

$$\text{ISC} = \frac{\text{Number of recommended items}}{\text{Total number of items}} \tag{2.22}$$

2.1.3 *Recommendation aggregation*

Rank aggregation

Rank aggregation is the process of combining individual rankings from different systems into a single one. When it comes to RSs, various lists of ranked recommendations are merged to reflect an heterogeneous form of personalised information retrieval. To achieve this, a variety of aggregation methods have been developed, each tailored to refine a final ranking according to a particular algorithm [22].

Specifically, a *rank aggregation method* aims to find the best permutation of recommendation lists in terms of a given metric. It can be used to provide more accurate and diverse item suggestions by taking into account weaknesses and biases that specific recommenders may have, and to reduce the impact of items incorrectly ranked in high positions by certain recommender [118].

In information retrieval, rank aggregation methods have been employed to improve search results. Aggregating rankings from distinct search algorithms allows creating a more accurate list of retrieved documents. This is very valuable in domains where presenting the most relevant information to users is essential, such as web search engines, e-commerce platforms, and content recommender systems [22].

Aggregation methods can be either supervised or unsupervised [118] methods. The former target the aggregated ranking that optimises a given metric computed over ground truth data. The latter lack ground truth data, and rely on metrics computed on the available rankings of items.

Besides, unsupervised methods can be categorised into score-based and rank-based approaches. A popular score-based approach is Comb*, a re-ranking method that combines various scoring functions to enhance the order of results. Comb* offers versatility with three main

variations: CombMAX, which selects the maximum score among the functions; CombMIN, which selects the minimum score; and CombSUM, which selecs the sum of the scores [16, 55].

Rank-based methods, by contrast, encompass well-known techniques such as Borda Count (BC), Bordafuse, Condorcet, Median Rank Aggregation (MRA), and Reciprocal Rank Fusion (RRF). BC is a method that assigns points to items based on their rank, offering a structured approach to re-ranking. Bordafuse, a BC-based technique, integrates rankings to produce a refined, consolidated order of items. In the case of Condorcet, a voting-based approach is used to determine the best-ordered list by considering pairwise comparisons between items. MRA ranks items by their median position across individual rankings, providing a robust re-ranking strategy. Finally, RRF combines the ranks by considering the reciprocal of rank positions for each item, thereby contributing to the improvement of overall result relevance [16, 55].

2.2 MODEL-DRIVEN ENGINEERING

Model-driven engineering

MDE is a software engineering approach that positions models at the centre of the software development process. The main purpose of MDE is to reduce the cost of system development by elevating the level of abstraction and bridging the gap between problem-solving mindset of the developers and the way they express solutions [26].

In contrast to traditional development paradigms, where models are typically utilised during the design phase and for system documentation, MDE considers models as crucial artefacts throughout the entire software development life cycle. These models are created early on and play a significant role in specifying, designing, testing, and even generating code for the final applications [25].

MDE models serve as abstract representations of systems, capturing the necessary information to describe a system as a whole. Abstraction in this context involves condensing information, removing accidental details, and emphasising essential aspects. This approach enhances our understanding of the essence of the system and promotes familiarity. Furthermore, as these models are integral to software, their definitions must adhere to formal standards. To achieve this, modelling languages are employed to define models, including General-Purpose Language (GPLs) and Domain-Specific Language (DSLs).

GPLs are languages that can be used across various sectors or domains. For instance, the UMLs [1] or the Business Process Model and Notation (BPMN) [2] can be considered GPLs as they provide a wide range of modelling capabilities applicable to diverse industries and

1 http://www.omg.org/spec/UML/2.5.1
2 https://www.omg.org/spec/BPMN/2.0.2

use cases [25]. On the contrary, DSLs are programming or specification languages designed to cater to a particular problem domain. DSLs provide specialised primitives and concepts that accurately represent the abstractions of a particular domain [23, 166]. As a result, MDE is recognised as an effective approach to enhance software quality by automating repetitive, error-prone or time-consuming tasks [26].

2.2.1 *Domain-specific modelling languages*

Domain-specific languages

A modelling language works as a mean for designers to specify system models. For this purpose, as mentioned before, DSLs are employed. The specific purpose of a DSL is to facilitate the definition of concepts within a particular domain, context or organisation. These languages seek to simplify the task of individuals who need to express ideas and concepts within that specific domain [23, 25].

Figure 2.1 shows the core elements of a modelling language: *Abstract syntax*, *Concrete syntax*, and *Semantics*. The abstract syntax of a modelling language comprises the definition of its main concepts and their relationships, and it is typically defined by means of a meta-model. The concrete syntax assigns a representation to the elements of the abstract syntax. Finally, the semantics establishes the significance of the abstract syntax and, by extension, of the concrete syntax. The next three subsections explore each one of these elements [25].

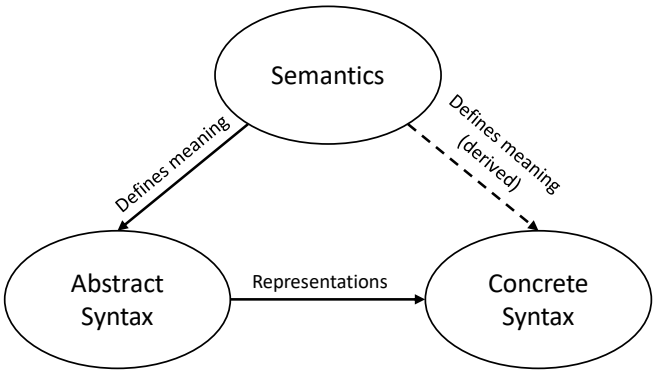


Figure 2.1: The three main elements of a modelling language.

2.2.2 *Meta-modelling*

Meta-model

Meta-modelling is the process of creating or defining meta-models. A *meta-model* is an abstract representation of a problem domain that encompasses all the necessary information to describe systems in that domain comprehensively. It can be used to define the abstract syntax of a modelling language, outlining the concepts, relationships and constraints of the language. By serving as a blueprint, it guides the

creation of models. In this sense, a meta-model is just a model that describes all models considered valid in a language.

Abstract syntax

The *Abstract syntax* of a modelling language, often characterised by a meta-model, defines the structure of the language. The meta-model specifies the domain elements, their characteristics, and the relationships among them. Hence, it is imperative for models to conform to their meta-models as they outline the structure and behaviour of the systems they depict.

Meta-models play a crucial role in defining modelling languages as they provide a description of the entire class of models that can be represented using the language. Commonly, meta-models are defined using notations similar to class diagrams, often complemented by the use of constraint languages. These notations provide a visual representation of the entities, attributes and relationships involved in the meta-model. These visual representations help conveying the hierarchical structure and dependencies between elements.

Figure 2.2 shows an example of a meta-model of a *Library System*. The *Library System* is composed of a list of *User*s and *Book*s. *User* is an abstract class, indicating that it cannot be instantiated. It possesses attributes such as a name, a username, and a password. *User* can be either a *Librarian* or a *Member* of the library (subclasses of *User*).

A *Librarian* has a librarianID, and a *Member* has a memberID. *Book*s, on the other hand, define the attributes title, author, category, and isBorrowed. This last attribute is a derived attribute, implying that it is calculated automatically based on other attributes and relationships; in this case, from the "borrow" relationship. The category of a book can be one of the following: *FICTION*, *ROMANCE*, *HISTORY*, or *MYSTERY*. Additionally, *Librarian*s can add *Book*s, while *Member*s can borrow and reserve *Book*s.

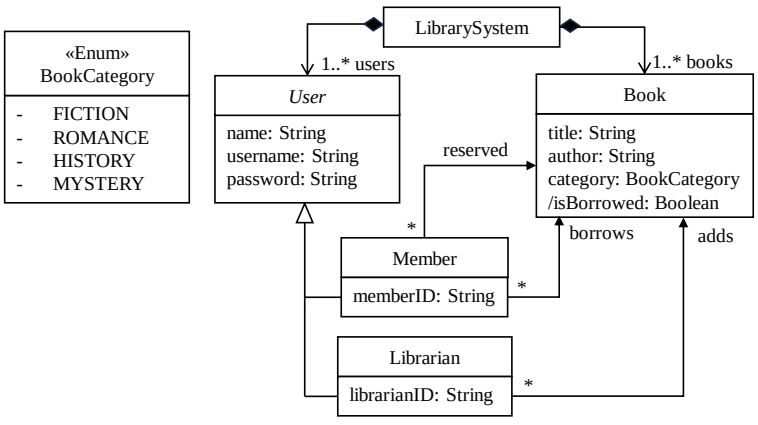


Figure 2.2: A meta-model example.

Model

A *model*, on the other hand, is an instance of a meta-model. It represents a specific system using the concepts defined in the meta-model. Models capture the structure, behaviour, and other characteristics of

the system, allowing for a more concrete representation of the intended system [25, 156].

Models should conform to their respective meta-models, ensuring that they preserve the structure and adhere to the rules defined within them. Figure 2.3 illustrates an example where a model conforms to the meta-model depicted in Figure 2.2. The *Library System* has four *Users*: one *Librarian* (Fredy) and 3 library *Members* (Lisa, Osmara, and Arsene) with their respective usernames, passwords and IDs. The librarian has added 3 books (Ender's Game, Wool, and 1491) with their corresponding information. Members have borrowed and reserved books. As an example, the first book's title is "Ender's Game", authored by Orson Scott Card and categorised as *FICTION*, and has been borrowed by one library *Member* (Lisa). Additionally, the same library *Member* has reserved the book titled "Wool", authored by Hugh Howey.

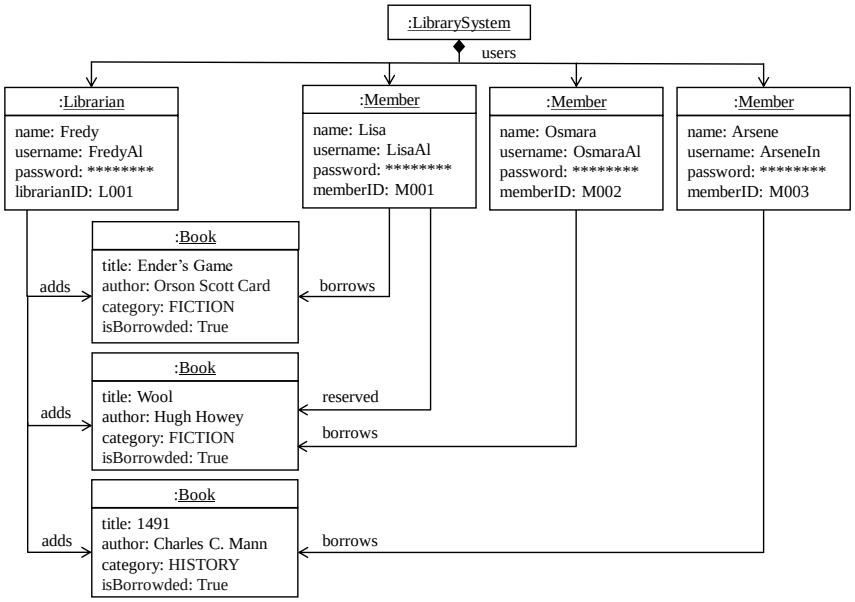


Figure 2.3: A model example instance.

Constraint language

On occasions, class diagrams may not provide sufficient expressiveness to specify intricate aspects of a system. In such cases, constraint languages allow specifying rules and restrictions that models created using the language must follow. These constraints ensure that the models are consistent, valid and semantically meaningful according to the intended semantics of the modelling language [25].

The meta-model of the Library System, for instance, lacks the ability to capture certain constraints. For instance, it fails to address the restriction that only *Book* objects that are currently borrowed can be reserved. Additionally, another constraint that needs to be considered is that the *Member* that reserves a book must be different from the one who has the book currently borrowed. To supplement meta-

models in addressing these concerns, the *Object Constraint Language* (OCL) [3] is utilised. OCL is a formal language employed to define conditions (referred to as invariants) that must be upheld for the model to be considered valid. OCL invariants are expressions that return a boolean value. An OCL invariant is defined in the context of a class, and evaluated on all instances on that class. This way, for a model to be considered valid, it must be structurally valid (i.e., it must conform to its meta-model), and in addition, it must satisfy all defined OCL invariants.

Object Constraint Language

As an example, Listing 2.1 shows the aforementioned constraints using OCL. The context (in this case, "Member") is specified after the "context" keyword, followed by the name of the invariant defined after the "inv" keyword ("reservationConstraint"). The keyword "self" refers to the object within the context, where the invariant is being evaluated. The "self.reserved" keyword verifies the member current status, while "->forAll()" means that the condition following this must be true for all elements in self.reserved.

Furthermore, "book.isBorrowed = true" signifies that for each reserved book, it checks that it is marked as borrowed (isBorrowed is true). Additionally, in"self.borrows->excludes(book)" the section "self.borrows" refers to the borrows property of the current instance of Member, which represents the books the member has borrowed. and "->excludes(book)" verifies that the book is not in the list of books the member has borrowed. Overall, this constraint ensures that, within the Library System, all books a member has reserved must be marked as borrowed, simultaneously confirming that these reserved books are not already in the list of books the member has borrowed.

```
1 context Member
2 inv reservationConstraint:
3   self.reserved->forAll(book | book.isBorrowed = true and self.borrows->excludes(book))
```

Listing 2.1: OCL constraints for the Library system meta-model.

The *Object Management Group* (OMG) [4] has proposed a 4-layer infrastructure that outlines a meta-modelling hierarchy that encompasses from languages to describe meta-models, to the actual systems. Figure 2.4 shows the proposed infrastructure. Within this framework, a meta-model, being a model itself, requires a modelling language to describe it, referred to as a meta-meta-model (layer 3). Thus, a meta-meta-model describes the set of all meta-models that can be described with it. The *Meta Object Facility* (MOF) [5] is the meta-meta-model set by the OMG. MOF stands as the highest level since it defines itself, having no level above it. Layer 2 encompasses the meta-models

Object Management Group

Meta-Object Facility

3 http://www.omg.org/spec/OCL/
4 https://www.omg.org/
5 https://www.omg.org/mof

created using MOF, encompassing both DSL meta-models and GPL meta-models. Layer 1 is where the models, which are instances of the meta-models, are situated. Finally, layer 0 contains the real systems or application data, which are instances of the models [156].

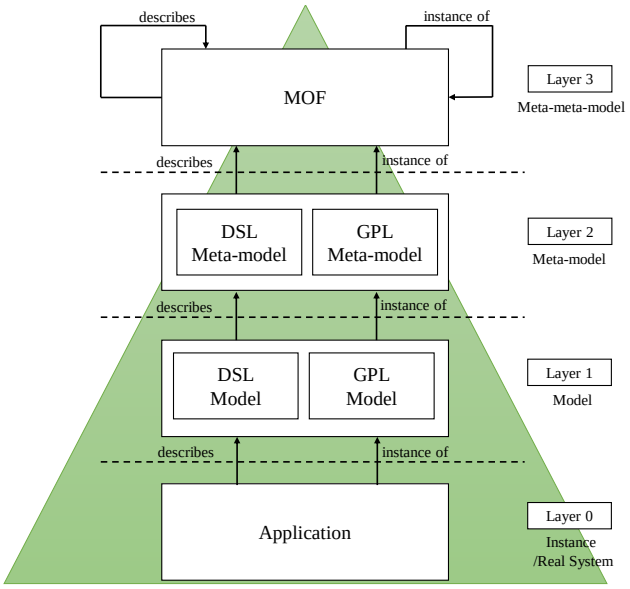


Figure 2.4: OMG 4-layer architecture.

2.2.3 *Concrete syntax*

Concrete syntax

The *Concrete syntax* plays an important role in defining the representation of a model and can encompass different forms, such as graphical, textual or tabular formats [25]. The elements of the concrete syntax are defined very carefully and are linked with elements of the abstract syntax. The concrete syntax should take into account specific requirements as well as the intended audience.

As we have mentioned above, the abstract syntax describes the structure of a model specifying the elements that can exist, how they relate to each other, and which are their functions. Then, the concrete syntax is described by mapping the modelling concepts described in the meta-model to their visual representations. In this manner, the visual notations introduce symbols for the modelling concepts. The assigned symbols for the modelling concepts allow visualising the models by using textual or graphical elements [25].

If the concrete syntax is textual, its elements may include keywords, identifiers, and punctuation and other textual symbols used to represent model components. Meanwhile, graphical concrete syntax elements may consist of shapes, lines, colours and icons. Having a textual language allows encoding information through sequences of characters, as is often seen in programming languages. Instead, graphical languages encode the information using spatial arrange-

ments of graphical elements [23, 25]. Further, integrity constraints are often introduced in the context of the concrete syntax to guarantee that the representation remains consistent, accurate and valid. These constraints allow maintaining the quality and correctness of instance models.

As an example, Listing 2.2 illustrates a textual concrete syntax used to represent the model of Figure 2.3. In this case, the concrete syntax elements in the listing are directly associated with abstract syntax elements, and the rules for creating and organising them are defined in the modelling language [23].

```
LibrarySystem {
  Users:
    Librarian Fredy ("L001") {
      adds ''Ender's Game'' {
        author: ''Orson Scott Card"
        category: Fiction
      }
      adds ''Wool'' {
        author: ''Hugh Howey"
        category: Fiction
      }
      adds ''1491'' {
        author: ''Charles C. Mann"
        category: History
      }
    }

    Member Lisa ("M001") {
      borrows ''Ender's Game"
      reserves ''Wool"
    }

    Member Osmara ("M002") {
      borrows ''Wool"
    }

    Member Arsene ("M003"){
      borrows ''1491"
    }
  }
```

Listing 2.2: An example of the textual concrete syntax for the Library system model.

2.2.4 *Semantics*

Semantics

The semantics of a modelling language defines the meaning of the models that belong to the language. This is often achieved through transformations that map models into another domain. These transformations are generally classified into: *Model-to-Model (M2M), Model-to-Text (M2T), and Text-to-Model (T2M) transformations*.

Model-to-model transformation

A *M2M* transformation is a program that transforms one model into another model. The initial model is called the source model, while the resulting model is referred to as the target model. More gen-

erally, M2M transformations can also be applicable to *multiple source models*, such as in the case of model merging, or *multiple target models*, such as in the case of transforming a platform-independent model into several platform-specific models. In addition, M2M transformations can be further classified attending to the following features [102, 25]:

- *Source and target languages.* Based on the modelling language in which the source and target models are expressed, M2M transformations can be classified as *endogenous*, when the models are expressed in the same language, or *exogenous*, when they are expressed in different languages. For instance, endogenous transformations include processes like refactoring or optimisation. Conversely, exogenous transformations involve operations such as reverse engineering or migrations.
- *Abstraction level of models.* M2M transformations can also be classified into *horizontal*, if the source and target models reside at the same level of abstraction, or *vertical*, if they reside at different levels of abstraction. Common examples of horizontal transformations include refactorings (an endogenous transformation) and migrations (an exogenous transformation). In contrast, refinements are an example of vertical transformations.
- *Technical space.* A technical space refers to a model management framework that encompasses concepts, tools, mechanisms, techniques, languages, and formalisms associated with a particular technology. Given this, source and target models for a model transformation may belong to the *same technical space* or to *different technical spaces*.

Model-to-text transformation

M2T transformations, on the other hand, produce text from models. This type of transformations typically generate source code in a programming language, documentation or configuration files. If the transformation specifically generates source code, it is then known as a code generator.

Text-to-model transformation

Finally, *T2M* transformations are typically used for reverse engineering such as transforming source code into a model.

2.2.5 *Technologies*

Eclipse Modeling Framework

While there are numerous technologies available for MDE, the *Eclipse Modelling Framework* (EMF) [158] is today's *de facto* standard for modelling in the Eclipse ecosystem. This is the reason why EMF has been chosen to implement the ideas proposed in this thesis (cf. Chapter 5).

EMF is a modelling framework that leverages the capabilities offered by the Eclipse platform. *Eclipse* [6] is an open-source integrated

6 https://www.eclipse.org/

development environment. Its architecture is based on Equinox, an implementation of the *Open Service Gateway Initiative* (OSGi) core framework specification whereby each unit of functionality in Eclipse is defined and deployed as a plug-in. This makes Eclipse a highly extensible environment. A plug-in is a component that extends Eclipse with new functionality. Many widely-used modelling languages and tools have been developed as plug-ins for Eclipse, and the most prominent ones have been gathered under the Eclipse Modeling Project. This project serves as the central hub for the evolution and promotion of model-based development. At the core of the Eclipse Modeling Project is EMF.

EMF is a modelling framework and code generation facility upon which most modelling-based tools in Eclipse build on. In EMF, *Ecore* allows the definition of domain-specific meta-models, which can then be used to create models conforming to those meta-models. For seamless interoperability between various modelling tools and platforms, XML Metadata Interchange (*XMI*) serves as a standardised representation for exchanging metadata information, including models. It offers a format specifically designed for serialising Ecore models into Extensible Markup Language (*XML*)-based representations [23, 158].

Ecore

In the realm of software development, specialised tools and languages play a crucial role in efficiently addressing specific problem domains. In the Eclipse ecosystem, we can find tools that facilitate the definition of textual DSLs, such as *Xtext* [7], and of graphical DSLs, such as Sirius [8]. *Sirius* is an open-source software project and a popular Eclipse-based tool for creating graphical DSLs, allowing users to create custom graphical modelling languages. Outside Eclipse, JetBrains MPS [9] is a powerful tool for defining textual DSLs using projectional editing, which permits overcoming the limitations of language parsers and constructing DSL editors.

For M2M transformations, there are eclipse-based solutions such as Atlas Transformation Language (ATL) [79] and Henshin [10]. ATL is a M2M transformation language and toolkit that enables developers to define transformations. ATL provides ways to produce a set of target models from a set of source models. In turn, Henshin is a model transformation language supporting both endogenous and exogenous transformations, and with a formal graph transformation semantics. Regarding M2T/T2M transformations, we find languages such as Acceleo [11] and *Xtend* [12]. Acceleo is an open source code generation language and widely used tool for M2T transformation. It allows using a model-driven approach to building applications.

7 https://www.eclipse.org/Xtext/
8 https://www.eclipse.org/sirius/
9 https://www.jetbrains.com/mps/
10 https://projects.eclipse.org/projects/modeling.emft.henshin
11 https://www.eclipse.org/acceleo/
12 https://eclipse.dev/Xtext/xtend/

Xtend [13] is a general-purpose language with template facilities for M2T transformations. Finally, Epsilon [14] comprises a family of languages and tools for model management and code generation, thus providing support for M2M and M2T transformations.

In this section, the emphasis is on introducing the two specific technologies that will be used for the implementation of the proposed approach, namely, *Xtext* and *Xtend*.

Xtext

Xtext is an open-source framework in Eclipse that provides a comprehensive solution for implementing textual DSLs. Given a specification of the DSL grammar, which can be either created from scratch or from an Ecore meta-model, *Xtext* generates a complete language infrastructure, encompassing various components such as the language parser, code generator or interpreter, and seamless integration within the Eclipse IDE. With *Xtext*, developers can create DSLs with ease and efficiency. The DSLs created with this framework can feature essential IDE functionalities like syntax-highlighting, code completion, error markers, automatic build infrastructure, and more, enhancing the development experience [23].

Xtend

On the other hand, *Xtend* is a powerful GPL language that closely resembles Java and offers seamless interoperability with it. It is designed to facilitate writing code generators, model transformations, and other tasks related to MDE. *Xtend* introduces advanced features such as type inference, lambda expressions, extension methods, and template expressions, which promotes conciseness and expressiveness in code. With its familiar syntax and additional capabilities, *Xtend* provides developers with a flexible and efficient language for tackling complex software development challenges [23].

2.2.6 *Feature models*

In the following chapters, feature models will sometimes be used to categorise the orthogonal features of a class of software systems. In the context of software development, a feature model works as a precise depiction of all the products within a Software Product Line (SPL). The feature model describes all these products in terms of their distinctive "features" and are generally visualised through feature diagrams [113]. The relationships between a parent feature and its child features (or sub-features) can be categorised as follows:

Feature models

- Mandatory: Selecting the parent feature requires the selection of the child feature.
- Optional: If the parent feature is selected, the child feature can be selected or not selected.

13 https://eclipse.dev/Xtext/xtend/
14 https://eclipse.dev/epsilon/

- Or: Selecting the parent features requires the selection of at least one of its sub-features.
- Alternative (xor): Selecting the parent feature demands the selection of exactly one of its sub-features.

Besides to the parental relationships between features, another available option is the use of cross-tree constraints. The most common cross-tree constraints are:

- A requires B: Choosing feature A for a product means that feature B must also be selected.
- A excludes B: Features A and B cannot coexist within the same product.

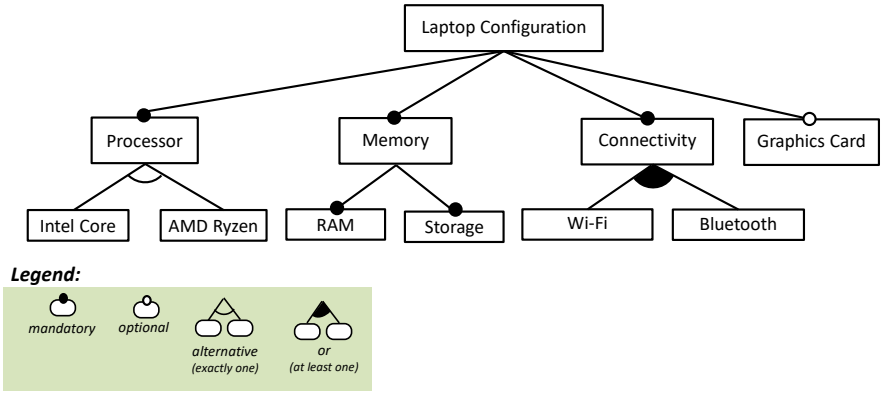


Figure 2.5: Feature model example.

For instance, in Figure 2.5, we illustrate the use of feature models for specifying and constructing configurable laptop configurations, and their representation as feature diagrams. The legend in the lower part of the figure indicates how the different feature relationships are represented. In the figure, the composition of each laptop configuration is determined by the features it incorporates. The root feature "Laptop Configuration" defines the SPL, and encompasses the mandatory implementation of Processor, Memory, and Connectivity, as well as optionally including a Graphics Card. The Processor must be either an Intel Core or an AMD Ryzen (exactly one choice must be made). The laptop also offers different Memory capacities, including mandatory implementation of RAM and Storage. Finally, the Connectivity can be Wi-Fi or Bluetooth (at least one choice must be made).

2.3 SUMMARY

RSs alleviate information overload by suggesting items based on user preferences, playing an important role in numerous applications. RSs

can be categorised into three main types: content-based, collaborative filtering, and knowledge-based systems. Additional categories include demographic-based, social-based, and hybrid systems. RSs are evaluated using diverse methodologies and metrics, including offline and online experiments, user studies, and metrics like MAE, RMSE, precision, recall, F1, nDCG, and novelty and coverage metrics. Moreover, rank aggregation methods refine suggestions by merging rankings from different systems, enhancing both accuracy and diversity.

Transitioning to MDE, this approach places models at the core of software development to increase abstraction and reduce costs throughout the entire development life cycle. MDE relies on the use of DSLs with abstract syntax, concrete syntax, and semantics. Models, representing specific systems, conform to their respective metamodels, which ensures their structural consistency. Constraint languages like OCL permit defining rules to maintain model validity.

In the Eclipse ecosystem, EMF has emerged as the de-facto standard for modelling, utilising Ecore to build meta-models, and XMI as a standardised format for models. Eclipse provides various tools working on EMF models, like Xtext and Sirius to specify their concrete syntax; ATL, Henshin, and ETL for model transformation; and Acceleo, Xtend and EGL for code generation.

Feature models become instrumental in software development, specifically for SPL development. They permit defining features and their relationships (like mandatory, optional, or, xor) along with cross-tree constraints like requires and excludes.

3

STATE-OF-THE-ART

This chapter provides a comprehensive overview of existing developments on RSs in the field of MDE. First, in Section 3.1, we survey RSs developed to assist in modelling tasks. Afterwards, in Section 3.2, we survey approaches to automatic generation of RSs. Finally, in Section 3.3, we summarise the chapter.

3.1 RECOMMENDER SYSTEMS IN MODELLING

Recent research shows that the modelling community is becoming increasingly interested in developing RSs for modelling tasks. In accordance with established guidelines for systematic mappings [125, 126], we conducted a comprehensive survey to evaluate the extent of usage of RSs for supporting modelling and MDE tasks. The study identified the addressed tasks and the most commonly used recommendation techniques. The majority of the analysed papers present RSs that facilitate some form of modelling or MDE task. Our survey [10] involved characterising, categorising and analysing existing research on RSs for MDE. Subsequently, we expanded the study by incorporating recent work in the field.

Scope of the search

To retrieve relevant papers for our analysis, we conducted searches on Scopus, ACM and Web of Science digital libraries using a formal query consisting of 23 terms. The query retrieved papers whose titles, abstracts or keywords included at least one term related to RSs and one term related to modelling or MDE.

The terms considered for the retrieval of the papers are shown in Table 3.1, requiring that each column of the table must have at least one term present in the title, abstract or keywords of every retrieved paper. We included terms like *model completion*, *model reuse* and *model repair* to the list, which are relevant but are not commonly used in the standard terminology and vocabulary. These terms, however, share the goal of recommending a concise selection of modelling items or actions from a large set of possibilities. Our initial search was conducted in September 2020 whereas our second search was in February 2023. The searches were limited to peer-reviewed papers written in English and published in journals, conferences, workshops and book chapters.

Table 3.2 shows the statistics of the search results. Initially, the query yielded 1,463 papers, comprising 980 from Scopus, 316 from the ACM digital library, and 167 from the Web of Science. After re-

Table 3.1: Terms used in the formal search query. Articles must contain in their title, abstract or keywords at least one term from each column.

Recommender Systems	Modelling / MDE
recommender	model-driven
recommendation	domain-specific language
model completion	state machine
model reuse	model transformation
model repair	code generation
transformation completion	code generator
transformation reuse	unified modelling language
transformation repair	UML
generator completion	
generator reuse	
generator repair	
quick fix	
quick fixes	
assistant	
assistance	

moving duplicates, the number of distinct documents was reduced to 1,182.

First phase of revision

Once the papers were retrieved, a two-phase filtering process was conducted to identify unique documents that propose modelling task recommendations. In the first phase, four reviewers, including two MDE professors, one RS and information retrieval professor, and one doctoral student with expertise in both areas, examined the abstracts of all documents. Overall, 158 documents were selected for the next phase, and 1,024 documents were discarded for not meeting the inclusion criteria based on quality, language and focus. These inclusion criteria required the papers to be peer-reviewed, written in English, and related to modelling, recommenders or modelling assistants, while excluding papers that focus on using MDE techniques for creating RSs.

Second phase of revision

In the second phase of the filtering process, the 158 selected papers were read, and 60 were considered relevant for the study, while 9 were unavailable and 89 were not relevant. A snowballing process was then performed by analysing related work in the bibliography of the selected papers, resulting in the selection of 19 additional relevant papers. The final set included 79 papers published between 2004 and February 2023, covering over 19 years of research and account-

Table 3.2: Research papers retrieved per database.

Detail	Num. Papers
Databases queried	
Scopus	980
ACM	316
Web of Science	167
First revision phase	
Total retrieved	1,463
Unique	1,182
Discarded	1,024
First selection	158
Second revision phase	
Relevant	60
Not available	9
Not relevant	89
Snowballing	
Snowballing papers	19
Total relevant	
Total	79

ing for 59 different approaches. The distribution of papers over time is depicted in Figure 3.1.

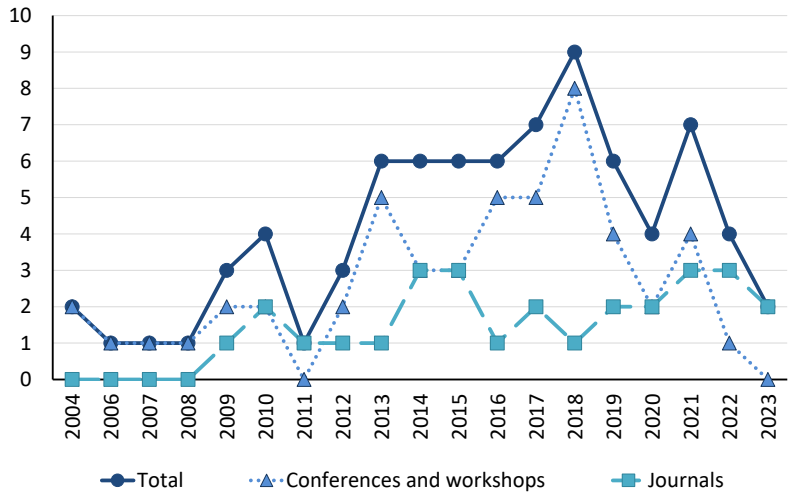


Figure 3.1: Relevant papers per year.

Our analysis is structured according to the four dimensions of the feature model [82] shown in Figure 3.2: the *domain*, *tooling*, *recommendation* approach, and *evaluation* methods and metrics. In the follow-

ing four subsections, we will characterise and categorise the chosen papers according to these dimensions. The legend displayed in the figure will be used in the feature models presented in the next subsections. Finally, in Subsection 3.1.5, general conclusions and opportunities are discussed.

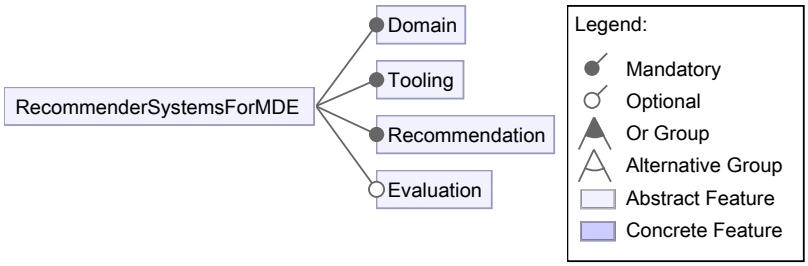


Figure 3.2: Dimensions for analysing the use of RSs in MDE.

3.1.1 *Domain*

The application domain is a crucial element of any recommender system, encompassing the three independent features outlined in the feature model shown in Figure 3.3.

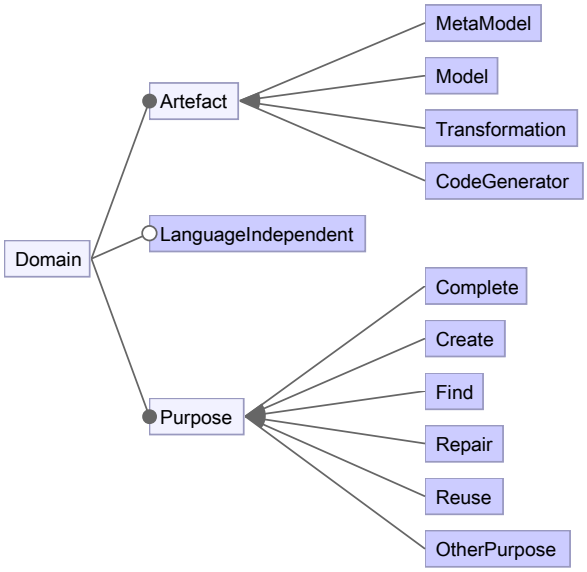


Figure 3.3: Domain dimensions for RSs in MDE.

Artefact type

First, we consider the type of artefact that is the subject of recommendation: *model*, *meta-model*, model *transformation* or *code generator*. These are the four main elements of most MDE solutions [25].

Language-independent

Second, we distinguish whether the RS is *language-independent* or, on the contrary, it is tied to a particular language for modelling (such as Simulink [15], UML, meta-modelling (such as MOF, Ecore), model

15 https://www.mathworks.com/products/simulink.html

transformation (such as ATL, QVT [16]), or code generation (such as Acceleo, EGL [138]).

Purpose of the RS

Finally, we look at the purpose of the RS, that is, the kind of task that the recommender facilitates. The surveyed papers target one or more of the following six types of tasks: *complete*, *create*, *find*, *repair*, *reuse* and *other purpose*. When the purpose is *complete*, the artefact already exists and the RS provides suggestions on how to extend it. When the purpose is *create*, the recommender helps in constructing the initial version of a new artefact from scratch. If the purpose is *find*, the RS facilitates the discovery of relevant elements or artefacts within a repository. Recommenders targeting *repair* tasks suggest solutions to fix errors in an existing artefact. These solutions may imply the creation, deletion or modification of different elements inside the artefact. When the purpose is *reuse*, the system helps in reusing an existing artefact (or part of it) within another artefact. This task goes beyond *find* as the recommender provides assistance in integrating the reused artefact in the new context. Finally, in the *other purpose* category, we collect the tasks with a purpose different from the mentioned before.

Table 11.1 in the Appendix A classifies the surveyed papers by purpose and artefact type, and marks the language-independent approaches with an asterisk (*). Since some systems can be used with various purposes, they can appear in several cells of the table, sometimes with different language-independent marks.

Overall, there are virtually no recommenders for code generators, and recommenders that help in creating new artefacts from scratch are scarce. Most RSs are targeted to models, especially for model completion and model repair tasks, and the context in the latter case is sometimes model/meta-model co-evolution. In the following, we analyse the application domain of the approaches grouped by their purpose.

Complete

complete. To complete an artefact, most approaches target model completion, and six of them are also applicable to meta-models. Besides, three approaches focus exclusively on meta-models, while two approaches deal with completing model transformations.

Approaches to model completion can be classified into two categories. The first one comprises techniques to recommend how to extend a partially specified model to make it correct (i.e., the recommended complete model satisfies every specified domain and meta-model well-formedness constraint). The proposed model completions are typically computed using *search-based* techniques, for example, with solvers based on Alloy [76] (a constraint solver over models), Prolog (a logic-based programming language) or via rules. DIG MDE [111], IPSE [63], Kermeta [105], Refacola [157], DiaGen [101]

16 https://www.omg.org/spec/QVT

and the works by Nair et al. [15] and Sen et al. [151, 152] belong to this category. Nair et al. use ensemble learning of recommender parameters based on a priority-based weight assignment, DiaGen is based on hyper-graph grammar rules, while the others use Prolog or Alloy for this task. This search may have a high computational cost. For this reason, when a partial model cannot be completed automatically because of its complexity, DIG MDE identifies the failing constraints and suggests how to manually change the model to enable its completion. MORGAN [43] helps with the specification of models by using an encoded graph-based format that leverages Natural Language Processing (NLP) to complete partially specified models. NEMO [44] assists modellers with performing model editing operations using an Encoder-Decoder Neural Network. Finally, Shilov et al. [154] assist in the enterprise modelling process by providing recommendations for the creation of process objects (e.g., concepts, resources, processes...) and their relationships with the use of Graph Neural Networks (GNNs).

The second model completion category comprises works providing step-wise recommendations on how to evolve a given model, which does not need to be partial, as in the first category. Suggestions usually come from repositories of existing models, fragments or patterns. For instance, SimIMA [3, 159] provides step-wise suggestions to evolve Simulink models based on model clone analysis; Heinemann [69] recommends elements defined in model libraries (e.g., blocks from Simulink libraries) based on data mining existing models; the approach by Kögel et al. [86, 87] recommends model changes applicable to the same context of the last model change; DoMoRe [7, 8] suggests domain concepts and names for new model elements; RapMOD [90, 91, 109] offers auto-completion actions for (UML) graphical models, similarly to the vision paper [146]; Elkamel et al. [52] recommend UML classes that are similar to the ones in the UML class diagram being developed; Li et al. [94] and Deng et al. [39] recommend activity nodes for process models; Rangiha et al. [129] recommend tasks and actor roles in a social process-modelling tool; Koschmider et al. [71, 72, 89] recommend process fragments to complete a process model; Baya [37] recommends mashup model patterns based on the context, the user and different expert recommendations, and helps in weaving the selected pattern into the partial model under development; and Hermes [49, 50, 51] allows building Eclipse-based RSs that help in completing models with recommended elements from other models in a repository.

Instead of profiting from repositories of models, PME's recommendations are based on an analysis of the language meta-model [119]. PME enables proactive (graphical) modelling, meaning that plausible modifications according to the models' meta-model are automatically applied, and the user is prompted only when several optional

modifications exist. SMART [67] supports the use of test-driven development to create UML diagrams (class, use cases, state machines and sequence diagrams). It uses an action language to specify behavioural tests, and when a test fails, it suggests ways to complete the model to make it pass the test.

Among the previous model completion approaches, six can also be applied to meta-models. DoMoRe works on domain models, like UML class diagrams and entity-relationship diagrams, and therefore can be used to add concepts of a domain of interest to meta-models. The approach of Kögel et al. recommends complementary changes to user editing action, and can be applied at the meta-model level, e.g., to recommend generalisation relations to a core super-class [87]. Refacola is a refactoring constraint language and framework, extended to (meta)model assistance in [157]. It is meta-level independent, providing assistance for completing partial (meta)models to become syntactically correct. The Hermes framework can be configured with recommendation strategies. It is applicable to models within the EMF ecosystem, and hence to meta-models as well. MORGAN and NEMO also help with the specification of meta-models by using the same mechanism.

There are approaches that focus exclusively on meta-models. Weyssow et al. [167] apply a data-driven approach that utilises a deep learning model to extract domain concepts. The model learns meta-model properties from a corpus of independent meta-models to abstract domain concepts. MemoRec [41] provides an effective means of completing meta-model specifications by utilising four encoding schemes to represent both the meta-models and their corresponding contents. Lastly, Burgueño et al. [28] provide autocomplete suggestions based on an automatic analysis of textual information using NLP techniques.

Regarding the completion of model transformations, CONVErT [17] synthesises transformation code starting from examples of source and target models and their correspondences. Correspondences are specified manually, but there is also a recommender of likely correspondences based on similarity heuristics such as the name, structure and neighbourhood of model elements. Finally, AXSM [73] is a mapping recommender integrated in a tool to build data transformations via declarative mappings, from which translators written in XSLT, Java or ATL can be synthesised. AXSM recommends potential mappings based on heuristics grounded on the data schemas and on prior user selections.

Create

create. Only three of the analysed works target the creation of artefacts, one for meta-models, one for models and one for transformations. The one for meta-model creation is DSL-maps [124]. Given the requirements of a DSL expressed as a mind-map, DSL-maps recom-

mends meta-modelling patterns addressing them. The designer can select patterns among the ranked suggestions, and the tool combines the patterns to synthesise an initial meta-model, which the designer can then refine.

The approach for modelling assistance for use case diagrams is called UCcheck [14]. This tool has a wizard to create new use case diagrams using an existing one as a reference, from which suitable actors and use cases are recommended. The last one, OCKHAM [160], utilises frequent subgraph mining on labelled graph representations of model differences to learn edit operations from model histories in repositories. It discovers frequent patterns in the model differences, filters and ranks them based on a compression metric, and generates a list of recommendations for meaningful edit operations.

Find

find. The analysed papers include approaches to query repositories, and suggest relevant artefacts for models and meta-models, but not for transformations or code generators. Extremo [148, 149, 150] is an extensible tool-independent assistant that helps finding relevant information for models and meta-models out of heterogeneous data sources (e.g., ontologies, XML schemas, RDF data, meta-models), and the results are ranked according to their suitability for the user. The rest of approaches are specific for some kind of model: the RS of Cerqueira et al. [36] finds and recommends sequence diagrams that match the user preferences; Matikainen et al. [100] tackle the problem of recommending the state machine from a library that implements the best policy to control a robot; and SBPR [83, 84] recommends process models from a repository according to the user business profile in LinkedIn (e.g., skills, interests and current position).

Repair

repair. Repair approaches have been proposed for all kinds of artefacts: models, meta-models, transformations and code generators. Regarding model repair, most works aim to recommend fixes for inconsistencies found in a given model (i.e., violations of the model's meta-model cardinality or well-formedness constraints). These approaches differ either in the applied technique to compute and rank the repairs, or in the application domain. In particular, IntellEdit [112] ranks quick fix solutions to model inconsistency problems according to the least-change principle; PARMOREL [18, 19, 74] determines the model repair actions based on the user preferences and on the experience gained from repairing under different personalisation settings; the Diagram Predicate Framework (DPF) [128] and the approach by Nassar et al. [110] implement repairs as transformation rules; DiaGen [101] represents models as hyper-graphs and uses hypergraph patches to produce recommendations for repairing models; Refacola [157] uses constraint-based rules; BPMoQualAssess [80] provides guidelines to improve the actual value of qual-

ity metrics for business process models; B-repair [31] is specific to the B formal specification language and ranks the suggested repairs based on their estimated quality; ReVision [115, 116] tracks model inconsistencies to the editing action originating them in the model history, and fixes this action to obtain a consistent model; MDSafeCer [106] detects missing information for supporting key evidence in process-based argumentations, and recommends how to resolve such deviations; ASIMOV [61] assists in the co-evolution of models and meta-models by proposing model co-evolution actions that a meta-modeller must have defined previously; Anguel et al. [13] also tackle the co-evolution problem but they automatically fix resolvable changes and recommend co-evolution actions to deal with non-resolvable changes; and Shilov et al. [154] identify likely wrong edges, types or labels and suggest a better fitting option.

There are also some model repair approaches that do not tackle model conformance, but target other kinds of model-related problems. In particular, Mani et al. [99] compute repairs for input test models that make a code generator produce an incorrect output; in addition to *complete*, the suggestions for fixing behavioural tests in SMART [67] can also be classified as repairs; the Business Application Modeller (BAM) [168] allows specifying temporal rules for process models and, for some types of rules, it recommends how to fix their violations; and AMOR [27] is a model repository for model versioning that includes a recommender of possible resolutions for model conflicts.

With respect to meta-model repair, two of the works target OCL integrity constraints [21, 38]. Batot et al. [21] tackle the co-evolution of OCL constraints upon meta-model changes. Their approach recommends a ranked list of OCL modifications that are correctly typed by the new meta-model version, and minimise the number of changes and information loss. Clarisó et al. [38] repair OCL constraints which are too restrictive or too lax. Their method suggests weaker or stronger candidate versions of the problematic constraint, and the user can select one of them. In addition, two of the model repair approaches can be used to repair meta-models as well. PARMOREL allows repairing meta-models having duplicate attributes in related classes, or properties modelled both as attributes and as references [19]. Refacola [157], on the other hand, can help repairing syntactically incorrect meta-models, e.g., with inconsistent opposite or containment references (typical problems at the model level that can also happen in meta-models).

We found only one work supporting model transformation repair. This is anATLyzer [142, 143, 144], a tool integrated with the ATL IDE that identifies errors and recommends a ranked list of quick fixes to (syntactically) repair the transformation. Fixes are ranked

taking into account the number of problems they solve, remaining errors and newly introduced errors.

Finally, we classify the approach by Mani et al. [99] as applicable to code generators because even if it suggests model repairs, these are applied in the context of code generation with XSLT.

Reuse

reuse. Recommenders in MDE have been applied to the reuse of models and transformations. Regarding model reuse, SimIMA [3, 159] recommends Simulink models similar to the one that is being developed, and which the designer can import or clone for their reuse; REBUILDER [64] finds UML diagrams similar to a given query, and supports their full or partial composition into the given design; Paydar et al. [120, 121] propose a reuse technique whereby the designer provides an input UML use case diagram, the most similar use cases are retrieved from a model repository, and then the activity diagrams associated to these use cases are semi-automatically adapted to (i.e., reused in) the new usage context; Koschmider et al. [71, 72, 89] propose both a recommender of process model fragments, and an explicit search facility to retrieve complete process models or fragments and insert them in the current modelling context, adapting them if needed; and Hermes [49, 50, 51] can incorporate model search strategies to find model elements suitable for reuse. Being generic, Hermes can also be applied to meta-models.

As for transformation reuse, it is supported by Refactory [131]. This tool allows defining generic refactorings over role models, so that developers can reuse the refactorings on new languages by binding the role model elements into elements of the language meta-model. Refactory includes a recommender that helps in identifying possible bindings, likely starting from some manually bound elements to avoid a high number of suggestions.

other purpose. The remaining papers have very specialised purposes. MAGNET [1] guides users on the next tutorials to speed up the learning curve of a modelling tool. ModBud [140] is an envisioned framework to build assistants that educate novice modellers on abstraction. Such assistants may provide recommendations on a constructed model by comparison with a prescriptive model devised by the assistant. Finally, Bobek et al. [24] propose a recommender for process modelling, which suggests the elements of a configurable diagram (i.e., a process with variability) that should be included in the current modelled process.

3.1.2 *Tooling*

Next, we analyse the tool support of the approaches using the criteria shown in the feature model of Figure 3.4.

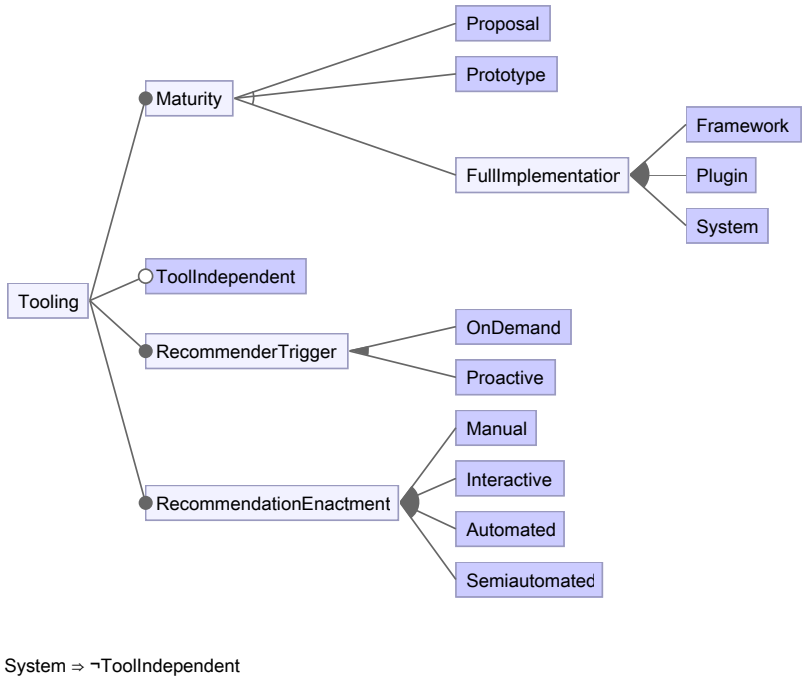


Figure 3.4: Tooling dimensions for RSs in MDE.

First, we look at the maturity of the supporting tools. We distinguish between *proposals* with no implementation, *prototypes* built as proof-of-concepts of the proposed ideas, and mature tools that make a full implementation available either as a *framework*, a *plugin* or a *system*. Frameworks typically offer generic functionality that can be customised by manually written code (e.g., by subclassing). Plugins encapsulate functionality that complements other tools, such as the Eclipse IDE. Systems can be either complete new applications that incorporate recommendation facilities, or extensions of existing MDE tools with a RS.

Second, we classify the approaches as *tool-independent* if they can complement or be integrated into other MDE tools. The constraint in the feature model states that systems cannot be tool-independent, since the RSs are embedded in the tools themselves.

Third, RSs may trigger recommendations *on demand*, *proactively* or both. In the first case, the user needs to explicitly start the recommendation process. In the latter case, the RS makes recommendations without user intervention, when certain conditions are met.

Finally, we analyse the support for enacting the recommendations. This can be *manual* if the RS provides a list of recommendations, and it is up to the user to decide how to use them; *interactive* if the RS allow the user to select a recommendation, which then becomes applied to the given context; *automated* if the recommendation is automatically applied without user intervention; and *semiautomated* if the recommendation enactment is automated, but the user may be prompted

during the process, e.g., to input some value or decide between alternative options.

Tables 11.2 and 11.3 in the Appendix A classify the revised papers according to these features. In the following, we discuss the different approaches attending to their maturity level, tool independence, recommender trigger, and recommendation enactment.

Maturity of the tool

maturity. The second column of Tables 11.2 and 11.3 shows the maturity level of the approaches. When there are several incremental papers on the same approach, the column only shows the maturity achieved in the latest one (i.e., the highest maturity level). Among the 59 approaches, 3 (5.8%) are proposals with no implementation, 26 (44%) present prototypes as proof-of-concept, and the remaining 30 (50.85%) provide full implementations. Most full implementations are either plug-ins (14) or systems/extensions of systems (13), while frameworks (3) are less pervasive. We categorise the Hermes [49, 50, 51] framework for developing RSs as a plug-in as well, as it uses a plug-in architecture that exposes and profits from Eclipse extension points.

Tool independence

tool independence. The third column of Tables 11.2 and 11.3 shows the independence from tools of the approaches. This feature applies to approaches that make a prototype or full implementation available, but not to proposals that have not been realised in practice (even though they may have the potential to become tool-independent). Most tool-supported RSs in MDE have been developed either as full software systems, or as extensions of the following existing systems: the ATL development environment [142, 143, 144], some data mashup tools [37], the Generic Eclipse Modeling System (GEMS) [111], the Ecore Diagram Editor [7, 8], the Diagram Predicate Framework (DPF) [128], DiaGen [101], Fujaba [63], AutoFOCUS3 [1], the AMASS platform [106], the AMOR model versioning system [27], Kermeta [105], the Generic Modeling Environment (GME) [15, 119], Sparx Enterprise Architect [90, 91], Papyrus [146, 147], Simulink [3, 159] and the meta-modelling tool AToM^3 [151, 152]. All these approaches built as complete systems or system extensions are tool-dependent. In some cases, the tools are implemented atop EMF to achieve generality. However, we only consider that an approach is tool-independent if, in addition, it provides explicit means to facilitate its integration with other tools. Under this perspective, only four approaches are truly independent from any modelling tool. We comment on these approaches next.

The framework developed by Batot et al. [21] recommends how to co-evolve OCL invariants upon Ecore meta-model changes (i.e., it is language-dependent); however, the framework is not specific for particular editors, and is extensible with new heuristics to guide the search of recommendations. Refacola [157] achieves tool indepen-

dence by being based on a domain-specific, constraint-based language to specify model-assistance operations. Extremo [148, 149, 150] is a modelling assistant that defines extension points (the extensibility mechanism provided by Eclipse) to allow its integration with external modelling and meta-modelling tools within Eclipse. Finally, Hermes [49, 50, 51] is not a concrete RS, but a framework with a plugin-based architecture to develop RSs within Eclipse. Its extension points allow defining new recommendation strategies and the integration with modelling editors and heterogeneous data repositories.

Other approaches can be used with several tools, but are still tool-dependent. This is the case of UCcheck [14], an assistant for use case diagrams coded in Python that supports use case diagrams specified with TTool – a free software from Telecom Paris – and the Cameo Systems Modeler.

Finally, several approaches have defined algorithms or techniques that can be applied to a wide range of modelling tools, such as Burgueño et al. [28], Weyssow et al. [167], MORGAN [43], MemoRec [41], NEMO [44] and OCKHAM [160]. It is important to note that these approaches do not provide explicit means for facilitating their integration with any modelling tool.

Recommender trigger

recommender trigger. As the fourth column of Tables 11.2 and 11.3 shows, most RSs provide recommendations on user demand (42 approaches out of 59). Fewer approaches provide recommendations proactively without user intervention (14 out of 59), typically by monitoring the user editing actions to update the recommendations in return. Only a few tools (3) can trigger the recommendations both on demand and proactively: the recommender of domain model elements DoMoRe [7, 8], the envisioned modelling learning environment ModBud [140], and lastly, the generic RS framework Hermes [49, 50, 51] supports both ways to trigger recommendations from modelling editors. Finally, MemoRec [41], MORGAN [43], NEMO [44], Shilov et al. [154], OCKHAM [160] and Weyssow et al. [167] do not give enough details on how to access the recommendations.

Recommendation enactment

recommendation enactment. The last four columns in Tables 11.2 and 11.3 show how the different works enact the recommendations. In most cases, recommendations can be applied either manually or interactively. Automated enactments typically occur in model completion and model repair. As an example, DIG MDE [111] automatically completes a model, and if this is not possible, it recommends the user how to fix the model manually. In turn, the tool by Nassar et al. [110] allows repairing models either automatically or interactively. Six approaches provide semiautomated enactment of recommendations: two are co-evolution approaches [13, 61] that automatically

infer and apply a migration strategy, but the user may need to select between alternative solution steps, e.g., in the case of non-resolvable changes; the others correspond to the proactive modelling approach in PME [119] and Nair et al. [15], where models are automatically modified according to the models' meta-model, and the user is only prompted if several optional modifications exist. Finally, since Hermes [49, 50, 51] is a framework to build RSs, it provides mechanisms to support all types of recommendation enactment.

3.1.3 *Recommendation*

MDE researchers have applied diverse recommendation approaches for a variety of tasks and purposes. In this subsection, we characterise, categorise and analyse the works on MDE recommenders according to the features shown in the diagram of Figure 3.5.

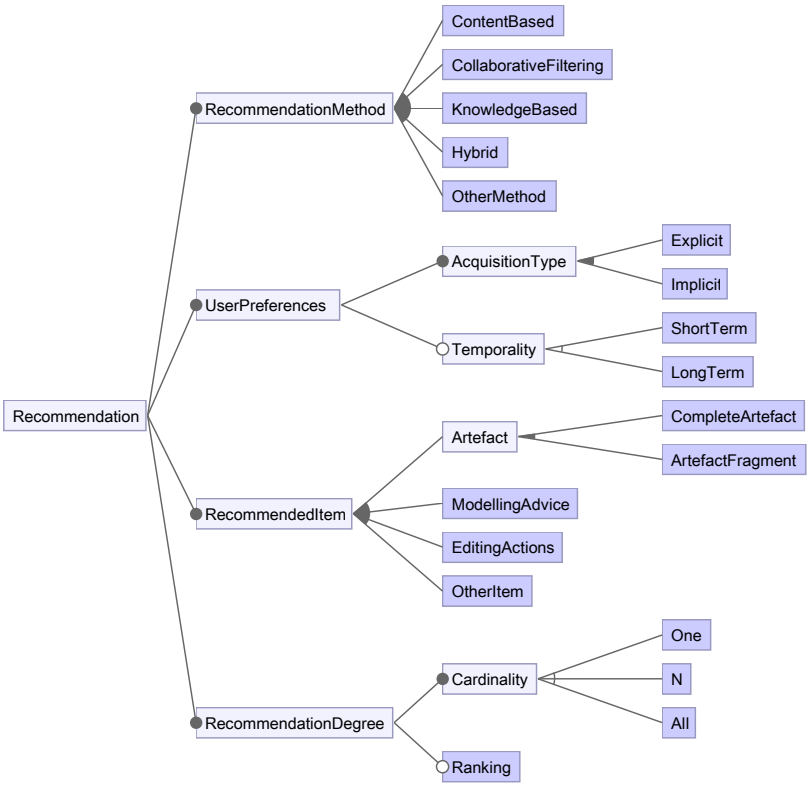


Figure 3.5: Recommendation dimensions for RSs in MDE.

As a first feature of analysis, we consider the recommendation method used. The majority of the RSs apply one of the four main techniques explained in Section 2.1: *content-based*, *collaborative filtering*, *knowledge-based* and *hybrid*. In addition, some works use ad-hoc techniques that do not fall into the previous categories. They are represented by *OtherMethod* in Figure 3.5.

Second, RSs collect user information to provide personalised recommendations (feature *UserPreferences* in the diagram). In this re-

spect, we investigate how this information is collected (feature *AcquisitionType*). In some cases, the user's preferences are gathered *implicitly* by monitoring the user's interactions with the system or analysing the current state of the modelling/MDE activity. In other cases, the user needs to *explicitly* provide her preferences to the system, for example via questionnaires. In addition, we examine the *temporality* of the collected preferences, which can reflect recent, likely temporal preferences for the task at hand (i.e., *ShortTerm*) or more general and enduring preferences (i.e., *LongTerm*).

Third, we analyse the types of items provided as recommendations (feature *RecommendedItem*). These can be *complete* artefacts (e.g., a model), *fragments* of an artefact (e.g., a class), *advices* that the user can profit from during a modelling activity, or *editing actions* (e.g., in the context of model repair). The diagram includes the *OtherItem* feature for items that do not fall in any of the previous categories.

Finally, the feature *RecommendationDegree* comprises the amount of recommendations presented to the user (*Cardinality*) and whether they are ranked (*Ranking*).

Table 11.5 in the Appendix categorises the surveyed approaches according to these features. Taking this categorisation into consideration, we start by analysing the approaches attending to the recommendation method they use, and then, we analyse them based on the other features.

Content-based recommenders

content-based. These approaches use different content encoding and similarity notions to represent and relate items for generating personalised recommendations.

First, we comment on the content-based approaches that recommend complete artefacts. Cerqueira et al. [36] compare two alternative encodings of sequence diagrams (*bag-of-words* and a vector encoding structural features) for the recommendation of sequence diagrams matching the user's preferences. The RS uses a content-based filtering algorithm to find the closest sequence diagrams.

The RS proposed by Paydar et al. [120, 121] facilitates the reuse of models with functional requirements of web applications. For this purpose, the system recommends similar use cases to the one provided by the user, and then adapts the activity diagrams linked to the selected use case to the provided one. Item similarities are computed based on name similarity of the use case elements and on the diagram context.

Similarity has also been exploited to recommend artefact fragments. For example, Elkamel et al. [52] use similarity metrics to suggest similar classes to newly created classes. The developer may accept the suggested classes with all or some of their attributes and methods. MORGAN [43] generates a textual representation of models, followed by encoding relevant features into a graph-based represen-

tation. A graph kernel function utilises the extracted graphs to offer modellers meaningful recommendations for completing partially specified models.

DoMoRe [7, 8] addresses the same problem by means of semantic similarities. It relies on an extensive knowledge base – called *SemNet* – of several million domain-specific terms and their relationships to provide context-sensitive recommendations. In particular, DoMoRe suggests names for new elements, and possible related concepts to the selected one (e.g., upon selecting a class, the system suggests possible sub-/superclasses, container and aggregated classes, and related and associated classes). Extremo [148, 149, 150] also employs semantic similarity based on WordNet to provide a ranked list of recommended model elements, upon an explicit query of the user.

Content-based similarity has been applied to transformation development as well. CONVErT [17] helps discovering and specifying transformation correspondences using concrete visualisations. A RS suggests mappings between source and target models based on different similarity heuristics, choosing mappings that resemble examples provided by the user. In a similar vein, AXSM [73] recommends mappings based on similarity criteria (source/target element tag names, element types, structural similarity, example data item equivalences) and previous user selections within the MaramaTorua tool. Refactory [131] supports the definition of generic refactorings over role models so that developers can reuse the refactorings on new languages by mapping the role model elements into elements of the language meta-model. The tool includes a RS to complete the mapping using structural similarity and other heuristics, like name similarity.

Collaborative filtering recommenders

collaborative filtering. These approaches exploit information about past behaviour or opinions from the user community [77]. In some cases, users correspond to developers for which personalised recommendations are generated, and in other cases, users (and items) are mapped to elements within the artefacts that are target of recommendations.

MAGNET [1], PARMOREL [18, 19, 74] and ModBud [140] belong to the first case. MAGNET is a RS within the AutoFOCUS3 modelling tool to help beginners to learn using the tool. It monitors user actions and proposes short videos illustrating what to do next. The recommendation model is based on data collected during a tutorial with a previous set of users. PARMOREL uses reinforcement learning to find a sequence of actions that repair the issues present in a model. The algorithm initially reuses the experience obtained from other users' repairs, and learns after each repair. ModBud is an envisioned framework to build modelling bots to assist novice users. The

authors foresee using machine learning to predict good modelling decisions for given design requirements.

Matikainen et al. [100] address the second case. Their RS selects the best-performing state machine to control a robotic vacuum cleaner. Room layouts are interpreted as users, robot state machines as items, and item ratings are based on the performance of the robot state machines on the room layouts.

Knowledge-based recommenders

knowledge-based. Most approaches belong to this category. They use techniques that can be generally classified as *constraint-based* or *case-based*. Constraint-based techniques determine the recommendations by looking for a set of items that fulfil established domain-dependent rules. Case-based techniques, by contrast, provide recommendations to a problem by examining past solutions for alike problems (cases) [77].

Some of the constraint-based recommenders found in the literature are built upon technologies such as Alloy and Prolog. Specifically, Sen et al. [151] use Prolog as a backend of the AToM^3 language workbench [93] to suggest completions of a partial model. The work was extended by using Alloy [152] to recommend the closest valid complete model within a given scope. Kermeta [105] also uses Alloy to provide completion suggestions. Refacola [157] provides a constraint-based language to express model-assistance operations in a declarative way. In the domain of education, IPSE [63] relies on Prolog to guide users in creating a class diagram. The guidelines are explicitly modelled by the teacher by means of constraints suggesting hints whenever matched. For the domain of embedded systems, DIG MDE [111] uses Prolog to guide the user in completing combinatorially challenging modelling problems on the basis of user-defined rules.

RSs for completion and repair are sometimes based on (graph transformation) rules. DPF [128] computes completion rules which ensure the satisfaction of well-formedness predicates. RapMOD [90, 91, 109] matches editing operations in UML structural diagrams to a catalogue of modelling activities, and ranks the candidate activities by relevance. Different from model completion, rule-based model repair may require deleting elements to obtain a valid model. Hence, Nassar et al. [110] derive graph transformation programs able to fix an invalid model by first deleting superfluous objects and links, and then adding necessary elements. DiaGen [101] uses hyper-graph grammar rules and hyper-graph patches (graph modifications) to propose both model completions and repairs. Similarly, ReVision [115, 116] proposes model repairs based on consistency-preserving editing rules, with heuristics that avoid undoing former editing steps.

For quality assurance, BPMoQualAssess [80] recommends improvements for process models based on rules modelling expected quality criteria (e.g., regarding size, nesting levels, and element ratios), and UCcheck [14] provides advices for improving use case diagrams based on sets of rules and guidelines.

Recommenders for model/meta-model co-evolution can also be rule-based. This is the case of ASIMOV [61], where the language designer specifies the migration assistance rules, and modellers use them to obtain recommendations for model migration. By contrast, the approach by Anguel et al. [13] suggests a migration strategy based on meta-model matching and the use of logic programming.

Some rule-based RSs target behavioural models. MDSafeCer [106] recommends how to resolve flaws of safety argumentations attached to process models. For this purpose, it first identifies the problematic elements, and then uses rules to provide advices to resolve the deviations. Also for process modelling, BAM [168] uses model-checking to detect errors in process models, and suggests corrections for the errors in relation to user-defined validation rules and Dwyer's temporal specification patterns [48].

Some works use patterns following a case-based approach. In particular, Baya [37] relies on a knowledge base of curated patterns, several similarity metrics and ranking algorithms as a basis for the recommendation of the next steps when building mashup models. Moreover, it applies weaving to incorporate the recommended pattern into the mashup model. Savary-Leblanc [146, 147] also relies on a knowledge base to produce modelling suggestions by implementing a multi-criteria rating-based RS. SimIMA [3, 159] introduces a comprehensive approach that empowers modellers with data-driven guidance by utilising a knowledge base of configurable repositories and sources. This approach incorporates clone detectors to estimate similarities between Simulink models and leverages near-miss clones to recommend similar models. The process model recommenders of Li et al. [94] and Deng et al. [39] extract task relations and patterns from process models, which are then used to recommend activity nodes for the current model. The AMOR [27] model versioning system recommends resolution patterns for conflicts between two model versions. The patterns can be mined from repositories or specified manually. DSL-maps [124] uses a catalogue of patterns to transition from the requirements of a DSL (given as a mind-map) to its design (given as a meta-model). It performs a lexical analysis of the requirements to match them against an ontology-based description of the patterns, and suggests a ranked list of patterns to realise the requirements. Mani et al. [99] also use patterns to assist when repairing faults in input models of code generators. Their approach identifies correct output fragments that are similar to the incorrect one, and suggests repair actions based on run-time data.

Finally, probabilistic forms of knowledge representation can be also applied, for example based on Bayesian networks. REBUILDER [64] combines case-based reasoning with WordNet and Bayesian networks to enable reusing UML diagrams, or part of them. Bobek et al. [24] also use Bayesian networks to recommend following tasks when instantiating a configurable process model.

Hybrid recommenders

hybrid. Some works combine several recommendation methods to benefit from their strengths and mitigate particular limitations. The surveyed papers have combined content-based techniques with collaborative filtering, social-based and knowledge-based methods.

Three approaches combine collaborative filtering with content-based recommendations. The first one, by Kögel et al. [86, 87], recommends model changes by looking at the previous model history (e.g., what other developers did on previous model versions) and co-occurring model changes.

Heinemann [69] evaluates the use of association rules and collaborative filtering to recommend Simulink library elements for the current model. The collaborative filtering method considers models as users, and elements as items. Finally, the RS of Koschmider et al. [71, 72, 89] uses both similarity metrics and frequency of use by the community to recommend complete process models or fragments.

For behavioural modelling, B-repair [31] suggests automatic repairs of faulty models written in the B formal specification language. The approach uses two types of rules (hence being knowledge-based) to suggest fixes in state machine transitions. Then, it uses machine learning (features learnt from state machine transitions, a content-based approach) to estimate the quality of the repairs and rank the recommendations.

Finally, SBPR [83, 84] combines the traditional content-based approach with social-based recommendation to suggest business process models for reuse. For this purpose, it extracts information from the user profile in LinkedIn[17]. Similarly, the approach by Rangiha et al. [129] profits from social tagging to recommend suitable actors and roles in a social business process modelling tool. In addition, it recommends tasks based on similarity metrics.

Other methods

other method. A few works use non-traditional recommendation methods based on search, static analysis or sequence based.

On the one hand, two approaches use model search as the underlying technique for recommendation, both targeting OCL. Clarisó et al. [38] generate potential fixes to OCL constraints by using mutation. Batot et al. [21] tackle the co-evolution of OCL constraints and meta-models using multi-objective optimisation guided by criteria like correctness and minimisation of changes and information loss.

17 https://www.linkedin.com/

On the other hand, several works provide recommendations out of the static analysis of models, meta-models or OCL expressions. PME [119], which extends the generic modelling environment (GME) to support proactive modelling, recommends further editing actions (e.g., connecting an object to another) upon user actions (e.g., selecting an object). The recommendations are created by the syntactic analysis of the meta-model and OCL constraints. IntellEdit [112] recommends quick fixes for repairing models based on the static analysis of failing OCL expressions. It ranks the recommended fixes by the amount of required changes (from lower to higher). AnATLyzer [142, 143, 144] extends the ATL IDE for developing model transformations with the detection of type errors and suggestions of quick fixes. Errors are detected by static analysis and model finders. Proposed quick fixes are ranked by the number of errors corrected. The ranking can be calculated dynamically using speculative analysis [107] (i.e., the simulated execution of all possible repairs and the analysis of their consequences), or statically using rankings precomputed on a set of transformations with injected faults. Lastly, SMART [67] supports test-driven development of UML models. It statically analyses the test cases and their execution logs to report errors. Moreover, it suggests quick fixes for automatically solving structural errors (e.g., adding a missing model element), and provides guidance to solve behavioural errors triggered during the test case execution (e.g., displaying a sequence diagram with the test case execution, or a summary of the changes in attribute values or the model state).

Finally, NEMO [44] utilises an Encoder-Decoder neural network to support modellers in performing model editing operations. It recommends the next item for a given sequence of items.

Any method

any method. The framework Hermes [49, 50, 51] for the creation of RSs can be extended with any recommendation strategy and recommendation method. It provides facilities to define the recommendation context, which can be obtained either implicitly or explicitly. Developers of RSs can persist user preferences (long-term temporality) and set filters and ranks for their strategies.

Once we have classified the works according to the recommendation method, we characterise how they collect the user preferences (acquisition type), the temporality of the user preferences, and the size and ordering of the recommendation sets (recommendation degree).

Data acquisition type

acquisition type. All works but Extremo collect data implicitly. The most common type of implicit data are the user's previous interactions with the system (including the current selection of elements in the editor) and the in-progress model. In some cases, like SBPR, this includes user information from LinkedIn.

In addition, 13 approaches [3, 18, 17, 19, 28, 36, 38, 71, 72, 73, 74, 89, 111, 129, 140, 148, 149, 150, 151, 152, 159] also collect data explicitly. In these cases, the data are acquired through questionnaires, parameters, tags or requirement definitions, in combination with implicit acquisition methods like analysing the user's in-progress model.

Temporality of the data

temporality. Most approaches (88.1%) collect preferences for their use during a short period of time (typically the current modelling session or model state). PARMOREL [18, 19, 74] uses long-term preferences by storing the experience gained from each repair, allowing the algorithm to improve its performance in consecutive executions. The process model recommender of Rangiha et al. [129] exploits persistent social tags to express, e.g., required skills for tasks and skill-sets of users. Hermes [49, 50, 51], being a framework, allows developers to persist user preferences as required by the recommendation strategy. Burgueño et al. [28] incorporate historical data pertaining to previously accepted or rejected suggestions. Nair et al. [15] include history-based recommendations using past modelling actions performed. Finally, only one work [140] does not provide detailed temporality information.

Recommendation degree

recommendation degree. When it comes to the recommended items, most approaches (49.2%) present *all* recommendations found by the method to the user. Since this might be overwhelming if there are many options, some approaches (3.4%) present just *one* recommendation, and others (44.1%) present the top N recommendations.

The filtering criteria vary depending on the recommendation method, and are frequently used to *rank* the suggestions. Examples of filtering criteria include the most similar models (as in [36]), the quick fixes repairing more errors (as in [144]), or the fixed model or constraint having the lowest number of modifications with respect to the original one (as in [38, 112]). Sometimes, the quality of the recommendation is estimated using pre-trained machine-learning models, as in [31]. In the case of anATLyzer [144], the user can choose either a fast but less accurate ranking of recommended quick fixes (based on a pre-computed estimation of the repair power of quick fixes); or a slower but more accurate one (based on a speculative application of the quick fixes to the current transformation). As Table 11.5 in the Appendix A shows, some early works do not provide information about cardinality or ranking.

3.1.4 *Evaluation*

In this subsection, we review how the RSs have been evaluated on the basis of the two orthogonal features shown in Figure 3.6: the followed evaluation *methods* and the used evaluation *metrics*.

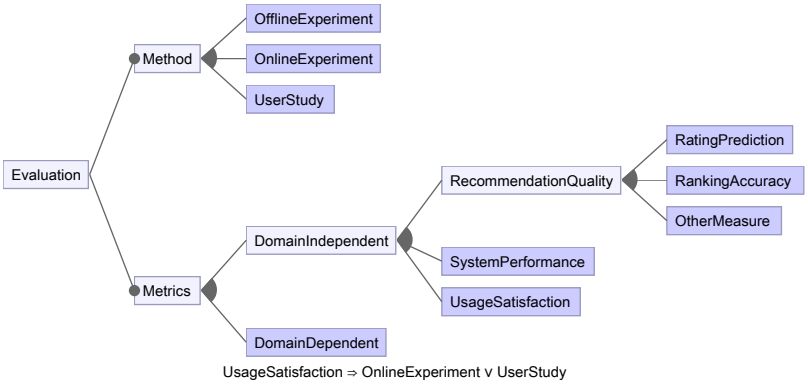


Figure 3.6: Evaluation dimensions for RSs in MDE.

As the figure shows, we first distinguish three types of evaluation methods [65]: *offline experiments*, *online experiments*, and *user studies*. Offline experiments correspond to analytical studies on datasets, online experiments are user-centric studies that evaluate the system in a real setting, and user studies consist of experiments planned for small groups of participants.

In addition, we identify several metrics to assess different recommendation goals and quality measures. We classify the metrics as *application-independent* and *application-dependent*. In turn, application-independent metrics are divided into three groups. The first one comprises traditional metrics used to evaluate RSs regardless the application or task for which they have been developed. Here, we distinguish between *recommendation quality* (accuracy) metrics – i.e., *rating prediction* metrics (e.g., MAE and RMSE) and *ranking accuracy* metrics (e.g., precision, recall, nDCG and Mean Reciprocal Rank (MRR)) – and *other measures* that capture non-accuracy recommendation characteristics, such as diversity, coverage and novelty. A second group of domain-independent metrics is related to *system performance*, such as consumed time and required resources to perform a task. The third group of domain-independent metrics is formed by *usage satisfaction* metrics, such as user engagement, perceived usefulness and trust on the system, as well as system usability, responsiveness, security and privacy. As Figure 3.6 shows, measuring usage satisfaction requires performing on-line experiments or user studies, as offline experiments do not involve users.

Finally, application-dependent metrics are devised for particular MDE applications and tasks. They include metrics such as the average number of constraint violations in model repair recommendations, or the total number of valid matches in the recommendation of model transformation mappings.

Table 11.6 in the Appendix A categorises the analysed RSs according to the method (*offline*, *online* and *user study*) and metrics used for their evaluation. An approach can appear multiple times in the

table if it was evaluated by means of several methods. Additionally, Table 11.7 in the Appendix A presents a matrix crossing the methods and metrics used in the papers. Overall, we can observe that online experiments are the least used evaluation method, and that neither rating prediction nor non-accuracy metrics are used; the former are indeed in disuse in the RS research field. A total of 17 approaches (29%) have no evaluation.

Offline experiments

offline experiment. This is the most popular evaluation method, used in 30 of the revised approaches. Making use of data records with past user behaviour and feedback, among other information, offline experiments simulate past and present real conditions without requiring the participation of users during the evaluation process.

These experiments exploit available datasets to compute a variety of aspects about a RS, such as its scalability and performance, the precision and quality of its recommendations, and the reduction of modelling effort. However, since data repositories of models are not as common as for example those for programming languages, a fundamental issue about offline experimentation for MDE recommenders is the availability of artefacts over which the evaluation can be performed. To address this issue, we have observed four solutions in the literature. The first one is the generation of synthetic data. For the case of repair recommenders, the set of artefacts is typically generated by applying mutation operators over several seed artefacts to obtain faulty artefact variants. This approach was used by anATLyzer to evaluate the recommendation of quick fixes over transformations [142, 143, 144]; by B-repair to evaluate fixes over state machines [31]; by IntellEdit to evaluate if its content-assistant solves errors in models [112]; by Matikainen et al. to evaluate the recommendation of state machines for robotic cleaners [100]; by Mani et al. to evaluate the effectiveness of its model repair recommender [99]; by Savary-Leblanc [146, 147] to replicate the work of expert architects; and by OCKHAM [160] to identify and simulate the application of edit operations. The seed artefacts may come from third parties (as in the case of anATLyzer and Mani et al.), be generated synthetically (as in IntellEdit and Matikainen et al.) or manually (as in B-repair).

A second solution is to locate repository sources of the appropriate type. For example, CONVErT was evaluated through models from the Illinois Semantic Integration Archive [17], Extremo gathered heterogeneous information from several sources such as OMG meta-models or the AtlanMod meta-model Zoo [148], Refacola used the whole AtlanMod Ecore meta-model Zoo [157], Heinemann used models of a Simulink repository [3, 69], Matikainen et al. used floor plans from the Google SketchUp database of 3D models [100], SBPR used process models from different sources [84], ReVision [115, 116] used models from the Eclipse Git repository, Weyssow et al. [167]

used meta-models from the AtlanMod meta-model Zoo and the MAR repository [70], MemoRec [41] mined meta-models from GitHub, while MORGAN [43] collected Java projects from Maven and meta-models from GitHub.

A third solution consists of taking example artefacts from published papers (as in B-repair), or datasets used by other authors (as in the case of Kögel et al. in 2016 [86, 87], and NEMO [44]).

Finally, the fourth solution is to obtain real-world artefacts from companies, like Li et al. [94] and Deng et al. [39], who used a dataset of 221 business processes collected from a local government in China, in combination with a synthetic dataset. Table 11.8 shows the public (i.e., available) datasets and repositories used in the surveyed papers.

In addition to mutating artefacts to introduce faults, we have found other modifications in artefacts. In RSs for model completion, the models of the considered dataset are removed elements to enable triggering the recommendations, and their effects are compared with the original model. This is the strategy followed by Heinemann [69], Li et al. [94], Deng et al. [39], Baya [37] and Burgueño et al. [28]. In the first case, half of the model elements were removed; in the second case, the recommendation starts from the second activity node; and in the two last cases, model portions of increasing size were systematically removed.

Some approaches require data for building a recommendation model. For this purpose, the dataset is partitioned into sets for training and test, as done by Li et al. [94], Deng et al. [39] and Heinemann [69]. To estimate the generalisability of the method and avoid problems related to overfitting and selection bias, k-fold cross-validation is recommended for statistical analysis [35]. This way, Deng et al. [39] use 5-fold cross-validation: the dataset is partitioned into five subsets, one is taken for testing, the rest for training, and the procedure is repeated 5 times with each subset. Similarly, Shilov et al. [154] used 5-fold cross-validation and Heinemann [69] used 10-fold cross-validation.

Regardless the use of datasets, some systems are empirically compared against baselines, which can be naive methods as done by Heinemann [69], who used a RS that suggests the most popular Simulink blocks in libraries. A few cases use existing recommenders built by other researchers, like Li et al. [94] and Deng et al. [39], who compare their approach against two other recommenders for process models. In other cases, the system is evaluated with and without its recommendation component enabled [119, 131]. Finally, some approaches are evaluated analytically, like Extremo [149, 150], whose extensibility is assessed via integration with several third-

party tools and formats, or PME [119], where the authors built an analytical model to estimate the modelling effort.

Online experiments

online experiment. Only two of the revised approaches were evaluated using online experiments, both in the context of external projects. ASIMOV [61] was evaluated using a real commercial scenario named Alps Furniture. Two groups of users were asked to co-evolve models either using ASIMOV or manually, and the results were analysed to assess the effort and time reduction achieved when using the tool. The domain modelling recommender DoMoRe [7, 8] was used in various industrial and research environments, and the user feedback and experience allowed identifying potential aspects for improvement.

User studies

user study. There are 15 approaches evaluated with user studies. These typically involve a small group of users who perform some tasks, making it possible to analyse the effectiveness of the users on completing the tasks with and without the recommender, as well as to gather information about user experience via questionnaires [133].

We have identified 3 types of user studies, in which: i) users perform tasks using the proposed recommender; ii) users utilise the recommender in an A/B testing setting (i.e., some users perform tasks with the recommender, and some others without it); and iii) the recommendations are compared to the decisions an expert user would make (i.e., the expert user plays the role of oracle function).

The first type of user studies was applied to AXSM [73] for evaluating usage satisfaction; to the RS proposed by Cerqueira et al. [36] for evaluating the usage satisfaction and the accuracy of its sequence diagram recommendations; to CONVErT [17] for getting user feedback on the usefulness and usability of the tool to develop transformations aided by interactive recommendations; to DSL-maps [124] for assessing the perceived usability and usefulness of its pattern assistant for building meta-models; to IPSE [63] for measuring usage satisfaction about its support on learning UML skills; to MAGNET [1] for getting user feedback on the usefulness of the recommendations to learn using AutoFOCUS3; to RapMOD [90, 91] for measuring the quality of its graphical model auto-completion recommendations and the reduction of modelling effort; and to Nair et al. [15] for evaluating the average reciprocal hit-rank and the perceived usability and usefulness.

The second study type was used by Baya [37] to evaluate (in a crowdsourced user study) whether recommending and weaving mashup model patterns reduces the development time, the number of user interactions, and the time between user interactions. In addition, the participants filled-in a questionnaire to evaluate their satisfaction

with the tool. Also in this category, Elkamel et al. [52] evaluate the relevance and accuracy of the recommended elements for UML diagrams, and Koschmider et al. [71, 72, 89] asked two sets of students to create process models with and without recommenders. In the latter case, the authors measured the time spent, the quality of the results and the usage satisfaction.

Finally, three approaches compare their recommendations with the a-priori choices of expert users. The authors of anATLyzer [144] evaluated the usefulness of its quick fixes and the utility of its ranking with respect to the free choices made by two independent developers. Paydar et al. [120, 121] used the opinion of experts as the golden standard to evaluate the accuracy of their algorithms for detecting behaviour/concepts in use cases, annotate activity diagrams with entities from class diagrams, and recommend use cases based on similarity metrics. OCKHAM [160] conducted a semi-structured interview with domain experts to evaluate the quality of the recommendations.

Metrics independent of the application

application-independent metrics. The most used ranking accuracy metrics are precision, recall and F-measure [3, 17, 28, 36, 37, 39, 43, 44, 69, 84, 86, 87, 90, 91, 112, 120, 121, 41, 160, 167]. Some papers consider additional metrics to evaluate the accuracy of the recommendations, such as MRR [3, 120, 121, 167]; 11-point interpolated average precision [120, 121]; the average number of recommended alternative solutions per successful recommendation [157]; the hit rate, which is the fraction of correct recommendations in the recommendation list [15, 39]; or relevance and accuracy rates [43, 44, 52, 41, 147, 154].

Several authors measure the performance of their approaches, being time metrics the most common, in particular, the time to compute recommendations [19, 28, 37, 44, 74, 83, 91, 101, 115, 119, 41, 148, 157], and the time spent by the user to perform a task [37, 61, 90, 109].

Finally, usage satisfaction metrics include mostly feedback from the users after using the system. The feedback is collected informally [7, 8, 73], by means of questionnaires [1, 15, 17, 37, 63, 124] or asking the users to rank the provided recommendations using a Likert scale [36, 160].

Metrics dependent of the application

application-dependent metrics. These are metrics specific for MDE activities, such as the number of model editing operations [61, 90, 99, 119], the edit distance between conflict pairs [27], the average number of properties changed per applied quick fix [157], the number of attempts to co-evolve a model [61], the number of repair alternatives [115], the lines of code needed to integrate a meta-modelling tool with the RS [149, 150], the number of meta-model constraints

fixed in a co-evolution scenario [21], the amount of constraint violations [112], the coverage of a room layout model [100], or the number of valid meta-model/role model matches [131].

Additionally, some metrics are related to the completeness or correctness of the recommendation approach, such as how complete a set of quick fixes is [99, 144], the validity of quick fixes or co-evolution actions (they completely remove an error) [61, 99, 144, 157], or the impact of quick fixes (number of problems removed or introduced by their application) [112, 144].

3.1.5 *Conclusions and opportunities*

Existing RSs for MDE target five main tasks: complete, create, find, repair and reuse. These tasks can be performed over models, meta-models, transformations or code generators. The majority of approaches focus on completion and repair (together, 76.6% of the approaches), followed by reuse (9.1%), find (6.5%), other purposes (3.9%) and create (3.9%). Also, most recommenders work over *models* (70.1%), followed by *meta-models* (22.1%), *transformations* (6.5%) and *code generators* (1.3%). In our study, we have identified numerous language-independent approaches [7, 8, 13, 19, 43, 49, 50, 51, 74, 86, 87, 101, 105, 110, 112, 115, 116, 119, 128, 148, 149, 150, 151, 152, 154, 160, 157], but most RSs are specific for a modelling language. It is worth mentioning that there is tool support for 94.92% of the approaches, though some of them (44%) are prototypes. This demonstrates the feasibility of developing RSs for MDE tasks, but more effort may be needed to increase the number of mature, fully-developed tools. Most recommenders help in the modelling activity on user demand, but proactive approaches that monitor the user activity to update the recommendations are not uncommon.

Most RSs for MDE are knowledge-based (47.5%), followed by content-based (13.6%), based on collaborative filtering (11.9%) and hybrid (10.2%). Additionally, other methods accounted for (15.3%) and any method (1.7%). Additionally, most of the information to build personalised recommendations is collected implicitly. Only 13 works consider explicit preferences of users, and all but one of those cases use implicit information as well. Finally, 42 out of the 59 approaches (71%) were evaluated through offline experiments, the most frequent kind of evaluation. Some of the revised RSs were evaluated using domain-independent metrics applicable to general RSs, specifically ranking accuracy metrics (mainly precision, recall and F-measure), time metrics and usage satisfaction collected via questionnaires.

Our analysis of the state-of-the-art reveals some gaps in the targeted tasks and artefacts. Most approaches focus on models, a handful on meta-models, very few on transformations, and hardly any on code generators. Similarly, the purpose of most RSs is complet-

ing and repairing models. However, there are few recommendation approaches for finding relevant artefacts, reusing them in a given context, and creating artefacts from scratch. For the latter case, we envision RSs proposing initial artefact templates out of higher-level descriptions, maybe defined using natural language. Many of the studied papers present RSs for a specific language or tool. Such recommenders tackle a single problem and are "hard-wired" into the systems they were designed for. Hence, an open line of research is devising solutions that allow adapting the recommendation algorithms, the users' preferences or the evaluation metrics to the users' needs. Additionally, one of the main barriers when building RSs for MDE activities is the lack of data that can be used for training the recommenders.

When it comes to recommendation methods, a large percentage of the RSs in MDE are knowledge-based and content-based. Building upon this trend, there are plenty of opportunities for further research on leveraging collaborative filtering in MDE, for instance, via model and code sharing platform. Moreover, the few revised works that apply collaborative filtering to MDE neglect long-term users' preferences, highlighting an important problem worth investigating.

3.2 AUTOMATED GENERATION OF RECOMMENDER SYSTEMS

Most RSs for modelling were developed ad-hoc, by hand and from scratch. Since this requires high expertise and effort [108], we identify the need for methods and tools that automate their construction [10]. Following this idea, some recent works propose solutions to reduce the amount of time, effort and expertise required to develop and integrate RSs.

LEV4REC [47] is a low-code tool that allows the configuration of RSs for non-modelling environments by means of a feature model. Some aspects that can be customised include the recommendation algorithm or the underlying libraries. From this information, it generates the source code of the RS ready for deployment. However, being a generic tool, LEV4REC does not support the customisation of RSs for modelling languages, or the deployment and integration of the RSs with modelling tools. Likewise, it does not support the exploration of data preprocessing policies or the analysis of the RS performance.

Fellmann et al. [58] propose a reference model focused on data requirements of RSs for process modelling. The model can be used as a guide for developing new process modelling recommenders or evaluating existing ones. Hence, it is specific to process modelling and does not provide automation or code synthesis.

Hermes [50] allows building Eclipse-based RSs that help in completing models with recommended elements from other models in

a repository. It can be applied to models and meta-models within the EMF ecosystem. It has a plugin-based architecture with extension points for defining new recommendation methods, data repositories and the integration with modelling editors. However, the recommenders are specific for Eclipse modelling tools, and need to be coded. In contrast, as Chapters 4 and 5 will show, the approach herein proposed allows generating recommenders automatically from a high-level description, using a DSL, and without coding. Moreover, such recommenders can be integrated with any modelling tool via a REST API, and be evaluated to assess their performance.

Rojas et al. [137] propose an MDE framework to create mobile RSs of geographic points of interest. It allows the definition of the structure, behaviour and navigation of the RSs, as well as the customisation of the user preferences and similarity criteria for points of interest. Later on, they proposed a similar solution for trips and tours recommendation [136]. In both cases, the domain of the recommendation is fixed.

Espinosa et al. [53, 54] present an MDE solution to assist non-expert users in applying data mining. They propose a framework that reuses past experiences of data mining experts to calculate how accurate a new dataset is and recommend the one with the best performance. The framework supports the customisation of the data mining task to perform, the mining algorithm, the evaluation method and the metrics. Although it allows customisation flexibility, recommendations are specific to data mining applications.

UbiCARS [103] is a framework for automating the development of RSs in e-commerce using a graphical DSL. The framework aims to expedite development time, improve recommendation accuracy, and leverage efficient context-aware recommendation algorithms. By creating and manipulating various models, including domain, user preferences, and item characteristics, the framework facilitates the design process. With the utilisation of MDE, practitioners can construct UbiCARS by designing the model using the provided DSL editor. Subsequently, the framework automatically configures and deploys the UbiCARS app and CARS system. While offering customisation flexibility, the recommendations generated by the system are tailored specifically for e-commerce applications.

Kaindl et al. [81] propose an approach to semi-automatically generate dialogue-driven recommendation processes and their graphical user interfaces (GUIs) using knowledge discovery through text mining techniques. The recommended process consists of two stages: data preprocessing and recommendation generation. In the data preprocessing stage, raw data is collected and transformed into a suitable format for recommendation generation. The recommendation generation stage focuses on creating recommendations based on the preprocessed data, incorporating user preferences and feedback into the

process. As before, the system recommendations are customisable, but specific for retail applications.

Also with the aim to speed up the development of RSs, the RSs community has coined the term *Recommendations-As-a-Service* (RaaS) [18] to refer to cloud platforms that enable the creation of RSs using a few clicks or Lines of Code (LoC), by automating the steps of the recommendation generation process, from data indexing to recommendation generation and display. As an example, the engine *Recombee* [19] allows building recommendation services for any domain that has a catalogue of items and is interacted by users. The engine only supports content-based recommendation, but its recommendation model is customisable and allows defining business rules to filter or boost items based on their properties. The engine provides API endpoints to manage the (JSON-based) items, users and their interactions, and to get recommendations. While *Recombee* is generic, some RaaS are domain-specific, like *bX* [20], *BibTip* [21] and *Mr. DLib* [22] for digital libraries. All these approaches expose recommendation APIs as web services; however, the APIs are not modelling-specific, so their fine-tuning for modelling tasks is cumbersome.

Overall, after conducting this analysis, it becomes evident that there is a lack of tools that automate all the steps in the creation and evaluation of RSs for modelling languages, as well as for their automated integration in modelling environments. Additionally, to our knowledge, there is currently no proposal enabling the aggregation of recommendations for modelling.

3.3 SUMMARY

In our state-of-the-art analysis, we categorise the RSs for MDE based on four dimensions: domain, tooling, recommendation approach, and evaluation. Concerning the application domain, key features include the artifact type (meta-model, model, transformation, and code generator), language independence, and purpose (complete, create, find, repair, reuse, or other). The majority of existing RSs in MDE focus on completion and repair (76.6%), primarily for models (70.1%). Most RSs are language-dependent, but language-independent approaches exist.

The tooling analysis delves into maturity, tool independence, recommender trigger, and recommendation enactment. Maturity levels range from proposals to prototypes and mature tools, with most implementations being plugins or complete systems/extensions. Four

18 https://recommender-systems.com/resources/recommendations-as-a-service-raas/
19 https://docs.recombee.com/
20 https://www.exlibrisgroup.com/products/bx-recommender/
21 http://www.bibtip.com/en
22 http://mr-dlib.org/

approaches are genuinely tool-independent, while others may be used with multiple tools but remain tool-dependent. Recommender triggers are categorised as on-demand, proactive, or both, with a few supporting both modes. Recommendation enactments vary from manual and interactive to automated or semi-automated. Tool support is prevalent (94.92%), with 44% being prototypes.

MDE researchers have applied diverse recommendation approaches for various tasks and purposes. Findings include four primary recommendation methods: content-based, collaborative filtering, knowledge-based, and hybrid. User preferences are acquired implicitly or explicitly, with temporality categorised as short-term or long-term. Recommendations encompass various item types (e.g., artefacts, modelling advice, editing actions), and the recommendation degree is determined by cardinality and ranking. Noteworthy approaches include content-based methods utilising similarity metrics, collaborative filtering leveraging user community data, knowledge-based techniques employing constraint and case-based methodologies, and hybrid methods combining different strategies. Other unconventional methods involve search, static analysis, sequence-based, and neural networks. Knowledge-based RSs dominate (47.5%), and implicit data collection is common for personalised recommendations.

When it comes to evaluating RSs in MDE, two key aspects are assessed: methods and metrics. The three primary evaluation methods include offline experiments, online experiments, and user studies. Metrics fall into application-independent (ranking accuracy, performance, and usage satisfaction) and application-dependent categories. Common metrics involve precision, recall, F-measure, MRR, time to compute recommendations, and user feedback. Application-dependent metrics focus on specific MDE activities, measuring aspects like model editing operations, constraint violations, and coverage. Some other metrics also evaluate the completeness and correctness of recommendation approaches. Evaluation is mainly through offline experiments (71%), emphasising ranking accuracy metrics, time metrics, and usage satisfaction questionnaires.

Part II

PROPOSED APPROACH

4

AUTOMATED ENGINEERING OF RECOMMENDER SYSTEMS FOR MODELLING LANGUAGES

This chapter describes our proposal to facilitate the construction of RSs for particular modelling languages. To provide a comprehensive understanding of our proposal, in Section 4.1, we begin with a running example, and in Section 4.2, we present an overview of the developed approach.

Next, in Section 4.3, we introduce Droid, our meta-model representing a textual DSL that facilitates the definition of each step in a RS configuration process. Subsequently, we examine each of these steps. In Subsection 4.3.1, we discuss the configuration of recommendation targets (users) and items. In Subsection 4.3.2, we explore the included data preprocessing techniques. In Subsection 4.3.3, we present the data splitting protocol and the recommendation methods. Lastly, in Subsection 4.3.4, we focus on the support for the evaluation of generated RSs, discussing the different evaluation methods and metrics employed.

Additionally, in Section 4.4, we present details of the deployment of RSs defined with Droid. This deployment relies on a generic recommendation service API called DroidREST. Next, in Section 4.5, we address the integration and reuse of RSs for modelling languages. This encompass an overview of the integration and reuse approach (Subsection 4.5.1), the recommendation indexer API (Subsection 4.5.2), the adaptation of RSs to the modelling notation (Subsection 4.5.3), and the aggregation of recommendations (Subsection 4.5.4). Finally, in Section 4.6, we conclude with a summary of the chapter.

4.1 RUNNING EXAMPLE

To explain our approach we will use it to build RS for assisting in the creation of UML class models by recommending attributes and operations (methods) for target classes. Models are defined using a modelling language. In MDE [25], it is a standard practice to describe the abstract syntax of modelling languages by means of a meta-model.

Figure 4.1 shows an excerpt of the meta-model of UML 2.0 class diagrams, which we use in a running example. It structures UML class models into Packages that can contain Classes, Datatypes and Associations. Classes can declare attributes (class Property) and Operations, as well as inherit from other classes (relation Class.superClass). Figure 4.2 shows an UML model (i.e., an instance of the UML meta-model) in abstract

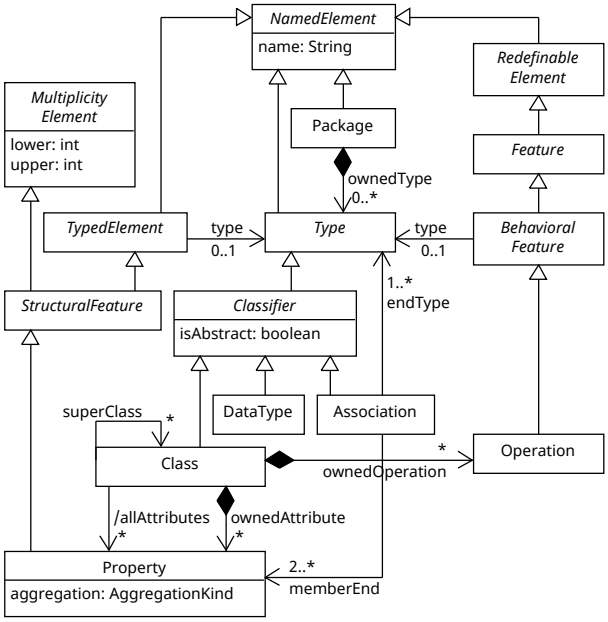


Figure 4.1: UML meta-model excerpt.

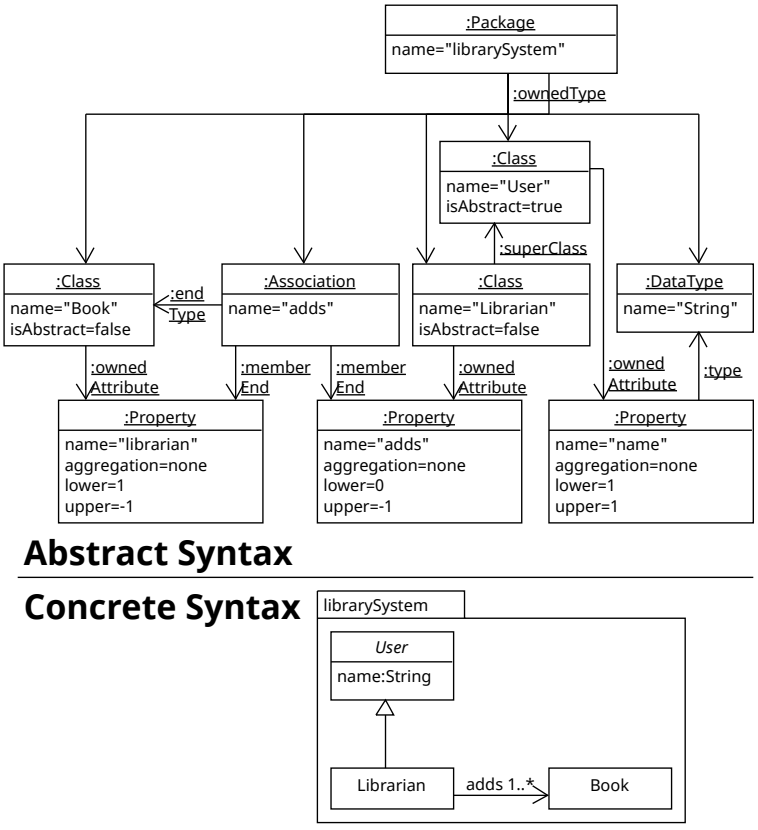


Figure 4.2: UML model in abstract and concrete syntax.

syntax on top, and using the standard concrete syntax of class diagrams at the bottom.

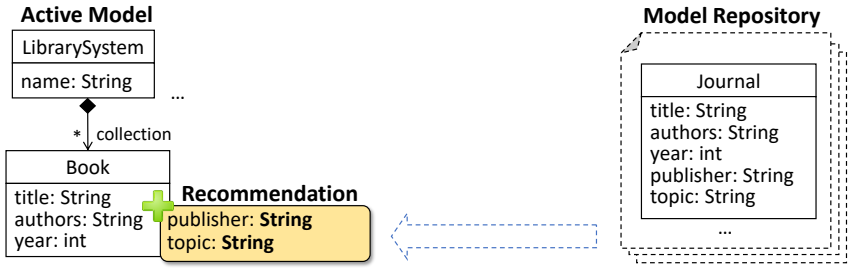


Figure 4.3: Obtaining recommendations to assist in the creation of a UML class model.

Figure 4.3 illustrates the behaviour of the RS that we aim to build. The left part of the figure shows a class model being created, which contains the classes LibrarySystem and Book. To help the designer to complete the target class Book, the RS would analyse similar classes previously defined in other models, and would return a ranked list of suitable new attributes and operations for the class. As an illustrative example, the figure shows a repository with previous models, among which one declares a class Journal.

As the classes Journal and Book have similarities, given that both represent information resources commonly found in libraries or digital repositories, and share essential attributes like title, authors and year, the RS propose adding the attributes specific to Journal, namely, publisher and topic—into the Book class, which are not currently present in the class.

Overall, the RS to build should recommend two types of items (attributes and operations) for a given target class. However, building such a RS by hand is costly. While there are well-known recommendation methods for building RSs (cf. Section 2.1), they need to be adapted for the modelling language (class models in our example) and task (recommendation of attributes and operations). Moreover, the performance of these methods may differ. For instance, one method may select classes with similar items to the ones in the target class, and recommend items from those classes; another method may recommend items that are similar to those the target class already has; while yet another method may recommend the most popular items (i.e., those appearing most frequently in classes). Selecting the best performing recommendation method for a modelling language needs to be an informed decision based on a set of metrics carefully chosen. However, computing those metrics by hand is costly. Finally, while some RSs for modelling have been proposed [10, 28, 41, 43], they are typically hardwired to a specific modelling language and cannot be used for other languages, even if they are similar.

To overcome this limitations, in the remainder of this chapter, we propose Droid, a model-driven solution to facilitate the creation, eval-

uation and deployment of RSs for modelling languages. A distinguishing feature of Droid is that it does not require deep knowledge of RSs or programming.

4.2 OVERVIEW OF THE APPROACH

Our approach aims to facilitate the adoption of RSs in MDE. With this goal in mind, we propose a model-driven solution to automate the synthesis of RSs for particular modelling languages. Our solution consists of a DSL called Droid. It supports the configuration of the kind of model elements to be recommended, the preprocessing techniques to be applied to input data, and automates the evaluation of different recommendation methods against configurable metrics. Additionally, the approach deploys selected RS in a REST API called DroidREST, facilitating its integration with different modelling clients.

Before delving into the specifics of the Droid DSL, we first detail the steps of the process to define and generate RSs with Droid. Figure 4.4 presents all the necessary steps.

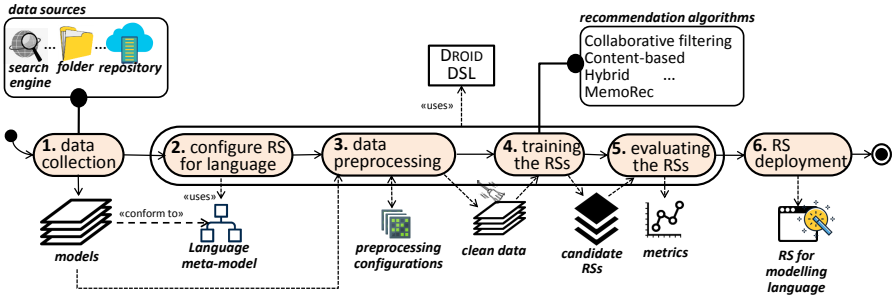


Figure 4.4: Process to create a RS with Droid.

Recommender systems creation process

The first step in the RS creation process is the data collection step. In this phase, the RS developer needs to gather all the necessary data for training and evaluating candidate RSs. Here, the term "candidate RSs" refers to the potential options that the RS developer considers for deployment later on. These candidates may vary from one another in terms of preprocessing techniques, recommendation methods, and/or the parameters used for the recommendation methods. The input data consists of models conformant to the meta-model of the modelling language the RSs are built for. For convenience, our proposed tool provides a user-friendly wizard that facilitates data collection from various sources. Moreover, additional data sources can be incorporated through an extension point.

Steps 2 to 5 are guided by the DSL, playing a crucial role in the process. In step 2, the RS developer leverages the DSL to define the items to be recommended (e.g., attributes, operations) and the target of the recommendation (e.g., classes). This process effectively configures the RS to align with the particular modelling language, ensuring that the

chosen items and target conform to the language meta-model. For more comprehensive insights, please refer to Subsection 4.3.1, where this information is explored in detail.

In step 3, the RS developer has the option to utilise the DSL for preprocessing the data gathered in step 1. This preprocessing step allows for the modification or elimination of irrelevant or malformed information. To facilitate this process, the DSL offers valuable preprocessing primitives, such as the removal of special characters like blank spaces or non-alphabetic characters. The RS developer can define multiple candidate preprocessing configurations, and can evaluate the effects of each one to determine the most suitable one. For a more comprehensive understanding, Subsection 4.3.2 provides a in-depth exploration of the preprocessing techniques available.

Step 4 involves the training of the candidate RSs. During this phase, the RS developer selects the data splitting techniques to apply to the data, as well as the candidate recommendation methods for training, having the possibility to experiment with various parameter values to optimise the training process. Our current implementation supports seven recommendation methods, and it allows for the integration of new methods through an extension point. Each candidate method is applied to the preprocessed data, resulting in the generation of different candidate RSs. For a comprehensive exploration of this step, please refer to Subsection 4.3.3.

In step 5, Droid automatically evaluates the resulting RSs against a set of performance metrics. The DSL allows selecting the set of metrics to consider, as well as the evaluation options. The result of the evaluation is a report with the value of the metrics achieved by each candidate recommendation method. This way, the RS developer can easily identify the best performing method. Subsection 4.3.4 includes additional details about this step.

In step 6, the RS developer selects the most suitable RS, which is then automatically deployed as a REST service. This deployment enables the easy reuse and integration of the RSs with different modelling clients. For a more comprehensive understanding, Section 4.4 provides details about the recommendation service.

4.3 THE DOMAIN-SPECIFIC LANGUAGE DROID

At the core of our approach, we have designed the DSL Droid. It allows the configuration of RSs for arbitrary languages that are defined by a meta-model. The DSL facilitates the configuration of the kind of elements to be recommended, the data preprocessing techniques, the recommendation method, the data splitting techniques, and the evaluation protocol. The DSL provides a high-level textual syntax for this task, which avoids the RS developer the use of programming lan-

guages like C or Java (typically more technical and complex) or the need to have deep expertise in libraries for RSs.

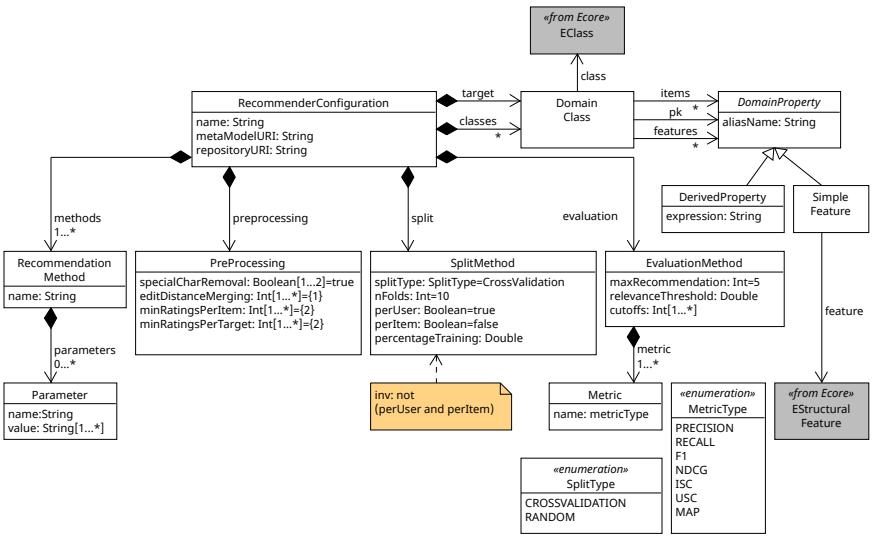


Figure 4.5: Meta-model of the Droid DSL.

DSL Droid meta-model

Figure 4.5 shows the meta-model of the DSL Droid, which serves as the foundational structure for defining and representing RSs for particular modelling languages. It enables the generation of RSs for both models and meta-models, providing a comprehensive framework for configuring RSs. RecommenderConfiguration is the root class that contains the other meta-classes, and specifies the name of the recommender system, the meta-model of the notation for which the candidate recommendation methods are being defined, and the set of instance models conformant to this meta-model. The instance models will be used to train and evaluate the candidate recommenders.

DomainClass allows specifying the type of the model elements that will play the role of target (user) in the context of a RS. In turn, DomainProperty is used to specify the type of the items to be recommended, which can be either simple features (attributes or references) of the specified DomainClass or derived features defined via expressions.

The RecommendationMethod class allows selecting the recommendation methods of interest (e.g., item popularity, content-based, collaborative filtering) and configuring their parameters (e.g., the neighbourhood size for collaborative filtering methods).

The PreProcessing class facilitates the configuration of different preprocessing techniques to ensure the data are in optimal state (e.g., the removal of special characters, the merging of words based on the Levenshtein distance).

The SplitMethod class allows customising how to split the set of provided instance models for training and testing the candidate recommenders. In particular, it defines the split type (random or cross-validation), the number of folds (if needed), the splitting method (per-

user or per-item), and the percentage of data used for training (the rest of the data will be automatically assigned for testing).

The EvaluationMethod class defines all the configurations related to the evaluation of the candidate recommenders, namely, the metrics used for the evaluation (e.g., precision, recall, F1), the maximum number of recommended items, and the relevance score threshold of a recommendation.

In the following subsections, we will delve into each of these configuration aspects in detail, exploring how they contribute to the generation of the candidate recommenders, and how they can be configured using the DSL textual syntax. Through a comprehensive understanding of these components, RS developers can configure and fine-tune the RS to meet their specific requirements and achieve optimal recommendation performance.

4.3.1 *Configuring the recommender systems to the modelling language*

To illustrate the configuration process, we will show how to configure a RS for UML class models. The goal is to develop a RS tailored to facilitate the creation of UML class models by offering recommendations of new class attributes and operations. Throughout this and the following subsections, we will use a different listing for each configuration step. Although these listings are from a single file, we present them individually by step for the sake of clarity.

Listing 4.1 configures the RS for the targeted modelling language, which is UML in our running example. First, lines 1–3 allow assigning a value to the attributes of class RecommenderConfiguration in Figure 4.5. In line 1, the keyword Recommender allows specifying the name that the RS will have. In line 2, the keyword Metamodel is used to indicate the Uniform Resource Identifier (URI) of the meta-model. This is the location of the meta-model that defines the structure and concepts of the modelling language under consideration (cf. Figure 4.1). In line 3, the keyword Repository allows indicating the URI of the repository that contains the instances of the meta-model that will serve as the input for training and testing the candidate recommendation methods.

Configuration of targets and items

Next, lines 5–10 specify the target of the recommendations – i.e., classes– and the modelling items that will be recommended – i.e., attributes and operations. Class (line 6) is a meta-class of the UML meta-model (cf. Figure 4.1). For each item to be recommended, the listing provides two pieces of information: an alias for the item ("attributes" and "methods") and the actual property used to access the item from the meta-class. In this case, ownedAttribute and ownedOperation (lines 7–8) are two associations stemming from Class (cf. Figure 4.1). The use of an alias allows modelling tools that integrate the RS to display a different name for the items than that of the properties in the meta-model. For instance, the listing specifies that "attributes" should be

```
1  Recommender: "LiteratureUMLRecommender"
2  Metamodel: "http://www.eclipse.org/uml2/5.0.0/UML"
3  Repository: "/UMLModels/instances"
4
5  Target {
6    class Class {
7      item "attributes" : ownedAttribute;
8      item "methods" : ownedOperation;
9    }
10 }
11
12 Identifiers {
13   class Class {
14     pk feature name;
15   }
16   class Property {
17     pk feature name;
18   }
19   class Operation {
20     pk feature name;
21   }
22 }
```

Listing 4.1: Configuring the recommendation target and the recommended item types.

displayed instead of ownedAttribute, and "methods" instead of ownedOperation.

Finally, lines 12–22 define the attributes to be used as identifiers of the target class and the items (their name in all cases). In addition to Class (which is the recommendation target), these lines declare the identifiers of meta-classes Property and Operation, as they are the target of associations ownedAttribute and ownedOperation in the UML meta-model. The identifiers are established using the class SimpleFeature, as depicted in Figure 4.5.

Furthermore, the Droid meta-model allows for the specification of derived properties using OCL expressions (illustrated by the class DerivedProperty in Figure 4.5). For the example of listing properties, a conceivable derived property might involve incorporating the type of the item into the identifiers of the meta-classes Property and Operation. In such cases, OCL expressions could be employed to access this information through the feature ownedAttribute.type, though our current implementation does not support it yet.

This configuration mechanism enables the specification of the types of objects receiving the recommendation (targets), the types of objects to be recommended (items), and their features. This is enough to configure RSs where the items are objects with attributes, connected to the target of recommendations. In our example, this permits the definition of a wide set of RSs, e.g., recommending classes for packages, or parameters for operations. However, this mechanism is not able to recommend patterns of objects, e.g., UML design patterns.

4.3.2 *Data preprocessing*

Data preprocessing aims at improving the quality of the input data, which, in our case, refers to models. The preprocessing step involves modifying or deleting irrelevant information from the original dataset to enhance the subsequent RS building [133]. The goal is to ensure that the data is in an optimal state for generating accurate and effective recommendations.

To achieve this, the DSL DROID offers four essential preprocessing options represented by the PreProcessing class (cf. Figure 4.5), which can be combined. Each option targets specific aspects of the dataset to enhance its quality.

The first preprocessing option is *specialCharRemoval*. It allows the removal of special characters such as numbers, blank spaces, and non-alphabetic characters (e.g., exclamation marks, commas, underscores and symbols) from the textual information within the models of the dataset. By eliminating these characters, the resulting data become more consistent and better suited for subsequent processing. This option can take on two values: *true* and *false*, indicating whether the special character removal should be applied or not.

The second option is *editDistanceMerging*. It involves specifying the Levenshtein distance, a metric that determines the similarity between two strings. The Levenshtein distance represents the minimum number of deletions, insertions or substitutions required to transform one string into another [153]. For instance, the strings "car" and "cat" have a Levenshtein distance of 1, as it only takes one substitution to transform "car" into "cat". By setting a threshold value for the Levenshtein distance, strings with a similarity below the threshold will be considered equal for preprocessing purposes.

The third option is *minRatingsPerItem*. It is used to remove items from the dataset that do not appear in a minimum number of targets. In other words, if an item is not defined by a sufficient number of targets, it is considered less relevant, and is removed from the dataset. This allows focusing on items that have a significant presence and are more likely to contribute to the accuracy of the recommendations.

Similarly, the fourth option *minRatingsPerTarget* is used to remove targets from the dataset that lack a minimum number of associated items. By eliminating targets with a limited number of items, the dataset becomes more refined, emphasising targets that have sufficient related items for facilitating meaningful recommendations.

Configuration of preprocessing technique

Listing 4.2 presents an example for the configuration of the data preprocessing using the DSL DROID. It shows how multiple values can be assigned to each option, making the engine to perform all possible combinations. In this example, the configuration includes two settings for *specialCharRemoval* (both true and false; in line 24), three settings for *editDistanceMerging* (values 2, 3, and 4; in line 25), three

settings for *minRatingsPerItem* (values 1, 2, and 3; in line 26), and three settings for *minRatingsPerTarget* (values 1, 2, and 3; in line 27). These settings result in a total of 2*3*3*3=54 different preprocessing combinations to be evaluated by the DROID engine. Through evaluating these combinations and calculating various metrics, the RS developers can choose the most appropriate preprocessing approach for their specific requirements.

```
23 PreProcessing {
24   specialCharRemoval: true, false;
25   editDistanceMerging: 2,3,4;
26   minRatingsPerItem: 1,2,3;
27   minRatingsPerTarget: 1,2,3;
28 }
```

Listing 4.2: Configuring the data preprocessing options.

The supported preprocessing options were selected based on an analysis of the most common forms of preprocessing that modelling RSs use for models (e.g., merging similar items [41]) and general-purpose RSs use for data (e.g., data cleaning, filtering scarcely rated items or targets [12]). Since the options can be combined and receive ranges of values, the RS designer can quickly specify large amounts of preprocessing configurations.

4.3.3 *Training the recommender systems*

Training the candidate recommenders involves two tasks: splitting the available data and exploiting part of such data for building certain recommendation model. More specifically, data splitting entails dividing the input dataset into training and test sets, and building the recommendation model is performed on the the training set. These tasks play a fundamental role in the development and offline evaluation of RSs, as they enable the models to learn from known (training) data, and to be evaluated with (test) data unseen by the models. The DSL DROID makes these tasks easier by providing means to configure and automate the data splitting and RS building processes.

The SplitMethod class in Figure 4.5 captures the data splitting options of the DSL DROID. It supports two techniques for data splitting: *CrossValidation* and *Random*. The *CrossValidation* technique is particularly useful for ensuring good generalisation and reducing overfitting, a common challenge in RSs [132]. It involves specifying the number of subsets, known as *k folds*, to be created. One subset is used for testing, while the remaining subsets are utilised for training. This process is repeated *k* times, with each subset taking turns as the test set. On the other hand, the *Random* splitting technique allows for random sampling of the data based on a specified percentage for

training and testing. The sampling is conducted following a uniform distribution.

In addition to the splitting technique, the splits can be performed either *per user* or *per item*. When performed *per user*, the subsets are created for each individual user, and when performed *per item*, the subsets are created for each available item. For instance, a *perUser* random split with an 80% of training data would utilise 80% of the preferences associated with each user (i.e., 80% of the attributes and operations of each class) for training, and the remaining 20% for testing.

Configuration of data splitting protocol

Listing 4.3 shows the data splitting configuration for the running example, which is a 10-fold cross-validation with a per-user distribution. This configuration specifies that the dataset will be split into ten subsets on a per-user basis. These subsets will be used for both training and testing the RS, ensuring a comprehensive evaluation of the performance of the candidate recommenders.

```
29 Split {
30   splitType: CrossValidation;
31   nFolds: 10;
32   perUser: true;
33 }
```

Listing 4.3: Configuring the dataset splitting into training and test sets.

Once the data splitting is configured, the next step is to train the candidate recommenders using a selection of recommendation methods and their respective parameter values (classes RecommendationMethod and Parameter in Figure 4.5). DROID supports seven recommendation methods. Among them, the following two methods do not have parameters: *Item Popularity* (ItemPop) and *Content-Based Cosine Similarity* (CosineCB). The remaining five methods require specifying the neighbourhood size as a parameter: *User-Based Collaborative Filtering* (UBCF), *Item-Based Collaborative Filtering* (IBCF), *Context-Aware Collaborative Filtering* (CACF), *User-Based Collaborative Filtering with Content-Based Similarity* (CBUB), and *Item-Based Collaborative Filtering with Content-Based Similarity* (CBIB).

The DSL DROID allows the integration of new recommendation methods and their respective parameters through a dedicated extension point (cf. Section 5.1.1). For generality, the meta-model of DROID, as well as the extension point, support methods with any number of parameters. The current implementation, however, is limited to methods with zero or one parameter (the latter being the neighbourhood size of the five methods mentioned above). The main benefit of this extension point is the flexibility to integrate new recommendation methods without the need to modify the DSL.

To train the candidate recommenders, the RS developer selects the desired recommendation methods and specifies values for their pa-

rameters. The DROID engine then trains the multiple candidate recommenders, each with a specific combination of recommendation methods and parameter values.

Configuration of recommendation methods

Listing 4.4 configures the recommendation methods for the example, where we have selected all the seven supported methods. The parameter values appear after the method name between parenthesis, and specify different user/item neighbourhood sizes (i.e., number of most similar users/items with which generating recommendations). For instance, in line 36, the method CACF is configured with two possible neighbourhood sizes (5 and 10), which accounts to the creation of two different candidate recommenders, one for each neighbourhood size. Overall, the listing specifies a total of 12 distinct candidate recommenders based on these parameter combinations.

```
34 Methods {
35   ItemPop, CosineCB, CBIB("5","10"), CBUB("5","10"),
36   CACF("5","10"), IBCF("5","10"), UBCF("5","10");
37 }
```

Listing 4.4: Selecting the candidate recommendation methods to train the RS.

Overall, extensions points make the DSL extensible by enabling the addition of new recommendation methods in an external way. Instead of hard-coding these methods, e.g., as an enumerate or class in the meta-model of Figure 4.5, the class RecommendationMethod specifies the name of an available method in its attribute name, and sets values to the method parameters. The DSL editor has information of the available methods (these are the externally implemented extension points), so it is able to type-check that the referenced method exists and the parameter values are consistent with the parameter types defined in the extension point (cf. Subsection 5.1.1).

4.3.4 *Evaluating the recommender systems*

The final step concerns the configuration of the evaluation of the created candidate recommenders. For this purpose, the DSL DROID allows choosing the evaluation metrics and other important values for assessing the performance of the selected recommendation methods. This is achieved through the EvaluationMethod class in the meta-model of Figure 4.5.

Configuration of evaluation protocol

Listing 4.5 shows the configuration of the evaluation in the running example. Line 39 declares the metrics to be computed. It includes all currently supported metrics in DROID: *Precision*, *Recall*, *F1* (the harmonic mean of *Precision* and *Recall*), *nDCG*, *ISC*, *USC*, and *MAP*. These metrics provide comprehensive insights into different aspects of the performance of the RSs.

```
38 Evaluation {
39   metrics: Precision, Recall, F1, NDCG, ISC, USC, MAP;
40   cutoffs: 1,2,3,4,5;
41   maxRecommendations: 5;
42   relevanceThreshold: 0.5;
43 }
```

Listing 4.5: Choosing and configuring the evaluation metrics.

Line 40 specifies the number of top items in the ranked recommendation lists used to compute the metrics. These values, known as *cutoffs*, determine the length of the recommendation lists considered in the evaluation. By selecting appropriate cut-off values, the RS developer can evaluate the performance of the RSs for different sizes of the recommendation lists.

Furthermore, line 41 sets the maximum number of items that the RS will recommend. This restricts the total number of recommendations generated by the RS during the evaluation process. This way, users can define a practical number of recommendations that aligns with the requirements of their application and constraints. In addition, when the RS is deployed, it will also use this value as the default number of recommendations to be populated. However, as Section 4.4 will show, the deployed recommendation service permits customising and modifying this number as needed.

Finally, line 42 states the minimum value of the estimated preference score above which an item is considered a good recommendation. This value, referred to as the *relevanceThreshold*, serves as a filter to exclude items with low estimated preference scores. By setting an appropriate relevance threshold, the RS developer can ensure that only items with a certain level of preference are considered as potential recommendations. This filtering mechanism enhances the quality and relevance of the generated recommendations.

After the RS developer have configured all the previous steps, Droid automatically creates the candidate recommenders, and computes the selected metrics over them. Subsequently, the RS developer selects the most suitable RS among the candidate ones, which can be automatically deployed as explained in the next section.

The metrics are computed on the input data reserved for testing by the split technique (Subsection 4.3.3). When considering a *perUser* split, for each target user (i.e., classes in our example), a metric is computed using the user's items (i.e., attributes and operations in our example) in the test set and the items recommended by a target method, measuring to what extent the two sets of items are close. For example, *precision* measures the percentage of recommended items that belong to the user's test set, and *recall* measures the percentage of the user's test items that the method recommends. This can be measured for the entire recommendation lists, or for a *cutoff* of the

Table 4.1: Endpoints of the recommendation service API.

Endpoint	/features
Description	Returns the features of all RSs within the recommendation service.
Method	GET
Output	Features of the recommendation service (JSON).
Endpoint	/recommend/⟨*name*⟩?(newMaxRec=⟨*maxRec*⟩)?, (threshold=⟨*threshold*⟩)?, (itemType=⟨*type*⟩)?
Description	Returns a list of recommendations from the RS *name*, for a given target (within a context, if required). All parameters are optional: newMaxRec (integer) is the maximum number of recommendations to retrieve, threshold (double) is the threshold for the ranking, and itemType ([string]) is the type of the recommended items.
Method	POST
Body	Target and its context, if required (JSON).
Output	List of recommendations (JSON).

top k items of each list. Moreover, in this process, the relevance of a recommended item can be established by a *threshold* score value.

Overall, the supported metrics are some of the most common ones for evaluating RSs [65, 169].

4.4 DEPLOYMENT OF RECOMMENDER SYSTEMS

DroidREST

After carefully evaluating the candidate RSs, the RS developer can hand-pick her preferred RS. The chosen RS is then seamlessly deployed onto a recommendation service API, named *DroidREST*. Client modelling tools can initiate POST requests to this service, providing parameters such as the name of the recommender and the recommendation target (e.g., a class in our running example). The outcome of this request is an ordered list of recommended items for the specified target, which in our case would include attributes and operations.

It is essential to note that *DroidREST* operates as a generic recommendation service, where the deployment of a new RS only involves uploading a series of specific configuration files. This means that a new service is not created for every deployed RS, but rather, *DroidREST* is able to interpret the uploaded configuration files of the different RSs to return convenient recommendations. The configuration files, uploaded to *DroidREST* upon the deployment of a RS, encompass crucial details of the RS and determine its characteristics.

Table 4.1 shows the primary endpoints of *DroidREST*. The /features endpoint allows clients to retrieve the metadata of all RSs within *DroidREST*. Figure 4.6 shows a conceptual model of the metadata.

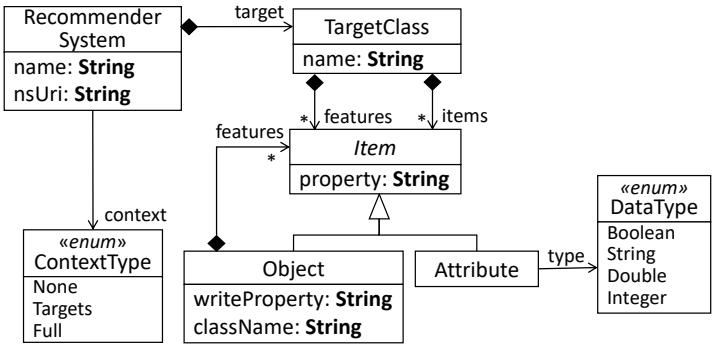


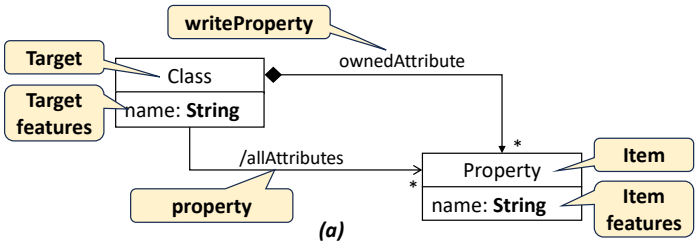
Figure 4.6: Conceptual model of RSs assumed by our approach.

Class RecommenderSystem defines the name of the RS, the meta-model *nsURI*, and a description of the modelling *context* that the RS needs to compute the recommendations. The context can be None (only the target of the recommendation is needed), Full (requires the whole model that contains the target element), or Targets (requires all objects with the same type as the target element). For the moment, *Droid* produces RSs where the type of context is Targets or None.

The metadata also defines the target class of the recommendations and its identifying features. For simplicity, our approach assumes that a RS only serves recommendations (e.g., attributes and operations) for a target type (e.g., UML classes). If several target types are supported, then one RS for each target needs to be deployed. In addition, the metadata describe each item type to be recommended. For recommended attributes, it specifies their name and type, and for recommended objects, it specifies the reference name (connecting the target class to the type of the recommended object) and the features used to identify the object. We distinguish between the reference that provides access to an item (i.e., enabling to read the item, Item.property in Figure 4.6), and the reference where to store an item (i.e., enabling writing the item, Object.writeProperty in Figure 4.6). Next, we use an example to illustrate the difference between them.

In our running example, we created a RS of attributes and operations for UML classes. For the sake of simplicity, in the remainder of this chapter, let us assume that the RS was configured to recommend only attributes. Figure 4.7 (a) shows an excerpt of the UML meta-model with the relevant parts for this (restricted) RS. [23]. The figure identifies that Class is the target element, the feature identifying Classes is their name, the recommended items are of type Property, and the feature provided when recommending an attribute is its name. In addition, Propertys are *read* via the allAttributes derived reference, but they are written on the ownedAttribute composition reference. The former contains all attributes owned and inherited by the class, and the latter only contains the owned ones. The rationale for distinguish-

23 The meta-model is slightly modified to ease understanding.




```
"services": [
  { "name": "LiteratureUMLRecommender",
    "details": [
      { "nsURI"  : "http://www.eclipse.org/uml2/5.0.0/UML",
        "context": "Targets",
        "target": { "name"    : "Class",
                    "features": [ {"property" : "name", "type" : "String" }]},
        "items":   [ { "name"           : "Property"
                       "property"       : "allAttributes",
                       "writeProperty" : "ownedAttribute",
                       "features"       : [ { "property" : "name",
                                              "type" : "String" }] }]
      }]
  }]
```

(b)

Figure 4.7: (a) Excerpt of the UML meta-model, annotated with the role of the elements in the example RS. (b) Encoding excerpt of the RS returned by the /features endpoint.

ing them is that modelling tools need to provide the items that any target object already has owned and inherited attributes in our example, available via allAttributes. However, when a recommendation is accepted, the item needs to be created and assigned to the target; in our example, ownedAttribute is used for this purpose.

Figure 4.7 (b) shows the metadata (in JSON format) that would be returned when invoking the /features endpoint on the RS. In this case, the name of the RS is *LiteratureUMLRecommender*, and the RS needs to receive all other possible targets in the model (i.e., all Classes) as context.

The second endpoint in Table 4.1 is /recommend, which allows clients to request recommendations by specifying the RS name as a path parameter, and providing a JSON file with the target of the recommendation, its current items, and its context (if needed). In addition, clients can use optional query parameters to customise the maximum number of recommendations to retrieve (*newMaxRec*), the minimum ranking value threshold (*threshold*), and the desired recommended item type when several are possible (*itemType*).

Figure 4.8 shows a recommendation request example for the RS in Figure 4.7. Part (a) shows a UML model being edited, for which the user solicits recommendation for class *Book*. Part (b) shows the encoding of the request. In this request, the target Class is named *Book*, and has two Propertys called *title* and *author*. They are encoded in the *read* feature allAttributes. As specified in Figure 4.7, the only feature that identifies Propertys is their *name*. Since the context of the RS is set to

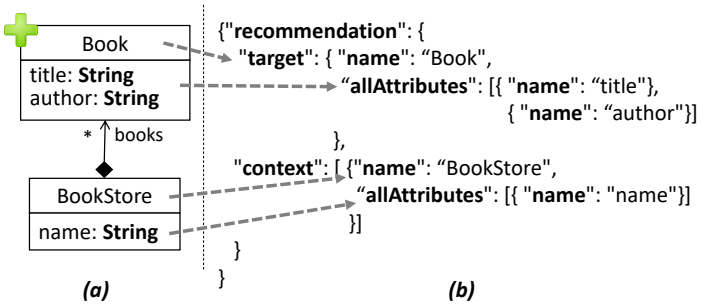


Figure 4.8: JSON representation for a /recommend request.

Targets, the recommendation request needs to include the name and attributes of all other Classes in the model. In this case, there is just one additional class named *BookStore*.

```
{"recommendations": [
    {"name": "LiteratureUMLRecommender",
     "items": [
       {"className": "Property",
        "pk": {"name": "year"},
        "value": 1},
       {"className": "Property",
        "pk": {"name": "publisher"},
        "value": 0.5}]
    } ] }
```

Figure 4.9: JSON representation for a /recommend response.

Figure 4.9 shows an example of response for the recommendation request displayed in Figure 4.8. The response consists of a list of item recommendations, calculated using the RS named *LiteratureUMLRecommender* (as specified in the request). The recommendations are for the target Class *Book*, and the recommended items are two Propertys with name *year* and *publisher*, and with relevance values 1 and 0.5 respectively.

Implementation-wise, *DroidREST* is implemented in Java using Jersey[24] and Tomcat[25]. It has three main classes: *Recommender*, which handles the requests from the clients; *ContextItem*, which parses the JSON files received by the /recommend endpoint to extract the recommendation target and its items from the modelling context; and *RecommenderGenerator*, which generates the recommendations for the given target taking its context and the query parameters into account. Since retrieving the recommendations is direct, the service has response times in the order of milliseconds.

24 https://eclipse-ee4j.github.io/jersey/
25 https://tomcat.apache.org/

4.5 INTEGRATION AND REUSE OF RECOMMENDER SYSTEMS

RSs are increasingly being used to assist developers in all sorts of software engineering tasks [134]. Modelling, a key aspect, is no exception, witnessing a surge in recommender proposals specifically designed for modelling languages [10]. These recommenders assist in constructing models or meta-models by suggesting novel attributes, references for classes, or related classes for existing ones [9, 28, 41, 46, 147, 167]. Employing diverse methods, ranging from classical content-based and collaborative filtering to natural language processing, knowledge graphs and pre-trained language models, they each present their unique strengths and weaknesses [9, 41].

Given this growing plethora of modelling recommenders, we wonder if it is possible to reuse these RSs for another modelling notation, and integrate them within a particular modelling tool. These reuse and integration pose practical challenges, including differences in modelling languages, the need to combine multiple RSs for varied item suggestions, and technical deployment considerations [147]. To tackle these hurdles, this section delves into the integration and reuse challenges of RSs for modelling languages.

So far, we have seen how to configure and deploy a RS for a modelling language using the DSL Droid, and how to obtain recommendations via the *DroidREST* service once the RS is deployed. In addition, for our proposal to be comprehensive, the last ingredient would be to be able to integrate the RSs (those defined with Droid, but also other third-party RSs) within a modelling tool. In the simplest case, which is the integration of a specific Droid recommender into a modelling tool, the developer can just manually extend the modelling tool to initiate a POST request through the `/recommend` endpoint of *DroidREST*, and then, to handle the prioritised list of recommended items that the request returns. However, in a more general setting, the integration and reuse of RSs encompasses several key dimensions, which have been succinctly represented as a feature model in Figure 4.10, following the approach introduced by Kang et al. [82].

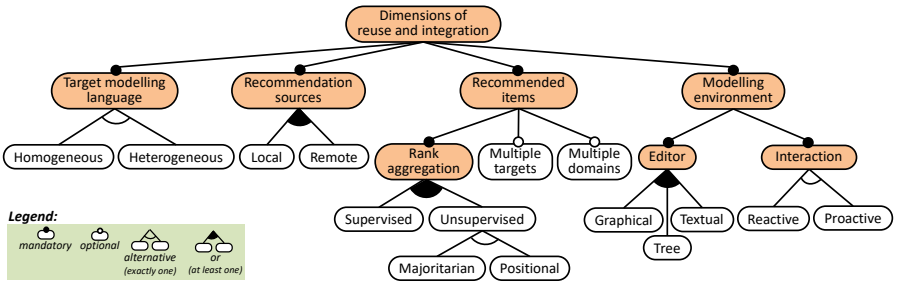


Figure 4.10: Dimensions of reuse and integration of RSs for modelling languages.

The identified dimensions are the following:

target modelling language. A RS can be integrated in modelling environments developed for the same modelling language as the RS supports (*homogeneous*), or alternatively, it can be reused for a different – albeit similar – modelling language (*heterogeneous*). For instance, a RS for meta-modelling languages like Ecore [158] may be reused for UML class diagrams, and vice versa [11]. For this purpose, a mapping between the target modelling language and the RS is needed. Having the possibility to set this bridge is useful in cases where there is not enough data (i.e., models) to train a RS for a (domain-specific) modelling language, but a RS for a similar notation exists.

recommendation sources. The recommendations may be generated *locally* if the RS is deployed on the computer where the modelling tool is running [28, 41, 46, 167]. In addition, recommendations may come from services deployed on a *remote* server [11, 147]. An example of the latter option is our recommendation service *DroidREST*. This option is more flexible, as it allows reusing recommenders within different tools, and aggregating recommendations from several sources.

recommended items. Integrating several RSs within a same modelling environment enables the recommendation of items for *multiple targets* (e.g., for both classes and interfaces in class diagrams) and for *multiple domains* (e.g., RSs for medical, banking or transportation domains). When combining several RSs for the same type of target and item, the rankings of their recommendations need to be aggregated. As explained in the background (Subsection 2.1.3), the approaches to recommendation *rank aggregation* are broadly classified into *unsupervised* and *supervised*. The former can be further categorised as *majoritarian* and *positional*. When computing a numerical score for each item in the aggregated recommendation list, positional methods use the absolute position of the item in the individual rankings, while majoritarian methods compare pairwise each item [118]. Unsupervised methods are generally simple, efficient and flexible. However, if ground truth data are available, supervised methods may be more effective. These methods can use a variety of techniques, like learning to rank [96] or genetic programming [164].

modelling environment. RSs need to be integrated into concrete modelling tools, typically offering *graphical*, *tree* and/or *textual editors*. The *interaction* with the RS may either be activated explicitly by the user (*reactive*) or be *proactive*, offering suggestions to the user when deemed appropriate (e.g., as in [98]).

Based on these dimensions, we propose an automated approach for the reuse and integration of RSs into modelling tools. The approach

is enabled by a dedicated tool called IronMan, which operates independently of Droid, and currently targets Eclipse-based modelling tools.

4.5.1 *Overview of the approach*

IronMan

To automate the steps needed to reuse and integrate RSs within a modelling tool, we have developed an Eclipse plugin called IronMan (Integrating RecOmmeNders for Modelling lANguages). This guides in all steps of the integration task, including the discovery and selection of available RSs, the adaptation of a RS to the modelling language utilised in the modelling tool (if needed), the configuration of the aggregation method in case of selecting multiple RSs, and the integration of the resulting recommender within the modelling tool (currently, Sirius- or tree-based Eclipse editors).

A comprehensive methodology for integrating and reusing RSs has been devised, as illustrated in Figure 4.11. This methodology effectively addresses all the dimensions of integration outlined in Figure 4.10.

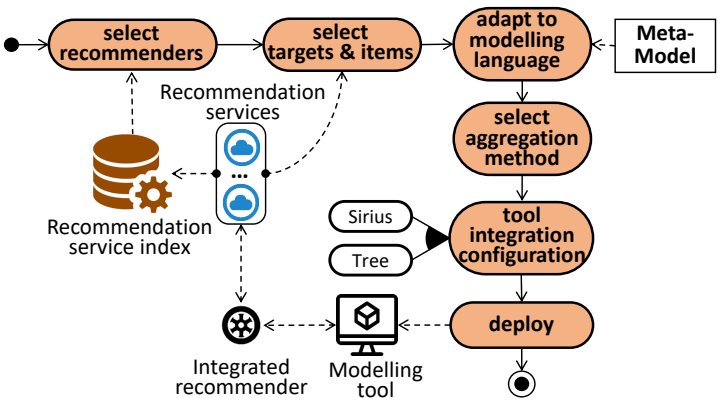


Figure 4.11: Overview of our methodology for RS integration.

In a first step, our approach relies on deploying the RSs as services conformant to a reference REST API that provides a uniform mechanism for requesting recommendations and accessing the features of the RSs. This simplifies the integration of arbitrary RSs into modelling tools, avoiding the need to build custom, heterogeneous integrations. In particular, we propose as reference API for recommendation services the one described in Table 4.1. The recommenders created with Droid comply with that API, but the approach also works for any other RS as long as it complies with the API. This way, the first step in the integration consists in discovering the available RSs by means of a RS indexer. The indexer can filter the available services by diverse criteria, like the modelling language for which the RSs provide suggestions.

In a second step, the developer selects the recommendation targets and items (a subset of those provided by the RSs selected in the previous step). If several RSs of the same kind of items are chosen, the user will need to select an aggregation method for the recommendations. Moreover, if the modelling language where the RSs are to be integrated differ from the language supported by the RSs, then the developer will have to adapt the RSs via a mapping.

As a last step, the developer configures the integration with the modelling tool. Currently, we support the adaptation of EMF-based modelling languages, and the integration with Sirius graphical editors and tree editors. However, our extensible architecture facilitates future integrations with other technologies, like Xtext [23].

After performing these steps, our approach automatically integrates the assembled RSs within the modelling tool. The result is a plugin that communicates with the selected RS services to obtain recommendations, aggregating and adapting them to the modelling language.

4.5.2 *Recommendation indexer*

As previously explained, the first step in the approach is discovering and selecting the RSs of interest. For this purpose, we have developed a recommendation service indexer API that provides a standardised method to *register* and *update* recommendation services, and to *explore* the registered services, enabling clients to *discover* and access the recommendation services that best suit their needs.

Recommender indexer

Table 4.2 shows the REST endpoints of the indexer, which enable clients to register, update, explore, and discover services. The `/register` endpoint allows registering new recommendation services in the indexer. To register a service, clients need to provide its URL using the `/register?urlName=`⟨*url*⟩ endpoint, where ⟨*url*⟩ is the URL of the recommendation service. Several RSs can be placed within the same URL, and the registered URLs must define the endpoints defined in Table 4.1. In particular, upon registering a recommendation service, the indexer invokes its `/features` endpoint (cf. Table 4.1) to cache the characteristics of the RS.

The `/updateRegistration` endpoint allows clients to update a previously registered service. Similar to `/register`, clients only need to provide the URL of the service to be updated using the `/updateRegistration?urlName=`⟨*url*⟩ endpoint. As before, the indexer will then invoke the `/features` endpoint of the recommendation service.

The `/services` endpoint returns the list of all registered recommendation services and their metadata in JSON format.

The last endpoint of the indexer, `/discover`, allows searching for deployed services using either the *name* or the *nsURI* of the RS. The *nsURI* is a unique identifier for meta-models, which is standard in modelling technologies like EMF [158]. This way, the API returns a

Table 4.2: Endpoints of the recommender indexer API.

Endpoint	/register?urlName=⟨*url*⟩
Description	Registers a new recommendation service.
Method	POST
Output	Ok/Error
Endpoint	/updateRegistration?urlName=⟨*url*⟩
Desc.	Updates a registered recommendation service.
Method	POST
Output	Ok/Error
Endpoint	/services?(nsURI=true)
Desc.	Returns all registered recommendation services and their metadata. The optional query parameter nsURI=true groups services by *nsURI*.
Method	GET
Output	Available recommendation services (JSON).
Endpoint	/discover?(name=⟨*name*⟩ | nsURI=⟨*uri*⟩)
Desc.	Searches for registered recommendation services with the given RS name or meta-model *nsURI*.
Method	GET
Output	Registered recommendation services that match the search criteria (JSON).

JSON list with all recommenders with the given name or defined over a meta-model with the provided *nsURI*.

4.5.3 *Adaptation of recommender systems to the modelling notation*

In cases where we want to reuse a previously developed RS designed for a specific modelling language, and integrate it into a modelling environment that utilises a slightly different language (albeit perhaps similar), adaptation is required. This adjustment aligns the RS with the modelling notation used in the new environment. We facilitate the reusability of RSs tailored to a particular modelling language for other notations through the use of structural mapping.

Our approach to reusing a RS, designed for a modelling notation different from the one used within the current modelling tool, involves establishing a structure-preserving mapping $m : RS \to MM$ between the classes and features used by the RS, and the elements of interest in the language meta-model MM.

Figure 4.12 exemplifies a mapping that adapts the RS for UML – which recommends Propertys for Classes – to Ecore. The adapted RS will then recommend EAttribute for Ecore EClasses. RS on the left shows an excerpt of the UML meta-model containing the elements designated as targets, features and items (cf. Figure 4.6). The map-

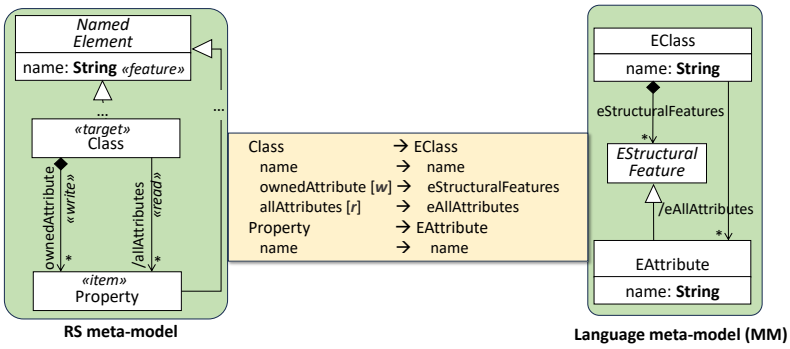


Figure 4.12: Adapting the RS to the modelling language.

ping maps the UML meta-model elements playing a role in the RS to corresponding elements in the Ecore meta-model MM to the right. In this mapping process, we adapt UML elements to correspond with Ecore elements. Specifically, the Class is mapped to EClass, the feature name is mapped to name, ownedAttribute to eStructuralFeatures, and allAttribute to eAllAttribute. Additionally, the Property class is mapped to EAttribute, with the feature name also being mapped to name.

Not any mapping is valid, but well-formed mappings need to preserve the structure of the source meta-model. For this purpose, we build on the notion of *binding*, which has been used to express generic model operations [92, 141]. Next, we use predicates $item(\_)$, $feature(\_)$, $property(\_)$, $writeProperty(\_)$, and $target(\_)$ to denote the role of the element in RS; predicate $relevant\ (e) = item(e) \vee feature(e) \vee property(e) \vee writeProperty(e) \vee target(e)$ to identify the elements that need to be mapped; $src(r)$ and $dest(r)$ for the source and destination class of reference r; and $c_i \leqslant c_j$ to denote that c_i is compatible with c_j (a subclass, or c_j itself).

This way, a mapping $m : RS \to MM$ is well-formed iff it fulfils the following conditions:

- **Definition domain:** m is defined exactly for each element e of RS s.t. $relevant(e)$.
- **Classes:** If c is a class in RS s.t. $relevant(c)$, then $m(c)$ is also a class in MM.
- **Class subtyping** is preserved and reflected: Given classes c_1 and c_2 of RS s.t. $relevant(c_1) \wedge relevant(c_2)$, then $c_1 \leqslant c_2 \iff m(c_1) \leqslant m(c_2)$.
- **Attributes:** If a is a relevant attribute defined or inherited by a relevant class c in RS, then $m(a)$ is also an attribute inherited or defined in class $m(c)$. The type of the attribute must be preserved or generalised in the mapping: $a.type \leqslant m(a).type$. For instance, an attribute of type *integer* can be mapped to a *double*.

- **References:** If r is a relevant reference from class c_1 to c_2 in RS, then $m(r)$ is also a reference from class c_1' to c_2' in MM, with $m(c_1) \leqslant c_1'$. In addition, we need a further constraint for $dest(r)$, which depends on whether r is *read* ($property(r)$) or *write* ($writeProperty(r)$):
  - $property(r)$: Any relevant superclass of $dest(r)$ (including $dest(r)$, if it is relevant) is mapped to a superclass of c_2', or to c_2': $\forall c_i \in RS \cdot dest(r) \leqslant c_i \wedge relevant(c_i) \implies c_2' \leqslant m(c_i)$
  - $writeProperty(r)$: Any relevant class compatible with $dest(r)$ is mapped to a class compatible with c_2': $\forall c_i \in RS \cdot c_i \leqslant dest(r) \wedge relevant(c_i) \implies m(c_i) \leqslant c_2'$
- **Composition** is preserved: If r is a relevant write composition in RS, then $m(r)$ is also a composition.

The condition for references enables a reference r to be declared exactly on the mapped class, or in a superclass (so that it is inherited). Similarly, the destination of the reference r can be the relevant class, a subclass (when $property(r)$), or a superclass (when $writeProperty(r)$), which then should be mapped preserving subtyping.

Typically, references that are *write* (allowing adding an item to a target) are composition references in RS, which needs to be preserved in the target by the last well-formedness condition. The mapping does not care about the cardinality of attributes and references, as it does not need to be preserved.

The mapping of Figure 4.12 is well-formed. This is so as both Class and Property in the UML meta-model are mapped to classes in the Ecore meta-model (EClass and EAttribute), and their attributes (Class.name and Property.name, which are actually inherited) are mapped to attributes of the target classes. Both references ownedAtttribute and allAttributes are mapped according to the conditions, e.g., m(eStructuralFeatures) = eAllAttributes, the source of both references coincide (m(Class) = EClass), and for the destination, m(Property) = EAttribute. Reference ownedAttribute is a *write* feature and a composition, and so is eStructuralFeatures.

Our mapping enables consistent adaptations between structurally similar (but not identical) meta-models. For more complex mappings, our correspondences could be extended with an expression language – like OCL – able to calculate derived elements in the target, or adapt attribute values. This is left for future work.

4.5.4 *Recommendation aggregation*

When combining several RSs for a modelling language, two scenarios can arise. In the first scenario, the RSs to combine tackle *different targets* (e.g., one RS provides recommendations for classes, and the other one for packages) or *different kinds of items* (e.g., one RS recommends attributes, and another operations). In such cases, no aggregation is

needed, since the items are completely disjoint. This way, the composed RS would just use a different recommender for each kind of item, returning the lists of ranked items with no modification.

In the second scenario, multiple RSs recommend the *same kind of items* for a given target (e.g., two RSs of class attributes that use different recommendation methods or different datasets). This scenario requires rank aggregation techniques to obtain a consensus ranking containing a subset of their items.

In this thesis, the aggregation of multiple RSs for the same item type is performed by means of unsupervised techniques, and more specifically, by score-based positional techniques, due to their simplicity and efficiency. These techniques sort the items based on their absolute position in the individual rankings. They receive as input a set of individual rankings, and use an aggregation function $f\colon U \times I \rightarrow R$ to combine the item position-based scores, being U and I the sets of users and items in the system, respectively [118].

In particular, we use BC and MRA as recommendation aggregation techniques (cf. Subsection 2.1.3). In our context, which is recommendation for modelling languages, BC works as follows. Let n be the number of unique items in the recommendation lists to be aggregated. For each individual recommendation list, the top ranked item is given n points, the second ranked item is given $n-1$ points, and so on. If a recommendation list does not include all candidate items, the remaining points are split evenly among the unranked items. Then, the scores of the items in each list are summed, the items are sorted in descending order, and the aggregated ranking consists of the top-N items (for a given N). MRA, by contrast, sorts the items according to the median of the position the items received in the individual rankings.

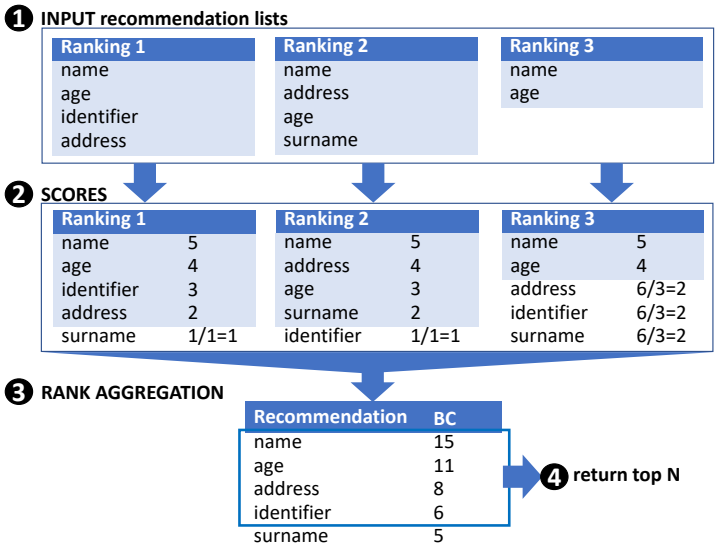


Figure 4.13: Rank aggregation example using Borda Count.

Figure 4.13 illustrates how BC technique works for the aggregation of three hypothetical class attribute recommenders. The upper part (label 1) shows the rankings of attributes that each RS suggests for a class named *Person*. The middle part (label 2) depicts the scores assigned to the attributes in each ranking. Since there are 5 unique (non-duplicate) attributes, the score of the first attribute in each ranking is 5, and this score is decreased for the subsequent positions of the ranking. The items that a RS does not suggest (e.g., *surname* in Ranking 1) receive equal portions of the remaining available points from the RS. The bottom part of the figure (label 3) shows the aggregated score of each item, calculated as the sum of the individual scores. Step 4 shows the returned top N list of recommendations. Incidentally, MRA would output the same aggregated rank, e.g., the rank of `name` is 1, which is the median of its positions in the three rankings.

Section 5.2 will provide details on the tool implementation of the presented approach to the reuse and integration of RSs into modelling environments.

4.6 SUMMARY

Our approach aims to facilitate the adoption of RSs in MDE by proposing a model-driven solution called Droid. This incorporates a DSL that automates the synthesis of RSs for specific modelling languages. The solution supports the collection of data for training and testing. Using the DSL, RS developers define the items to be recommended, preprocess the data, and train candidate RSs. Droid then automatically evaluates the RSs based on configurable metrics. The selected RS is deployed as a REST service (DroidREST) for easy integration with various modelling clients.

The Droid DSL offers four data preprocessing options: special character removal, edit distance merging, minimum ratings per item, and minimum ratings per target. RS developers can evaluate different combinations to choose the most suitable one for their specific requirements. The training process involves data splitting and building recommendation models, with options like CrossValidation and Random techniques. Seven recommendation methods are supported, but the tool architecture is extensible with additional methods and data encodings through extension points.

As the final step, the Droid DSL facilitates the selection of seven metrics to evaluate the candidate RSs (Precision, Recall, F1, nDCG, ISC, USC, and MAP). Additionally, it allows the definition of cutoff values, the maximum number of recommended items, and the relevance threshold. After evaluating the candidate RSs, the chosen RS is seamlessly deployed on the DroidREST recommendation service API. Key endpoints include "/features", providing metadata for all

deployed RSs, and "/recommend", enabling clients to request recommendations.

For the reuse and integration of RSs within Eclipse modelling tools, our approach uses IronMan. IronMan guides users in discovering and selecting available RSs, adapting RSs to the modelling language used in the tool, configuring aggregation methods for multiple RSs, and integrating the resulting recommender within the tool. Its flexible architecture enables the integration with various modelling technologies and the definition of new aggregation methods.

5 TOOL SUPPORT

This chapter provides a comprehensive overview of the tool developed to support the concepts introduced in Chapter 4. The chapter is composed of two parts. In the first part, Section 5.1 presents the Droid framework, highlighting its architecture and showing its structural design and components. Subsection 5.1.1 explores the extensibility options available in Droid, which allow users to customise and extend the framework according to specific necessities. Subsection 5.1.2 focuses on the functionalities tailored for RS developers, presenting the tool and features that facilitate the configuration of each step in the construction of RS for particular modelling languages. In the second part, Section 5.2 presents the IronMan framework, highlighting its architecture and showcasing its structural design and components. Subsection 5.2.1 delves into the extensibility features provided by IronMan, empowering users to tailor and expand the framework according to their requirements. Subsection 5.2.2 provides details on the IronMan plugin, including the recommendation services (5.2.2.1) and the integration of recommendations within client modelling tools on Eclipse (5.2.2.2). Finally, in Section 5.3, the chapter concludes with a summary.

5.1 DROID FRAMEWORK: CREATION OF RECOMMENDER SYSTEMS

Architecture of Droid framework

Droid is a framework that facilitates the construction of RSs for modelling languages of interest. Figure 5.1 depicts the architecture of the Droid framework. It consists of three main components: the Droid *Configurator*, the Droid *Service*, and the *Clients*. Additionally, the framework offers three extension points: DataCollection, DataEncoding, and RecommendationMethod.

The Droid *Configurator* is an Eclipse plug-in that allows the RS developer to configure, evaluate and synthesise RSs for particular modelling languages. It provides an Eclipse textual editor for the DSL presented in Section 4.3, enabling the configuration of candidate recommenders for a given modelling language (label 1). The configuration specified with the DSL serves as input to the *Data Preprocessor* (label 2). The results of each configured data preprocessing option are displayed in a dedicated view, where the RS developer can select the desired preprocessing option to apply (label 3). Similarly, the *RS Evaluator* takes the specified RS configuration as input (label 4), and leverages the external libraries RankSys [165] and RiVal [139] to evaluate the selected recommendation methods. The RS developer can

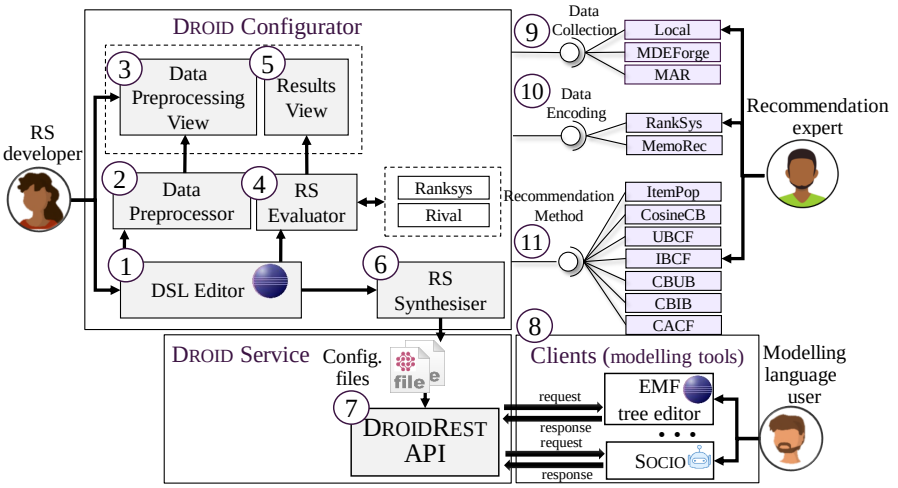


Figure 5.1: Architecture of DROID.

choose metrics to be computed for each RS, and the evaluation results are reported in an Eclipse result view (label 5). For more details about the DROID *Configurator*, refer to Section 5.1.2.

Based on the reported metrics, the RS developer can select the preferred RS, and the *RS Synthesiser* generates a set of configuration files from it (label 6). These files are uploaded into a REST API that implements a generic recommendation service, called *DroidREST* (label 7). This generic service can be customised for specific modelling languages using the configuration files that the *RS Synthesiser* produces. It employs a JSON-based model representation that allows clients to request recommendations, and the recommendations are sent as a response.

On the client side, any modelling tool can utilise the DROID *Service* to obtain recommendations and make them available to the users of the modelling language (label 8). Additionally, for the automated integration of RSs into existing modelling tools (currently, Eclipse tree-based editors and Sirius-based graphical editors), our tool IRONMAN can be leveraged. Section 5.2 provides further details on IRONMAN.

Finally, the framework offers three extension points (labels 9–11) that allow recommendation experts to extend DROID according to their needs. These extension points are:

- **DataCollection**: It enables the integration of additional data sources or the modification of the data collection process according to specific requirements.
- **DataEncoding**: It facilitates the incorporation of alternative data encoding methods or the adaptation of existing ones to accommodate specific data characteristics.
- **RecommendationMethod**: It enables the integration of new recommendation methods into DROID. Experts can implement and

incorporate their own recommendation algorithms or customise existing ones to suit their needs.

By offering these extension points, DROID provides a flexible and customisable architecture that empowers recommendation experts to enhance and adapt the framework according to their domain-specific requirements. The next section provides more details on these extension points.

5.1.1 *Extensibility options*

Extension points are the mechanism that Eclipse provides to extend the behaviour of a program in an external way. They declare a contract that extensions must conform to. At runtime, an extended program can query for existing extensions implementing an extension point, and invoke them as needed.

DROID can be extended via three extension points (cf. Figure 5.1) enabling the incorporation of new data sources (DataCollection) (label 9), data encodings (DataEncoding) (label 10) and recommendation methods (RecommendationMethod) (label 11).

Data collection extension point

Figure 5.2 depicts the extension point to define new data sources in DROID. Specifically, adding a new data source to DROID requires providing the name of the data source, the URL of the source, and a Java class implementing the interface IDataSource, with details on how to collect the data. The IDataSource interface must implement a definePage() method with the following parameters: the path of the location for the temporary download of the data, the url from where the data will be retrieved, and the selection of elements to be used.

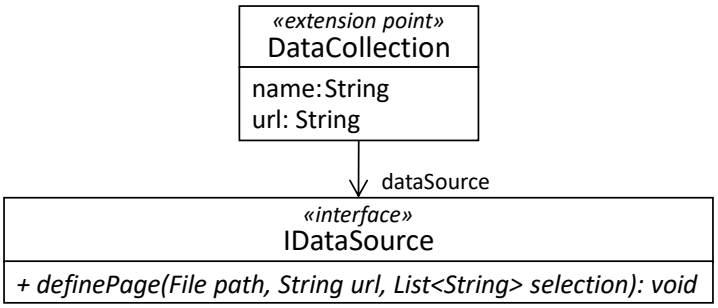


Figure 5.2: Data source extension point.

DROID currently integrates three data sources, implemented as extensions conforming to this extension point. The first one allows gathering data via browsing local directories. The second one gathers data by accessing the MAR model search engine [70], which returns models satisfying an input set of keywords. The third one retrieves models from the MDEForge (meta-)model repository [20], which supports queries via keywords, a minimum model size, and quality metrics that the models must satisfy.

Data encoding extension point

The second extension point allows the definition of data encodings. They represent the interactions between targets and items in some format – typically as (rating) matrices – which recommendation methods use for training and evaluation. Since each recommendation method requires a specific data encoding, the extension point enables the declaration and implementation of a given data encoding. As the class diagram in Figure 5.3 shows, the extension point requires providing the name of the encoding technique, the library it belongs to, a description of the data encoding, and an implementation of the Java interface IDataEncoding with the methods setPath() and generateDataEncoding(). The method setPath() receives as parameters the project and path for generating the *droid* file in preparation for the Java class generation. The method generateDataEncoding() receives the name of the library that performs the data encoding, and its implementation must define an Xtend class of type generator that provides, via its doGenerate() method, the code for the Java class generation.

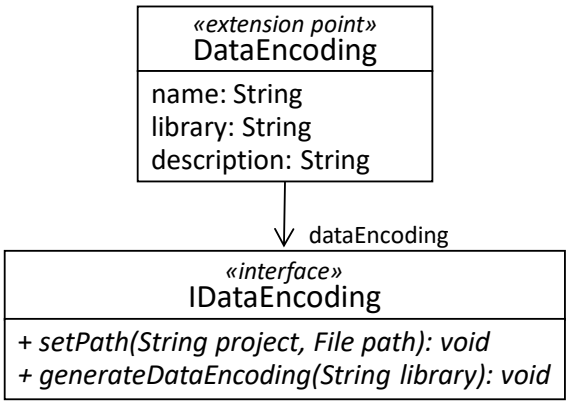


Figure 5.3: Data encoding extension point.

Currently, Droid has two extensions conforming to the DataEncoding extension point: RankSys (which supports the data encodings binary, frequency, and normalised frequency) and MemoRec. In RankSys, the three data encodings consist of matrices with the targets as rows, and the items as columns. In the binary encoding, each matrix cell is set to 1 if the corresponding target and item are "related" (e.g., a class contains certain attribute), and 0 otherwise; in the frequency encoding, a cell contains a real value that weights the relationship between a target and an item (e.g., the number of classes with same name in different models that contain a given attribute); and in the normalised frequency encoding, a cell contains the frequency value normalised to some range (usually [0,1]). On the other hand, the data encoding of MemoRec consists of pairs in the format target #item for each item found within a class. This encoding is employed to suggest additional items within a given context class [41].

Recommendation method extension point

The last extension point, shown in Figure 5.4, allows adding recommendation methods. Implementing this extension point entails pro-

viding a Java class (implementing the interface IRecommendationMethod) that invokes the associated recommendation method; the name of the used recommendation library; the method name and its category; the name of the data encoding technique used by the method (which must be the name of a DataEncoding extension); a description; an example of use (to be displayed in the editor to help the end-user, like UBCF("5", "10")); and the name and type of the method parameters. The IRecommendationMethod interface mentioned above must implement the methods setGenPath() and generateMethod(). The method setGenPath() receives the project and path for generating the *droid* file in preparation for the Java class generation. The method generateMethod() receives the name of the library and method of the recommendation method, and the data encoding type to be used. In addition, this method must define an Xtend class of type generator whose method doGenerate() provides the code for the Java class generation.

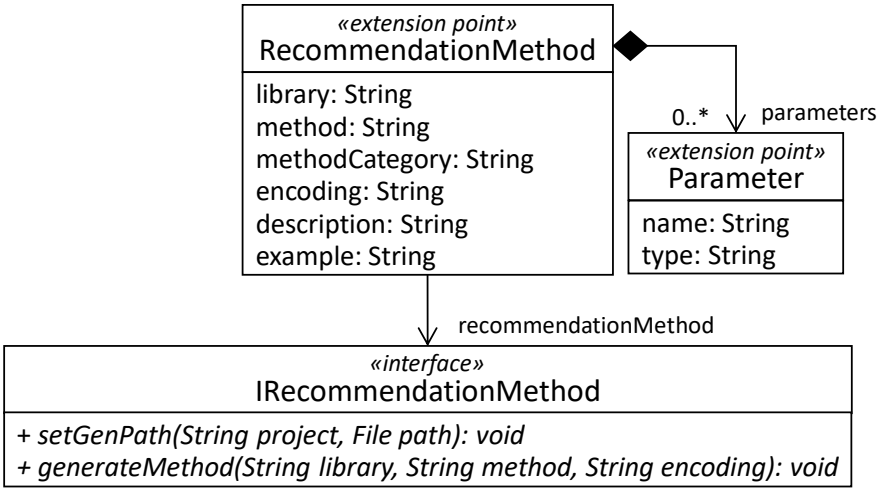


Figure 5.4: Recommendation method extension point.

Currently, DROID supports seven recommendation algorithms from two libraries, which have been integrated using this extension point. From MemoRec [41], it supports CACF, and from RankSys [165], it supports ItemPop, CosineCB, UBCF, IBCF, CBUB and CBIB.

5.1.2 DROID *configurator*

Droid wizard

The DROID *Configurator* is an Eclipse plug-in designed to help RS developers build RSs for modelling languages. It is available at `https://droid-dsl.github.io/`. The configurator includes a wizard for the creation of DROID projects in four steps, illustrated in Figures 5.5 and 5.6.

Technology selection

In steps 1 and 2, shown in Figure 5.5, the RS developer selects a new DROID project to be created (label 1). Then, the wizard requires specifying the name of the new RS, and the modelling (UML, XMI)

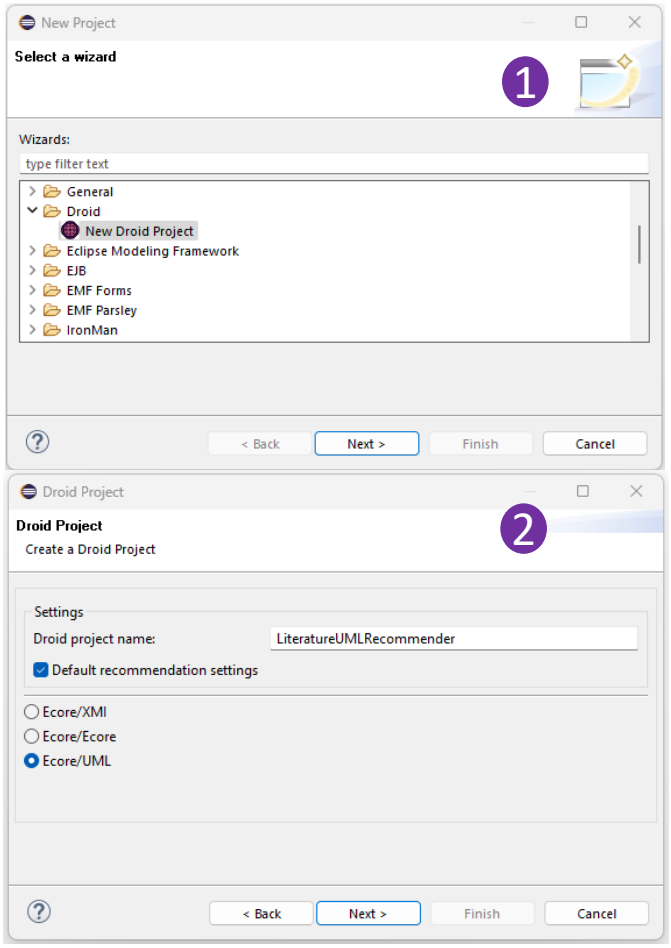


Figure 5.5: Wizard in action: (1) Selecting new Droid project, (2) Droid project basic configuration.

or meta-modelling (Ecore) technology that the RS will serve (label 2). To ease the configuration of the RSs, it also provides an option to automatically generate a default Droid configuration file with typical values. In steps 3 and 4, showcased in Figure 5.6, the wizard allows the RS developer to choose one of the available data sources to collect models and meta-models for training and evaluating the candidate RSs (label 3). After a data source has been chosen, the wizard presents a new page where users can provide additional information about the data to be collected. The content and options presented on this page will vary depending on the data source selected. For instance, label 4 in Figure 5.6 exemplifies a query designed for the MAR search engine. Finally, clicking on the button *Finish* creates the project.

Droid editor

After the Droid project has been created, the RS developer can configure the RS to be built using the Droid Editor. Figure 5.7 shows a screenshot of the Droid *Configurator* environment.

The editor (label 1) allows the configuration of RSs through the DSL presented in Section 4.3. This editor was built using Xtext, and features syntax highlighting, autocompletion, and markers for errors

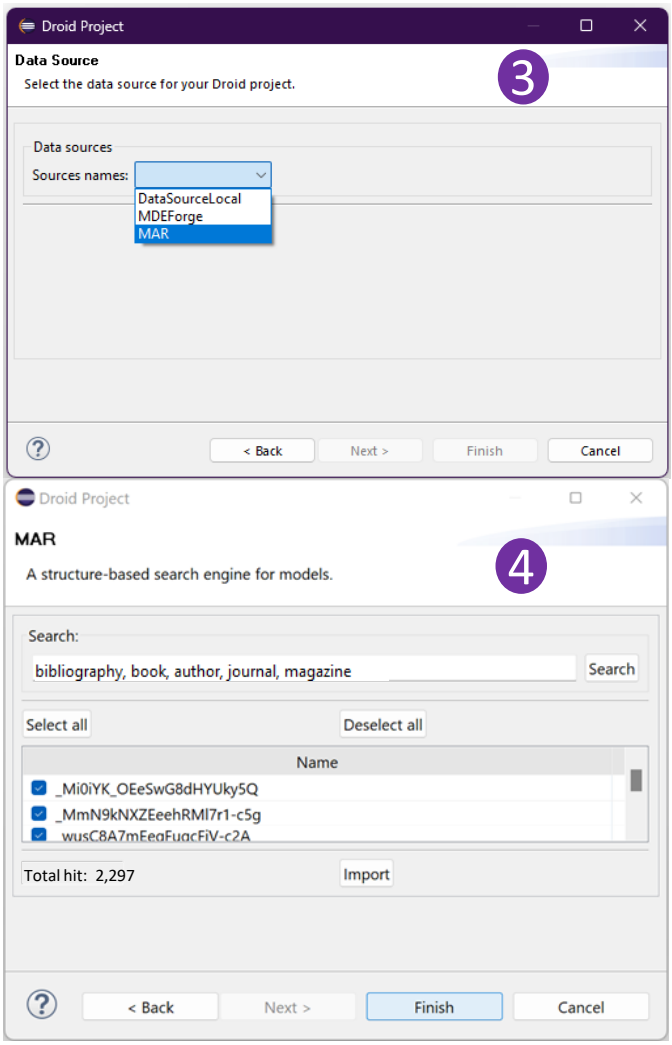


Figure 5.6: Wizard in action: (3) Selecting data source, (4) MAR repository data search configuration.

and warnings. A pop-up window (label 2) presents autocompletion options by listing the possible items that can be recommended for a given target. For instance, for the running example, it shows the properties that the meta-class Class from the UML meta-model declares. The figure shows an excerpt of the UML meta-model (label 3) with arrows to the elements selected in the screenshot.

Data Preprocessing view

The *Data Preprocessing* view at the bottom helps in understanding the dataset, and displays information of each preprocessing configuration. The data section (label 4) provides a user-friendly *Pre-Process* button that facilitates the execution of the preprocessing steps. In this section, the user can obtain essential information about the dataset, namely, the total number of models, loading models (i.e., not broken), and well-formed models; together with the minimum, maximum and average model size (measured as the number of model elements). The target/items section (label 5) shows the number of targets and items

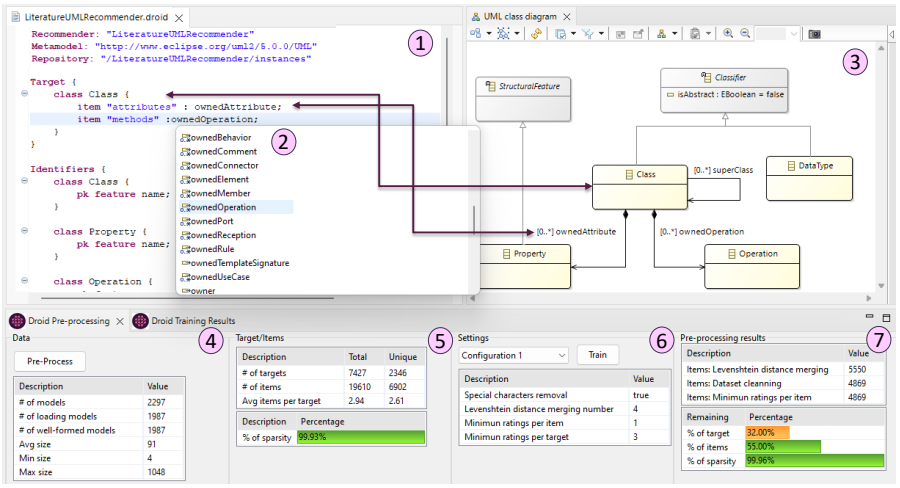


Figure 5.7: Screenshot of the Droid *Configurator*.

(total and unique), the average number of items per target, and the percentage of sparsity (i.e., the percentage of the target-item matrix that is not populated). The settings section (label 6) displays the options of each preprocessing configuration, which concern the removal of special characters, the Levenshtein distance for item merging, and the minimum rating per target and per item (cf. Subsection 4.3.2). Finally, the panel with label 7 shows the preprocessing results for the preprocessing configuration selected in the settings section (*Configuration 1* in the figure). The upper table reports the number of remaining items after applying each preprocessing configuration, and the bottom table displays the percentage of targets and items left after cleaning the data.

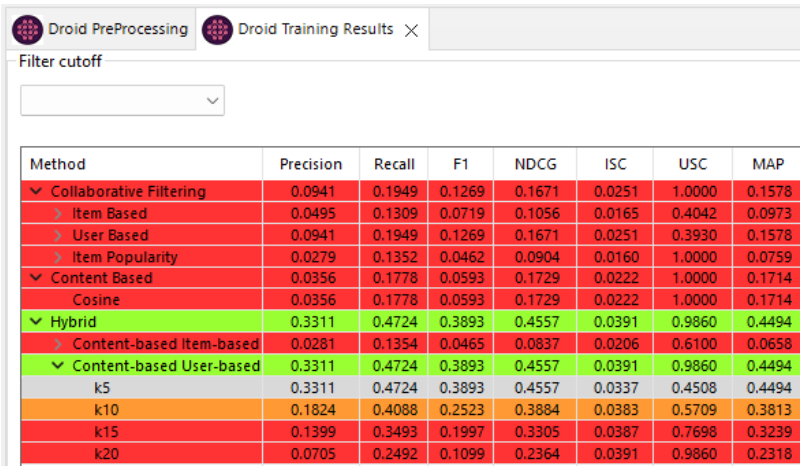


Figure 5.8: Training results view of the Droid *Configurator*.

Training of recommender systems

To train the candidate RSs, the RS developer needs to choose a preprocessing configuration, and click on the button *Train* (label 6 in Figure 5.7). This opens the *Training Results* view of Figure 5.8, which

shows metrics for all trained candidate RSs. They are displayed in a drill-down table, where each row corresponds to a recommendation method (of those selected in the *droid* file), and each column to a selected metric.

Results view

The table displays the method categories (e.g., Collaborative Filtering, Hybrid) according to the extension point details. Within each category, the recommendation methods that were selected using the Droid DSL are listed, and for each method, there is a block of rows corresponding to the values of each method parameter, if any. Columns 2–8 display the value of the metrics selected in the DSL. As an example, the Hybrid category includes two methods: Item-Based Collaborative Filtering with Content-Based Similarity (CBIB) and User-Based Collaborative Filtering with Content-Based Similarity (CBUB). These methods have one parameter, named *k*, which is the neighbourhood size. Consequently, a block is displayed with one row for each specified value of *k* in the DSL. For CBUB, the parameter *k* takes the values 5, 10, 15, and 20. To facilitate understanding, the colour of the rows depends on the *F1* value achieved by the methods: green for methods with an *F1* value in the top 20% values, red for methods whose *F1* value is below the median, and orange for the rest.

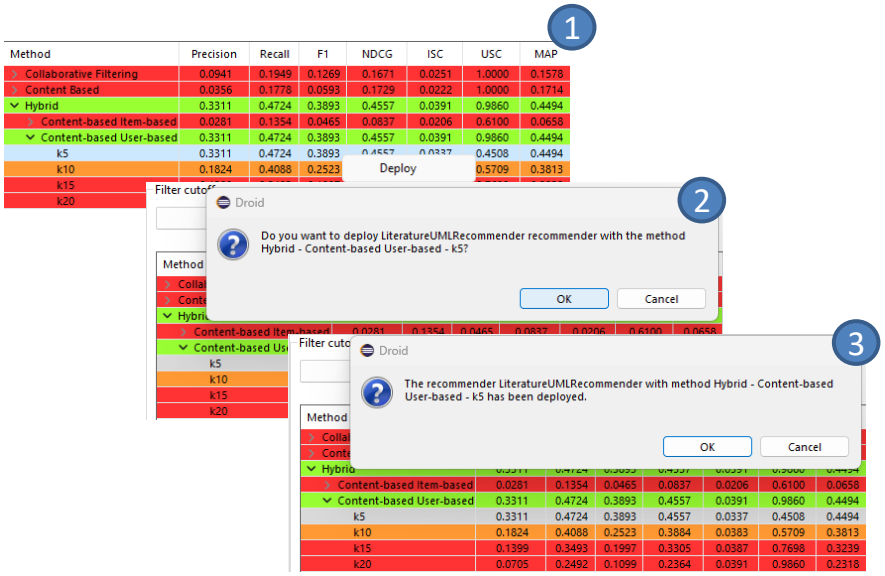


Figure 5.9: Deployment of the RS trained with Droid.

Figure 5.9 shows how to deploy a RS from the *Training Results* view. The deploy option appears by right-clicking on the RS to be deployed (label 1). Then, the RS developer confirms the selected RS (label 2). After this, Droid synthesises a set of configuration files, which are uploaded to a server. These files store the information needed to generate the recommendations, such as the items to recommend for a given target and context. This way, our generic recommendation service *DroidREST* can compute the recommendations based solely on

Deployment of Recommender Systems

these configuration files. This design facilitates re-deploying the RSs defined with Droid by just uploading new configuration files, without changing the server or the client code. Lastly, the RS developer is shown a confirmation of the deployment (label 3).

Our decision to deploy the RSs as services (instead of their direct deployment within a specific modelling tool) decouples the recommendation computation task from the modelling environment. This way, the same RS can be reused from different modelling tools, even if developed with technologies other than Eclipse. As part of the validation of the tool, in Chapter 8, we will showcase examples of modelling tools where RSs created with Droid have been integrated.

5.2 IRONMAN FRAMEWORK: REUSABILITY AND INTEGRATION

RSs created with Droid can be manually integrated with third-party modelling tools via the service *DroidREST*. Additionally, we support the automated integration of the assembled RSs within Eclipse modelling editors by means of the IronMan framework. We have realised IronMan as an extensible Eclipse plugin. Its source code is available at: https://github.com/lissetteag/integrate-recommenders-ironman.

IronMan automates the integration of existing recommenders within existing Sirius and tree-based modelling editors [158]. Moreover, it can bridge recommenders created for a modelling language for their reuse with a different language, and can aggregate the recommendations of several RSs. Interestingly, the tool is independent of Droid, meaning that it can be applied to Droid recommenders, but also to any recommender conformant to the reference API in Table 4.1.

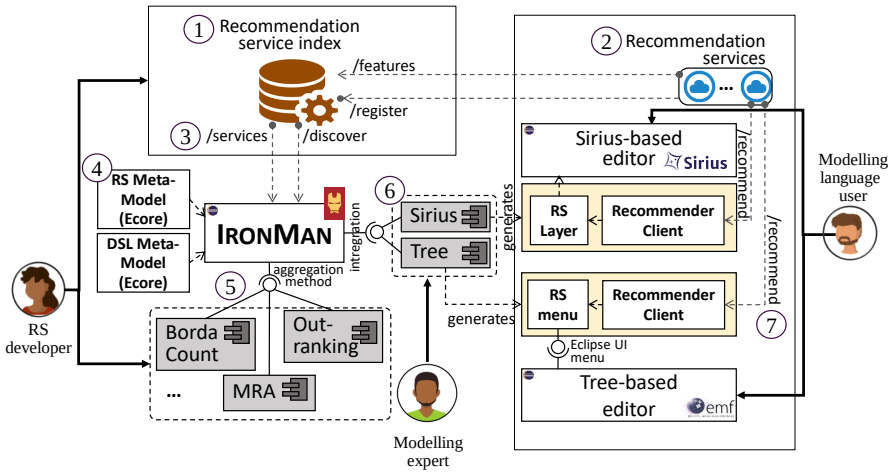


Figure 5.10: Architecture of IronMan.

IronMan architecture

Figure 5.10 shows a schema of the architecture of IronMan. To make the access to arbitrary recommenders homogeneous, it relies on our proposed reference recommendation indexer service (label 1)

(cf. Table 4.1 in Section 4.4) and recommendation service (label 2) (cf. Table 4.2 in Section 4.5), which allow RS developers to index recommenders to later on allow investigating their properties, and obtaining recommendations likely coming from several sources. The plugin uses the /services and /discover endpoints (label 3) of the recommendation indexer to obtain the available RSs and filter them by meta-model name. IronMan supports the adaptation of the RS to modelling languages other than the one targeted by the RS, by enabling the definition of a structural mapping between their meta-models (label 4), as described in Subsection 4.5.3. Additionally, IronMan provides two extension points (label 5 and 6):

- **AggregationMethod**: It enables the integration of additional rank aggregation methods or the modification of the existing rank aggregation methods.
- **Integration**: It defines code generators that create the necessary code to integrate existing RSs within a third-party modelling tool of a given technology.

By providing these extension points, IronMan represents a flexible and customisable framework, enabling RS developers and modelling experts to augment and tailor a RS for meeting their specific needs. The subsequent sections delve into a deeper exploration of these extension points and the IronMan plugin.

5.2.1 *Extensibility options*

IronMan can be extended through two extension points (cf. Figure 5.10), allowing for the incorporation of new recommendation aggregation techniques (AggregationMethod) (label 5) and integrations into Eclipse modelling tools (Integration) (label 6).

Recommendation aggregation extension point

Figure 5.11 shows the extension point for adding new aggregation techniques to IronMan. Specifically, it is necessary to provide the name of the aggregation technique, and an implementation of the IMetaSearchAlgorithm interface that encodes how to aggregate recommendations from multiple sources. The IMetaSearchAlgorithm interface defines the method algorithm(). This receives the Map parameter recommendations, which maps each source of recommendations to the list of item recommendations from that source. The method returns a Map assigning to each item a double value corresponding to the new ranking value after the aggregation.

IronMan currently integrates three aggregation techniques, implemented as extensions of this extension point: Borda Count, MRA and Outranking [56] (cf. Subsection 2.1.3).

Integration extension point

The second extension point allows the definition of integrations of RSs into Eclipse modellling tools developed with a particular technology. That is, these integrations inject recommendation facilities into

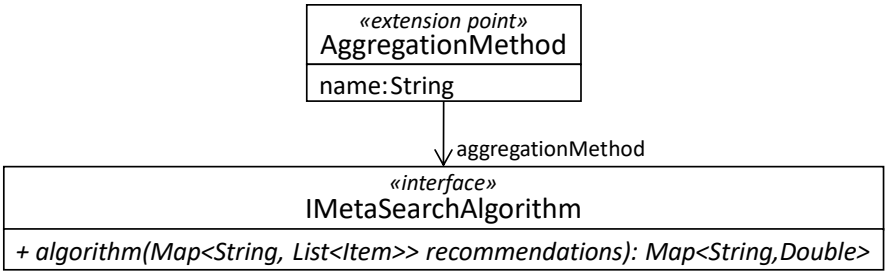


Figure 5.11: Recommendation aggregation extension point.

modelling tools of a given technology. Figure 5.12 depicts the extension point, by which defining a new modelling tool integration requires providing the name of the integration, and implementing the IIntegration interface to specify how to customise, display and apply the recommendations in the tools. The latter interface declares three methods. First, the method configure() must return false if the integration does not need any extra configuration. If it returns true, then the method customiseIntegration() should be used to configure all the necessary features on the client-side. Finally, the method genIntegration() has two parameters: algorithm, which is the name of the aggregation algorithm to be used, and services, which maps the URL of each registered recommendation service, to the list of services available at that URL.

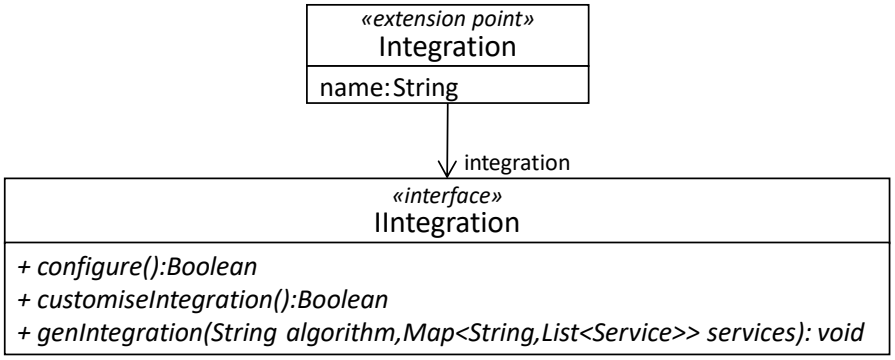


Figure 5.12: Integration extension point.

IronMan currently provides implementations of this extension point for Sirius and tree editors. For this purpose, the generated code makes use of the /recommend endpoint (label 7) of the chosen recommendation service, and the selected aggregation algorithm (if applicable). In the case of Sirius editors, IronMan generates a recommendation layer that enables requesting the recommendations. In the case of tree editors, IronMan generates a menu that is activated when objects of appropriate types are selected in the tree.

5.2.2 IronMan *plugin*

IronMan provides a wizard to adapt RSs to a different modelling language, configure the aggregation of recommendations for the same target (if needed), and integrate the selected RSs into modelling workbenches. Figures 5.13 and 5.14 show the five pages of the wizard. In Figure 5.13, page 1 allows selecting the available RSs from a set of recommendation services previously indexed. New indexed services can be added using the IronMan preference page within the Eclipse IDE. Users can select any combination of RSs, as long as all of them are for the same language. In the figure, the indexed services contain several RSs for UML and Ecore.

IronMan wizard

In page 2 of the wizard, the user can filter the recommended items of each selected RS. For example, in the figure, the page contains recommenders of attributes and operations for classes. The user might be interested in obtaining only attribute recommendations, and this can be selected within this page. Note that it is possible to select several RSs for the same target and items, or for the same target and different items.

In page 3, the user can adapt the RS to a modelling language if the RS targets a different language. For this purpose, the user first selects the meta-model of the modelling language, and then, a tree-table enables defining the mapping between the relevant elements of the RS and the modelling language. In the figure, the user maps elements from UML to Ecore. For instance, in UML, the reference to obtain all attributes is ownedAttribute, but in Ecore is eAllAttributes. Similarly, the composition reference to add Properties to Classes in UML is ownedAttribute as well, but in Ecore is eStructuralFeatures. The identifier of attributes in both UML and Ecore is name. Since the API provides the RS metadata, there is no need to store the RS meta-model locally.

Page 4 in Figure 5.14 is enabled only when the modelling language user selects several RSs providing recommendations for the same target and item, which need to be aggregated. The figure shows the three aggregation methods implemented using the extension point.

In page 5, the user selects the environments – Sirius and/or tree editor – where the RS will be integrated. For Sirius, the modelling language user has to select the viewpoint where the recommendation layer will be inserted. The figure shows the dialog where the modelling language user can select a view. The code generator produces plugin projects with the RS clients, which extend the modelling environments externally, without the need to have available their source code.

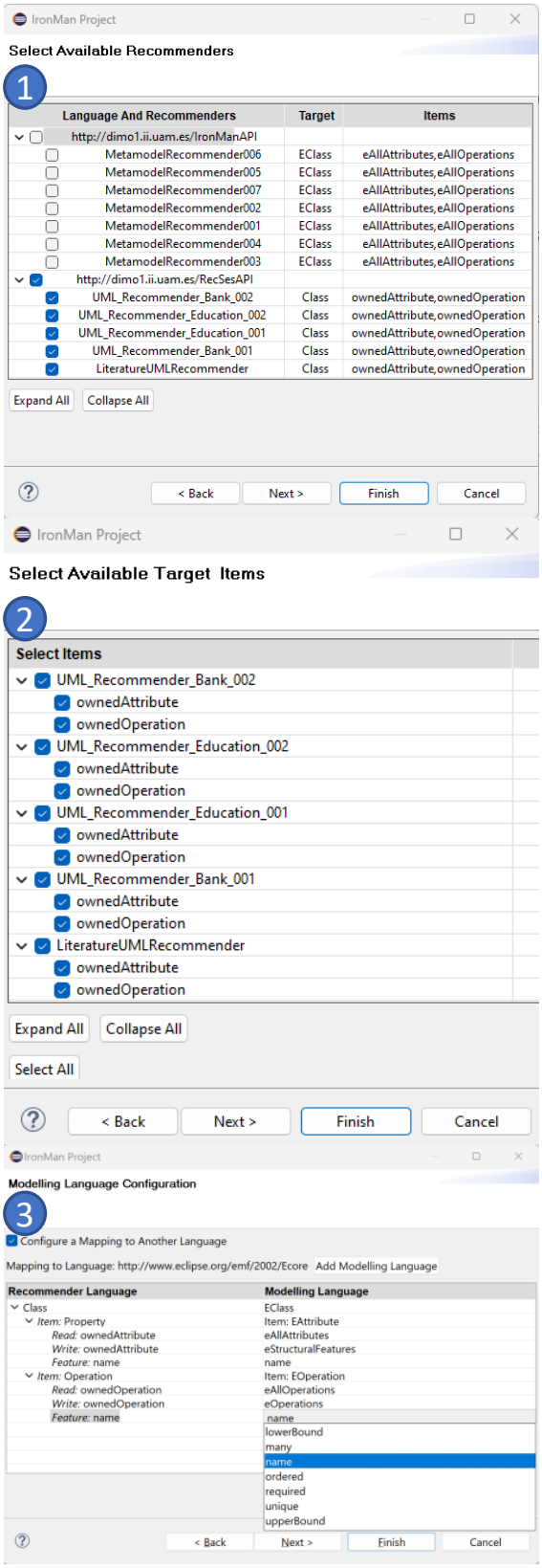


Figure 5.13: Wizard in action: (1) Selecting recommender services, (2) filtering the items to be recommended, (3) mapping the RS to the modelling language (optional step).

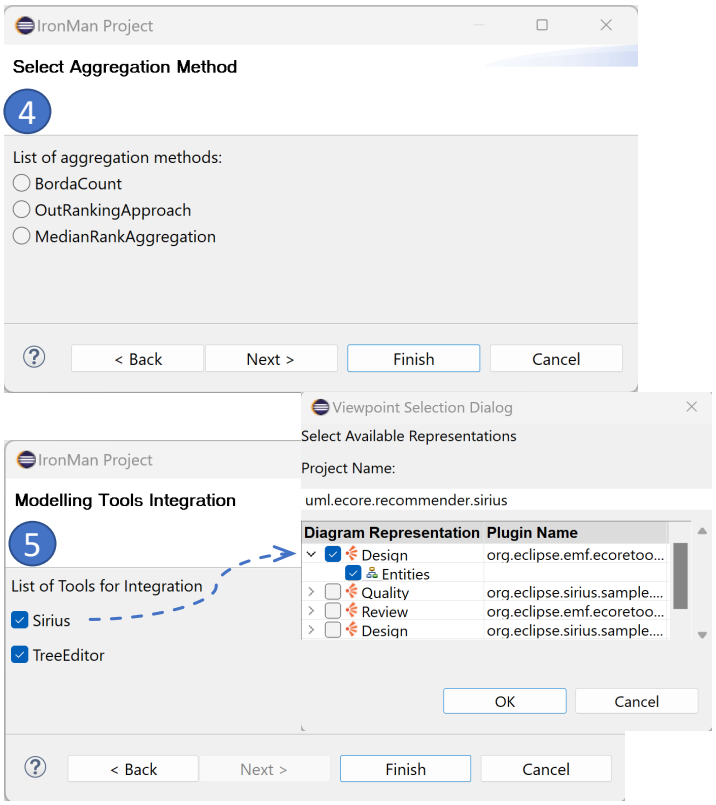


Figure 5.14: Wizard in action: (4) Selecting the aggregation method, (5) selecting the modelling tool where the RS is to be integrated.

5.2.2.1 *Recommendation service*

IronManAPI

The IronMan service indexer is realised as a Java-based REST service implemented using Jersey [26], a framework for building RESTful web services and APIs. It is deployed on Tomcat [27], an open-source web server and Servlet container. There are four core classes responsible for handling requests from clients. *ServiceRegistration* handles registration-related requests, such as service registration, registration updates, deletion, or queries of registered services. *ServiceFeatures* handles requests for deployed and registered services, as well as their metadata. *ServiceRecommend* is responsible for generating recommendations. Finally, *ServiceDiscovery* enables service discovery. The response time for any request is generally less than a second.

5.2.2.2 *Integration with client modelling tools*

Integration with client modelling tools

The IronMan code generator synthesises code that extends externally existing (Sirius and tree) modelling editors. The generated code takes the defined mapping into account. It uses the EMF reflective API to query the relevant features of the target object of the rec-

26 https://eclipse-ee4j.github.io/jersey/
27 http://tomcat.apache.org/

ommendations, and to create objects corresponding to recommended items when the user applies a recommendation.

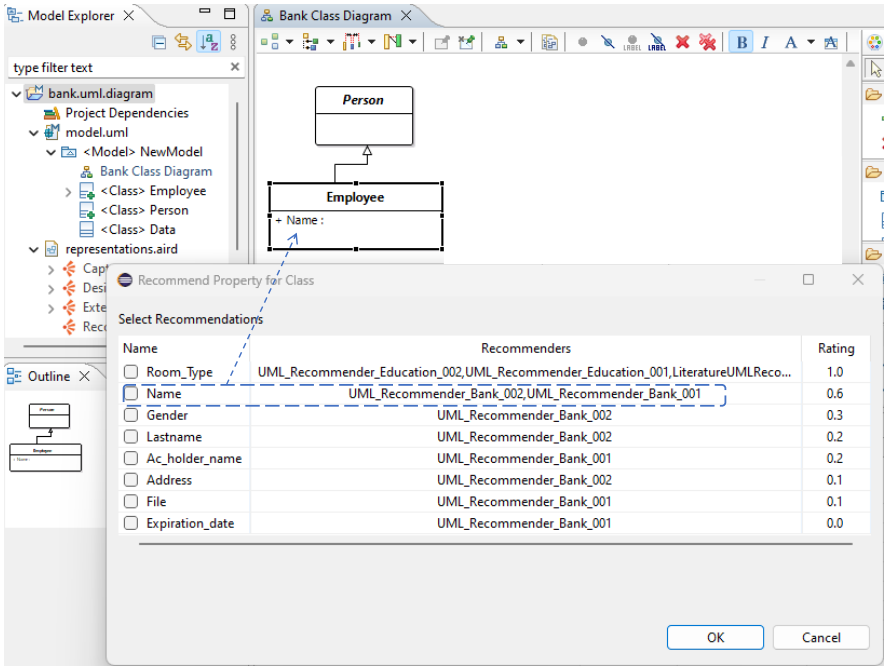


Figure 5.15: Integration of the RS within a Sirius editor.

Figure 5.15 shows a screenshot of the result of the integration of a RS within the Eclipse plugin of Obeo's UML Designer [28]. Once the recommendation layer is active, the RS can be invoked over objects of the target element type (UML classes in this case). The recommender dialog shows a ranked list of recommendations, and the RSs they come from. In this case, two recommenders provide the recommendations, and the list is aggregated using the selected rank aggregation method. When the user selects an item, the corresponding object is created and added to the target.

Figure 5.16 shows the integration of a RS within the standard Ecore tree editor [158]. The RS can be triggered upon selecting an `EClass`. When an `EAttribute` is selected in the recommendation list, the corresponding object is created and assigned to the selected `EClass`.

28 https://marketplace.eclipse.org/content/uml-designer

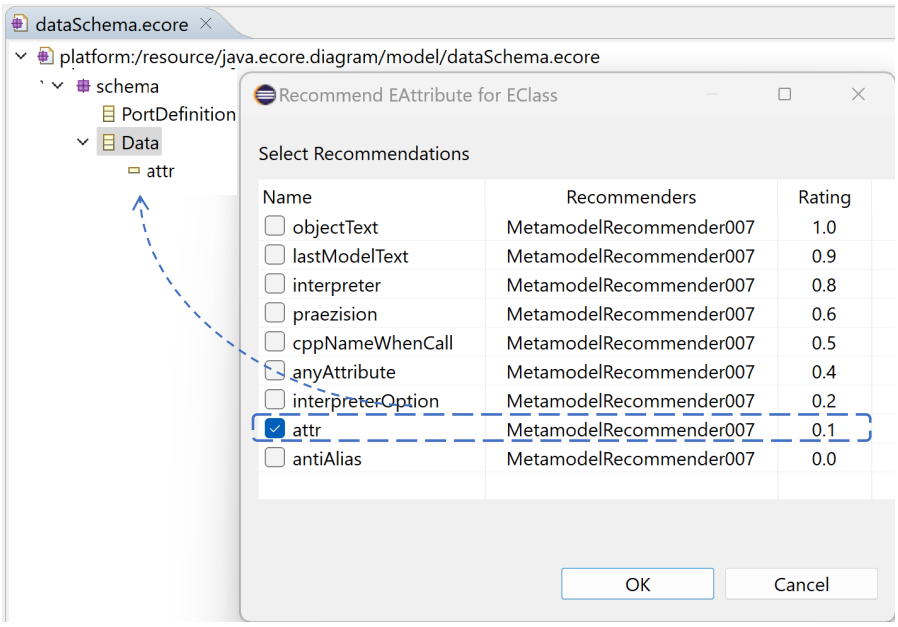


Figure 5.16: Integration of the RS within a tree editor.

5.3 SUMMARY

DROID is a framework that facilitates the construction of RSs for modelling languages. Its main components are the DROID *Configurator*, the DROID *Service*, and the *Clients*. The DROID Configurator allows RS developers to configure, evaluate, and synthesise RSs via a DSL. The synthesised RSs are uploaded to DROIDREST, a generic recommendation service. DROIDREST uses a JSON-based model for client requests and responses, making recommendations accessible to any modelling tool through the DROID *Service*.

DROID has three extension points. The first one permits defining new data sources, currently supporting local directory browsing, the MAR model search engine, and the MDEForge (meta-)model repository. The second extension point permits adding data encodings, and it currently includes RankSys (supporting binary, frequency, and normalized frequency encodings) and MemoRec (using pairs to suggest additional items within a context class). The third extension point facilitates the addition of recommendation methods, currently supporting seven algorithms from MemoRec and RankSys: CACF, ItemPop, CosineCB, UBCF, IBCF, CBUB, and CBIB.

IRONMAN is an Eclipse plugin that automates the integration of RSs within Eclipse modelling editors, aggregates recommendations from multiple RSs, and adapts RSs to languages other than their original targets. The architecture relies on a reference recommendation indexer service and a recommendation service. The flexibility of IRONMAN is enhanced by two extension points. The first one allows adding rank aggregation methods, and currently includes the meth-

ods BC, MRA, and Outranking. The second one allows defining code generators for the integration of the RSs with Eclipse modelling tools, with current support for Sirius and tree editors.

The IronMan wizard simplifies the adaptation, aggregation, and integration of RSs through five steps where users can do: selection (choose RSs from indexed services for the same language); filtering (filter recommended items for each RS); adaptation (adapt RSs to a modelling language by mapping relevant elements between the RS and the chosen language); aggregation (select the aggregation method); and integration (specify the environments, Sirius and/or tree editor, where the RS will be integrated).

Part III

EXPERIMENTS

6

OFFLINE EVALUATION OF DROID RECOMMENDERS

This chapter delves into an offline experiment focused on evaluating the accuracy of the recommendations generated by DROID recommenders. The primary objective of this study was to address the following first research question:

> **RQ1** *How precise, complete, and diverse are the recommendations of* DROID *recommenders?*

Additionally, we aimed to analyse if data preprocessing can impact the accuracy of the recommenders, addressing a second research question:

> **RQ2** *Can data preprocessing improve the recommendations of* DROID *recommenders?*

The chapter begins by detailing the setup of the offline experiment in Section 6.1 and the design of the experiment in Section 6.2. Following this, the results obtained from the experiment are presented in Section 6.3. Subsequently, Section 6.4 provides a discussion of the results. Section 6.5 presents potential threats to the validity of the evaluation. Finally, the chapter concludes with a summary in Section 6.6.

6.1 EXPERIMENT SETUP

To understand how good the recommendations of DROID recommenders are, we used DROID to build several RSs for UML class diagrams in three different domains: *Banking*, *Literature* and *Education*. We collected the models used to train and test the RSs from MAR [70], a structure-based search engine for models and meta-models, which can be queried via input keywords.

Table 6.1 shows the search keywords used to retrieve the models for each domain. In addition, the table shows the following information for each dataset: number of models, number of targets (i.e., classes), number of items (i.e., attributes and operations), and the average number of items per target. The number of models in the datasets was balanced among domains, ranging from 2,297 models (for *Banking*) to 2,771 (for *Education*). We selected a set of keywords per domain to perform the search. The keywords used to search for models for

Table 6.1: Description of the datasets.

Domain	Keywords	Models	Targets	Items	Avg. Items/-Targets
Banking	bank, accounting, finance, economy, investment	2,297	2,346	6,902	3.15
Literature	bibliography, book, author, journal, magazine	2,605	2,272	7,202	2.94
Education	professor, teacher, student, alumn, school	2,771	2,154	6,789	3.17
Total:		**7,673**	**6,772**	**20,893**	

the domain *Bank* were: *bank*, *accounting*, *finance*, *economy* and *investment*; for the domain *Literature*, we used the keywords: *bibliography*, *book*, *author*, *journal* and *magazine*; and for the domain *Education*, we used: *professor*, *teacher*, *student*, *alumn*, and *school*.

The search was executed on each domain individually using the set of keywords corresponding to that domain, and confining our search to UML models. The resulting datasets are available at `https://github.com/Droid-dsl/DroidConfigurator`. All models conform to the UML 2.0 meta-model, and contain class diagrams.

6.2 EXPERIMENT DESIGN

In our offline experiment, multiple RSs were constructed for each domain. All the RSs were designed to recommend attributes and operations for classes. Listing 6.1 provides the specific details of the used configurations. One configuration for each domain was defined, utilising the same setup, but modifying the dataset and the name of the RS to align with the respective domain. In the listing, the configuration for the *"Literature Recommender"* is illustrated as an example. Overall, the configuration defined the following options:

- For data preprocessing (cf. Subsection 4.3.2), the specifications were: *specialCharRemoval* with two options, *true* and *false*; *editDistanceMerging* with values 2, 3 and 4; and both *minRatingsPerItem* and *minRatingsPerTarget* with values 1, 2 and 3. Additionally, an experiment with no data preprocessing technique was executed to address RQ2.

```
Recommender: "LiteratureRecommender"
Metamodel: "http://www.eclipse.org/uml2/5.0.0/UML"
Repository: "/LiteratureRecommender/instances"

Target {
  class Class {
    item "attributes" : ownedAttribute;
    item "methods" : ownedOperation;
  }
}

Identifiers {
  class Class {
    pk feature name;
  }
  class Property {
    pk feature name;
  }
  class Operation {
    pk feature name;
  }
}

PreProcessing {
  specialCharRemoval: true, false;
  editDistanceMerging: 2,3,4;
  minRatingsPerItem: 1,2,3;
  minRatingsPerTarget: 1,2,3;
}

Recommendations {
  Split {
    splitType: CrossValidation;
    nFolds: 10;
    perUser: true;
  }
  Methods {
    ItemPop, CosineCB,
    IBCF("5","10","15","20","25","50","100"),
    UBCF("5","10","15","20","25","50","100"),
    CBIB("5","10","15","20","25","50","100"),
    CBUB("5","10","15","20","25","50","100");
  }
  Evaluation {
    metrics: Precision, Recall, F1, NDCG, ISC, USC, MAP;
    cutoffs: 1,2,3,4,5;
    maxRecommendations: 5;
    relevanceThreshold: 0.5;
  }
}
```

Listing 6.1: Droid configuration to define the RSs for UML class diagrams in the *Literature* domain.

- For data splitting (cf. Subsection 4.3.3), we defined 10-fold cross-validation with a *perUser* technique to split the datasets into training and test sets.
- For training (cf. Subsection 4.3.3), the selected recommendation methods were ItemPop, CosineCB, CBIB, CBUB, IBCF and UBCF. The context-aware recommendation method CACF was excluded from the experiment as it is not applicable to UML models, but just to meta-models. All methods were parameterised with the exception of ItemPop and CosineCB. The rest of the recommendation methods were parameterised with user/item neighbourhood sizes (k) of 5, 10, 15, 20, 25, 50 and 100. As commonly done in the RSs field [78], we consider ItemPop as a baseline to beat, being a non-personalised recommendation approach but capable of providing effective recommendations due to potential popularity biases in the data.
- For evaluating the RSs (cf. Subsection 4.3.4), we applied the ranking quality metrics *precision* (p), *recall* (r), *F1*, *nDCG* and *MAP*; and the coverage and diversity metrics *USC* and *ISC*. Additionally, a relevance threshold of 0.5 was specified, a maximum number of recommendations of 5, and cut-off values from 1 to 5.

6.3 EXPERIMENT RESULTS

We executed the 54 defined preprocessing options, and analysed some metrics of the datasets resulting for each option. We particularly examined the total number of items after applying the Levenshtein distance and removing the items without a minimum number of ratings. Furthermore, we analysed the percentage of targets and items available after each preprocessing option, as well as the percentage of sparsity in the dataset.

After executing the preprocessing option that yielded the dataset with the best metric values, a total of 30 recommenders were obtained. Table 6.2 shows the performance achieved by a subset of them in each domain/dataset (*Banking*, *Education* and *Literature*), as well as the average results across the three domains. The rows correspond to the recommendation methods and domains/datasets, and the columns to the performance metrics. For the sake of brevity, the table just shows three representative recommendation methods: ItemPop, CosineCB (a representative of content-based approaches), and CBUB, the best performing collaborative filtering method in this experiment). For CBUB, the table presents the results for neighbourhood size k=5, since it was the best performant. ItemPop and CosineCB do not have parameters.

Table 6.2: Offline experiment: Performance of the recommendation methods.

Method	Domain	p	r	F1	MAP	nDCG	USC	ISC
ItemPop	All domains	0.041	0.201	0.069	0.138	0.155	1.000	0.016
CosineCB		0.043	0.216	0.072	0.212	0.213	1.000	0.025
CBUB		0.370	0.515	0.431	0.482	0.492	0.436	0.032
ItemPop	Banking	0.078	0.383	0.129	0.272	0.301	1.000	0.023
CosineCB		0.068	0.339	0.113	0.335	0.336	1.000	0.036
CBUB		0.604	0.688	0.643	0.669	0.677	0.436	0.042
ItemPop	Education	0.019	0.084	0.031	0.066	0.073	1.000	0.008
CosineCB		0.026	0.132	0.044	0.129	0.130	1.000	0.016
CBUB		0.175	0.384	0.241	0.326	0.342	0.423	0.020
ItemPop	Literature	0.028	0.135	0.046	0.076	0.090	1.000	0.016
CosineCB		0.036	0.178	0.059	0.171	0.173	1.000	0.022
CBUB		0.331	0.472	0.389	0.449	0.456	0.451	0.034

We can observe that CBUB was the best-performing recommendation method across all domains. By contrast, ItemPop performed the worst across the domains except on *Banking*, where it performed slightly better than CosineCB.

Considering *F1* – the harmonic mean between *precision* and *recall* – the best recommendations were generated for the *Banking* domain, with an *F1* value of 0.643 for CBUB. The other two domains show a significant, but more undersized performance. Furthermore, the performance results were quite positive over the whole dataset (i.e., considering the three addressed domains together), with a maximum *F1* value of 0.431 for CBUB.

As for the other metrics, the *Banking* domain exhibited the highest values of *MAP* and *nDCG*, with values of 0.669 and 0.677, respectively, for CBUB. By contrast, ItemPop and CosineCB outperformed CBUB in terms of *USC* across all domains, reaching values of 1.0 for the metric. Finally, *ISC* was slighty higher for CBUB than for the other methods, with a value of 0.042 in the *Banking* domain.

Table 6.3 presents the precision of the same recommendation methods on each domain at cut-off k, $p@k$, for k from 1 to 5. We observe that the smaller the value k, the higher the precision. Likewise, CBUB outperformed ItemPop and CosineCB across all domains.

To investigate the impact of data preprocessing on the performance of our RSs, we constructed RSs using the same three datasets as before, but without applying data preprocessing to them. Table 6.4 outlines the performance achieved by the same recommendation methods analysed in Table 6.2, both in each domain/dataset and in average across the three domains. Following the format of earlier ta-

Table 6.3: Offline experiment: Precision@k of the recommendation methods.

Method	Domain	p@1	p@2	p@3	p@4	p@5	Avg.
ItemPop	All domains	0.102	0.077	0.061	0.048	0.041	0.066
CosineCB		0.210	0.106	0.071	0.054	0.043	0.097
CBUB		0.464	0.250	0.171	0.130	0.105	0.224
ItemPop	Banking	0.194	0.154	0.127	0.095	0.078	0.129
CosineCB		0.334	0.167	0.112	0.084	0.068	0.153
CBUB		0.663	0.347	0.235	0.176	0.141	0.312
ItemPop	Education	0.058	0.044	0.029	0.022	0.019	0.034
CosineCB		0.126	0.066	0.044	0.033	0.026	0.059
CBUB		0.293	0.176	0.122	0.098	0.078	0.154
ItemPop	Literature	0.053	0.033	0.028	0.025	0.028	0.033
CosineCB		0.169	0.085	0.056	0.044	0.036	0.078
CBUB		0.437	0.229	0.155	0.117	0.095	0.207

bles, the rows correspond to recommendation methods and domains/datasets, while columns represent the performance metrics. For CBUB, the table presents the results for neighbourhood size k=5.

Table 6.4: Offline experiment: Performance of the recommendation methods without data preprocessing.

Method	**Domain**	**p**	**r**	**F1**	**MAP**	**nDCG**	**USC**	**ISC**
ItemPop	All Domains	0.025	0.095	0.040	0.061	0.073	1.000	0.003
CosineCB		0.006	0.028	0.009	0.027	0.028	1.000	0.003
CBUB		0.112	0.254	0.155	0.201	0.219	0.679	0.031
ItemPop	Banking	0.021	0.083	0.034	0.050	0.061	1.000	0.003
CosineCB		0.005	0.027	0.009	0.026	0.026	1.000	0.003
CBUB		0.102	0.258	0.146	0.200	0.218	0.677	0.032
ItemPop	Education	0.030	0.111	0.047	0.075	0.088	1.000	0.003
CosineCB		0.006	0.029	0.010	0.029	0.029	1.000	0.003
CBUB		0.130	0.256	0.172	0.211	0.228	0.668	0.028
ItemPop	Literature	0.024	0.089	0.038	0.058	0.069	1.000	0.003
CosineCB		0.005	0.027	0.009	0.027	0.027	1.000	0.003
CBUB		0.104	0.249	0.146	0.191	0.210	0.693	0.034

We observe that, for all metrics, the recommendation methods showed better performance when data preprocessing was applied (Table 6.2) compared to omitting data preprocessing (Table 6.4). The only exception is a decrease of the *USC* values. For the *Banking* domain, the *F1* value increased from 0.146 to 0.643 when using data preprocessing, implying a 340% increase in performance. CBUB was the best-

performing recommendation method across all domains, regardless of whether or not data preprocessing was used. In contrast to the experiment with data preprocessing, CosineCB without data preprocessing performed the worst in all domains.

Additionally, Table 6.5 presents the precision of the same recommendation methods on each domain at cut-off k, $p@k$, for k from 1 to 5, for the experiment without data preprocessing. Similarly to the results reported in Table 6.3, the smaller the value k, the higher the precision. Likewise, CBUB outperformed ItemPop and CosineCB across all domains.

Table 6.5: Offline experiment: Precision@k of the recommendation methods without data preprocessing.

Method	Domain	p@1	p@2	p@3	p@4	p@5	Avg
ItemPop	All Domains	0.056	0.041	0.034	0.029	0.025	0.037
CosineCB		0.027	0.014	0.009	0.007	0.006	0.013
CBUB		0.185	0.117	0.088	0.071	0.060	0.104
ItemPop	Banking	0.038	0.033	0.029	0.024	0.021	0.029
CosineCB		0.026	0.013	0.009	0.007	0.005	0.012
CBUB		0.176	0.112	0.085	0.069	0.058	0.100
ItemPop	Education	0.072	0.051	0.040	0.034	0.030	0.045
CosineCB		0.029	0.015	0.010	0.007	0.006	0.013
CBUB		0.207	0.125	0.094	0.075	0.063	0.113
ItemPop	Literature	0.057	0.039	0.033	0.028	0.024	0.036
CosineCB		0.027	0.013	0.009	0.007	0.005	0.012
CBUB		0.171	0.113	0.086	0.070	0.059	0.100

6.4 DISCUSSION

To answer RQ1 (*How precise, complete and diverse are the recommendations of* Droid *recommenders?*), we analyse the performance of the recommendation methods looking at their *F1* values (cf. Table 6.2). *F1* is a commonly used metric in the information retrieval and RSs fields. It is the harmonic mean of *precision* and *recall*, and thus provides a compromise between the precision of the recommendations (the proportion of relevant items among the retrieved items) and their recall (the proportion of relevant items that were retrieved), ensuring that the evaluation considers both the relevance and coverage of the recommendations. The highest *F1* value was 0.643 for the *Banking* domain, 0.241 for *Education*, 0.389 for *Literature*, and 0.431 for all domains together. In all cases, CBUB achieved the highest *F1* value.

These results show that the RSs built with Droid can provide sensible recommendations for every class on the three domains. Addi-

tionally, we observe that the performance of the RSs varies across domains. Several factors can play a role here depending on the quality of the dataset, like the average number of preferences per target/item, or the rating sparsity, which is the proportion of existing target-item relations.

To answer RQ2 (*Can data preprocessing improve the recommendations of* Droid *recommenders?*), we analyse the performance of the recommendation methods by examining their metric values, particularly *F1* values, and comparing the results in Tables 6.2 and 6.4.

In the *Banking* domain, the *F1* value of the best performing method (CBUB) increased from 0.146 to 0.643 (340% improvement). For the *Education* domain, *F1* increased from 0.172 to 0.241 (a 40.12% improvement). In the *Literature* domain, it increased from 0.146 to 0.389 (a 166.44% improvement). Thus, we answer the question positively: preprocessing enhances the precision-based metrics while maintaining a balance with diversity/coverage, with the exception of *USC*, where values either remained constant or decreased for one of the cases.

In the first part of the offline experiment, presented in Table 6.2, the application of preprocessing techniques significantly improved the performance of all methods. These techniques facilitated the removal of items that did not meet a minimum frequency in targets and also enabled the elimination of targets with insufficient representation in the dataset. Moreover, employing methods like Levenshtein distance contributed to further enhancement in the overall performance. Additionally, for hybrid methods, applying data preprocessing to a sufficiently large dataset allowed establishing more valuable content-based user similarities within the collaborative filtering heuristic of the method.

In particular, the inclusion of a data preprocessing phase, which was absent in Table 6.4, allows creating RSs with an order of magnitude higher performance. This way, while preprocessing may lead to higher values of precision-based metrics (precision, recall, F1), it could entail a decrease in the coverage metric – like USC – which achieved higher values in the first experiment shown in Table 6.2.

6.5 THREATS TO VALIDITY

Next, we examine the threats to the validity of the offline evaluation.

External validity refers to the degree of generalisability of the results of an experiment. We used three different domains, and for each one of them, we constructed a specific dataset by searching models containing representative keywords of the domain (cf. Table 6.1). To increase even further the generality of our results, we could expand the selection of keywords, and incorporate more domains. Moreover, our experiment focused on a particular recommendation task, namely, the

recommendation of class attributes and operations. Hence, we cannot claim that our conclusions apply to other different modelling tasks.

Internal validity indicates to which extent a causal relationship exists between the conducted experiment and the presented conclusions. We attempted to avoid any bias on the data by using a third-party search engine to collect the diagrams of our datasets, and by using a set of model search keywords instead of hand-picking the models.

Construct validity is the extent to which an experiment accurately measures the concept it was intended to evaluate. To avoid an inadequate definition of the measured concepts, our offline evaluation established metrics in the RSs community.

Conclusion validity is the degree in which the conclusions are founded on an adequate analysis of the data. The offline experiment employed widely-used software (RankSys and RiVal) to measure the performance of the RSs, thus preventing any spurious measurements.

6.6 SUMMARY

We conducted an offline experiment to assess the accuracy of the recommendations generated by DROID recommenders. The research questions of the experiment focused on the precision, completeness, and diversity of the recommendations provided by DROID recommenders, and on the impact of data preprocessing on the accuracy of recommendations.

For the experiment, 54 preprocessing options were executed and analysed to assess their impact on the dataset. The preprocessing option leading to the best metric values resulted in the creation of 30 candidate RSs. The performance of these recommenders was evaluated across three domains (*Banking*, *Literature*, and *Education*) using the metrics *precision*, *recall*, *F1*, *nDCG*, *MAP*, *USC*, and *ISC*.

To assess the impact of data preprocessing, RSs were constructed without applying preprocessing to the datasets. The results indicated that, in general, recommendation methods performed better when data preprocessing was applied.

The discussion addresses two key research questions regarding DROID recommenders. First, the analysis of *F1* values reveals that CBUB consistently achieves the highest precision across the three domains, indicating sensible recommendations for each class. Variability in performance across domains is attributed to factors like dataset quality and rating sparsity. Second, the impact of data preprocessing on recommendations is examined by comparing results with and without preprocessing. For the *Banking* domain, CBUB's *F1* value significantly improved by 340% with preprocessing. The positive outcomes suggest that data preprocessing enhances precision-based metrics while maintaining a balance with diversity/coverage. However, it may lead to a trade-off, as evidenced by the decrease in *USC* values.

The absence of preprocessing results in RSs with lower performance, emphasizing its crucial role in achieving higher precision-based metric values and overall effectiveness in DROID recommenders.

7

USER STUDY EVALUATION OF DROID RECOMMENDERS

This chapter reports on a user study aimed to evaluate how people perceive the recommendations issued by Droid recommenders. Since users of modelling languages may have different perceptions of the quality and usefulness of the issued recommendations, the primary objective of the user study was to address the following research question:

> **RQ3:** *How do users perceive the recommendations of* Droid *recommenders?*

Additionally, we aimed to analyse the degree to which the outcomes of the offline evaluation reported in Chapter 6 might vary from those obtained in a user study context. We thus addressed the following research question:

> **RQ4:** *How do the offline experiment results compare to the ones of the user study?*

The chapter begins by describing the setup of the experiment in Section 7.1. Afterwards, in Section 7.2, it presents the design of the study, including details of the involved participants and the considered evaluation metrics. Following this, in Section 7.3, it reports the achieved results per evaluation metric. Subsequently, in Section 7.4, it provides a discussion of the results. Later on, in Section 7.5, the chapter gathers potential threats to validity and, finally, in Section 7.6, it concludes with a summary.

7.1 EXPERIMENT SETUP

The user study involves human participants performing one task, consisting in the assessment of the recommendations generated by the three recommendation methods discussed in the offline experiment (cf. Chapter 6): ItemPop, CosineCB, and CBUB. Analogously to the offline experiment, the evaluated recommendations belonged to three domains: *Banking*, *Education* and *Literature*.

We conducted the study remotely and asynchronously. Participants received an email invitation that included the URL of a website where the experiment took place and a unique user identifier. To encourage

maximum participation, we granted seven days to complete the experiment. After logging into the website with their identifier, the participants had to address either 3 or 6 evaluation cases [29]. Each case presented a class diagram composed of a target class and its modelling context, together with a list of recommended items (a mix of attributes and operations) for the target class. Then, for each recommended item, participants had to state whether they perceived the recommendation as:

- *Correct*: The item is suitable for the target class.
- *Obvious*: The item could have easily been proposed by the participant.
- *Redundant*: The item exists or is similar to an existing one in the diagram.
- *Contextualised*: The item belongs to the domain of the diagram.
- *Generalisable*: The item is also applicable to other classes of the diagram.

As an example, Figure 7.1 shows one of the cases from the experiment, as presented to participants. The web page displays: (1) a class diagram with the target class (tagged as «Target») and its context, (2) the recommended attributes and operations for the target class, (3) the five criteria to be evaluated by the participant, (4) the participant's level of confidence in her assessments, and (5) a textbox to specify missed items for the target class not found among the recommendations.

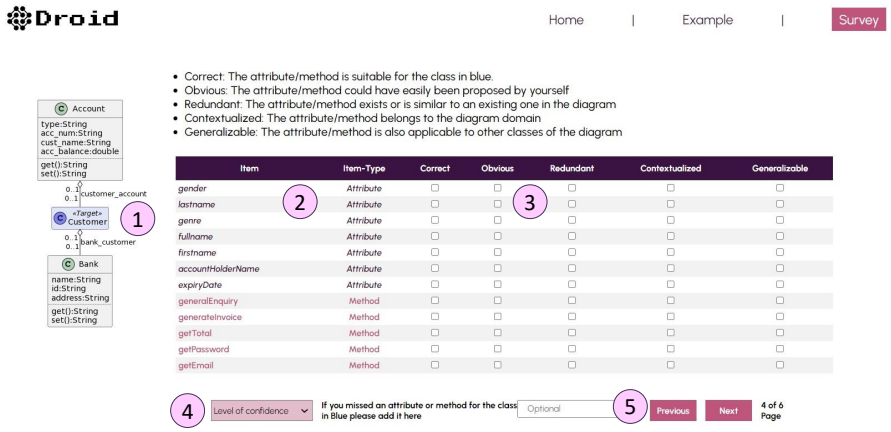


Figure 7.1: Example of a case evaluated in the user study.

29 To ensure the evaluation of 45 cases, the last 5 participants only evaluated 3 cases.

7.2 EXPERIMENT DESIGN

We assigned the cases to each participant randomly from the three domains, ensuring that each case was evaluated by exactly five participants. The recommendation lists for each target class included the five top items suggested by each recommendation method (ItemPop, CosineCB and CBUB). This means that each target class received a maximum of 3x5=15 recommendations, but there could be fewer, since we eliminated duplicates (i.e., items recommended by more than one method). We selected this size to make the evaluation less burdensome for the participants by receiving a reasonable, but not overwhelming number of options. Altogether, participants evaluated 45 cases, each one of them containing 15 recommendations, which resulted in 3,375 recommendation evaluations. The recommended items were listed in random order, so participants could not know which method each recommended item came from; they even did not know that the recommendations were automatically generated.

7.2.1 *Assessment metrics*

To evaluate the participants' perception of the recommendations, we considered their manual assessments (i.e., whether the recommendations were correct, obvious, redundant, contextualised or generalisable) to compute the following metrics:

- *Precision@k* (for k = 1,...,5), which measures the likelihood that a suggested item is relevant.
- *Serendipity*, which measures how unexpected the correct recommendations are.
- *Redundancy*, which measures to what extent correct recommendations are unnecessary for the user.
- *Contextualisation*, which measures whether a correct recommendation is related to the target's context (in the study, the class diagram of the target class).
- *Generalisation*, which measures whether a correct recommendation is extensible (valuable) to other targets.

Precision was computed out of the *correct* assessments of participants. It was measured as the percentage of *positively agreed* evaluations about correctness out of the total number of target-item pairs of the study (3,375). As commonly done in user studies with inter-rater agreement [60], we considered a target-item pair as *positively agreed* if it was marked correct by at least three participants out of the five who evaluated the pair. The computation of the other metrics is analogous.

Serendipity considers the negatively agreed *obvious* assessments, i.e., the target-item pairs that the participants marked as not obvious. Similarly, redundancy, contextualisation and generalisation consider the *positively agreed* redundant, contextualised and generalisable assessments, respectively. We did not measure recall, since participants did not have the complete set of correct items for a given target (as stated by human subjects).

It must be stressed that, to the best of our knowledge, our study is the first experiment where the metrics serendipity, redundancy, contextualisation and generalisation have been used to evaluate the model completion recommendation task. They, however, have been proposed or used as valuable and complementary metrics for evaluating recommender systems [130, 155].

7.2.2 *Participants*

We recruited the participants by email, inviting a potential set of candidates from the modelling and recommendation areas. Overall, 40 people completed the user study within the given seven day limit.

Prior to the experiment, the participants filled out a demographic questionnaire to collect some statistical data. Overall, 80% of the participants were male and 20% female. The majority were between 25 and 34 years old (55%), with the rest between 35 and 44 (27.5%), 45 and 54 (12.5%), and 18 and 24 (5%). Half of the participants were PhD students (50%), while the rest were PhD holders (37.5%), MSc holders (7.5%), and MSc students (5%). Participants were mostly academics (65%), but also industry employees (22.5%), researchers (7.5%) and students (5%). Note that, in this classification, researchers do not work at a university but in a research center, and industry employees work in a company with no research duty. Most participants (90%) had a high level of English, and all had studied computer-related subjects. A 95% of the participants declared a high level of knowledge on object-oriented programming, and 87.5% on class diagrams. Participants had between 1 to 27 years of experience in software development (10 years in average).

7.3 EXPERIMENT RESULTS

The following five subsections discuss the results of the user study in terms of the assessment metrics introduced in Subsection 7.2.1. First, in Subsection 7.3.1, the precision results are presented. Precision is the only metric that is common to both the offline experiment (cf. Chapter 6) and this user study. These results will allow for a comparison that will be depicted in the discussion (cf. Section 7.4). Afterwards, the serendipity results are detailed in Subsection 7.3.2. Following that, the redundancy and contextualisation results are presented in Subsec-

tion 7.3.3 and Subsection 7.3.4, respectively. Finally, the generalisation results are detailed in Subsection 7.3.5.

7.3.1 *Precision results*

Precision measures how accurate the recommendations are, being an indicator of the relevance of the recommendations for the users. In our study, we computed precision as the percentage of target-item pairs that the participants marked as *correct*. Table 7.1 shows the precision values achieved by the recommendation methods on each domain.

Columns three to seven show the precision at k, *p@k*, which is the precision of the top k recommended items, with $k = 1, ..., 5$. The recommendation method with the highest precision for the complete dataset (i.e., considering all domains jointly) was CBUB, with an average precision of 0.373. This is in accordance with the offline experiment, where CBUB was the best performing method. CBUB also achieved the best precision in the *Banking* (0.493) and *Education* (0.373) domains, but not in the *Literature* domain, where CosineCB had a higher precision (0.453).

Table 7.1: User study: Precision@k by domain.

Method	Domain	p@1	p@2	p@3	p@4	p@5	Avg.
ItemPop	All domains	0.267	0.356	0.267	0.356	0.289	0.307
CosineCB		0.244	0.133	0.467	0.311	0.244	0.280
CBUB		0.444	0.356	0.333	0.400	0.333	0.373
ItemPop	Banking	0.467	0.667	0.333	0.400	0.267	0.427
CosineCB		0.467	0.200	0.533	0.267	0.200	0.333
CBUB		0.600	0.667	0.400	0.400	0.400	0.493
ItemPop	Education	0.067	0.267	0.333	0.267	0.200	0.227
CosineCB		0.067	0.000	0.067	0.133	0.000	0.053
CBUB		0.467	0.133	0.333	0.533	0.400	0.373
ItemPop	Literature	0.267	0.133	0.133	0.400	0.400	0.267
CosineCB		0.200	0.200	0.800	0.533	0.533	0.453
CBUB		0.267	0.267	0.267	0.267	0.200	0.253

More in detail, Table 7.2 shows the *p@k* values (for all domains) depending on the confidence level that the participants indicated in the study for each case evaluated. The table only considers those assessments where the participants felt somewhat/fairly/completely confident. In this case, CBUB outperformed CosineCB and ItemPop in all domains. In particular, focusing only on the cases of *completely confident* participants, CBUB achieved an average precision value of 0.543.

Table 7.2: User study: Precision@k by confidence level for all domains.

Method	Confidence	#Part.	#Evals	p@1	p@2	p@3	p@4	p@5	Avg.
ItemPop	Completely confident	9	225	0.357	0.429	0.286	0.714	0.429	0.443
CosineCB				0.429	0.500	0.643	0.500	0.500	0.514
CBUB				0.429	0.714	0.571	0.429	0.571	0.543
ItemPop	Completely confident, Fairly confident	34	1860	0.341	0.409	0.432	0.500	0.455	0.427
CosineCB				0.386	0.386	0.591	0.477	0.409	0.450
CBUB				0.545	0.455	0.500	0.455	0.568	0.505
ItemPop	Completely confident, Fairly confident, Somewhat confident	40	2835	0.267	0.444	0.311	0.422	0.311	0.351
CosineCB				0.311	0.289	0.533	0.378	0.289	0.360
CBUB				0.467	0.400	0.356	0.422	0.422	0.413

Finally, Table 7.3 restricts the analysis of precision further by considering only *completely confident* assessments in two scenarios. The first scenario only considers those cases where the target class was contextualised, i.e., the class diagram contained other classes beyond the target class (Table 7.3a). In the second scenario, in addition to be contextualised, the target class had at least one attribute or operation (Table 7.3b). The precision of CosineCB and CBUB in these two scenarios increased, achieving *p@k* values of 0.582 and 0.600. However, as the table shows, these values came from a small number of participants and evaluations, so their statistical support is small. In Subsection 7.3.4, we will use these scenarios to analyse contextualisation.

Table 7.3: User study: Precision@k of *completely confident* assessments in two scenarios: (a) target class with context; (b) target class with context and items (attributes or operations).

(a) Precision@k contextualised

Method	Confidence	#Part.	#Evals	p@1	p@2	p@3	p@4	p@5	Avg.
ItemPop	Completely confident	8	180	0.364	0.364	0.273	0.636	0.455	0.418
CosineCB				0.455	0.545	0.636	0.636	0.636	0.582
CBUB				0.455	0.636	0.455	0.455	0.636	0.527

(b) Precision@k with contextualised and with items

Method	Confidence	#Part.	#Evals	p@1	p@2	p@3	p@4	p@5	Avg.
ItemPop	Completely confident	6	150	0.444	0.333	0.333	0.667	0.556	0.467
CosineCB				0.444	0.556	0.667	0.667	0.667	0.600
CBUB				0.444	0.667	0.556	0.444	0.667	0.556

7.3.2 *Serendipity results*

Serendipity is the ability to suggest items that go beyond popular or trending ones [133]. It measures the unexpectedness of the recommendations, which the user would not otherwise think of, even though they are relevant to the task at hand. In this sense, serendipitous rec-

ommendations are not necessarily novel (fresh) items recently added into the system. By contrast, they are items that, in addition to being unknown to the user, generate certain surprise the first time they are presented.

In the particular recommendation task that we addressed – class model completion – serendipitous recommendations can be very valuable as they consist of non-evident attributes and operations appropriate for the target class being modelled.

We measured serendipity as the percentage of target-item pairs that the participants marked as *non-obvious*. That is, recommendations marked as obvious are not considered serendipitous, and vice versa.

Table 7.4 shows the average serendipity values of the recommendation methods per domain. In general, participants perceived the recommendations as non-obvious, as serendipity values are close to 1. As expected, ItemPop generated slightly more serendipitous recommendations than CosineCB and CBUB when considering the dataset of all domains together, as some recommendations were generic or did not clearly belong to the target domain. If we look at each domain separately, the *Education* domain had the highest serendipity values, and the *Banking* domain the lowest ones. This may be due to the closeness of the *Education* domain to participants, who were mostly academics. We found no significant difference between the serendipity values of the recommendation methods for a same domain.

Table 7.4: User study: Serendipity@k by domain.

Method	Domain	#Evals	s@1	s@2	s@3	s@4	s@5	Avg.
ItemPop	All domains	3,375	0.822	0.822	0.956	0.844	0.822	0.853
CosineCB			0.844	0.889	0.600	0.822	0.844	0.800
CBUB			0.756	0.800	0.889	0.800	0.867	0.822
ItemPop	Banking	1,125	0.533	0.533	0.867	0.867	0.867	0.733
CosineCB			0.667	0.867	0.533	0.800	1.000	0.773
CBUB			0.467	0.533	0.733	0.867	0.733	0.667
ItemPop	Education	1,125	1.000	1.000	1.000	0.933	0.867	0.960
CosineCB			0.933	1.000	1.000	1.000	1.000	0.987
CBUB			0.867	0.867	0.933	0.667	0.933	0.853
ItemPop	Literature	1,125	0.933	0.933	1.000	0.733	0.733	0.867
CosineCB			0.933	0.800	0.267	0.667	0.533	0.640
CBUB			0.933	1.000	1.000	0.867	0.933	0.947

Finally, we computed the Pearson correlation scores between all pairs of metrics considered in the study (cf. Subsection 7.2.1), and in particular, the correlation between precision and serendipity is 0.545. This score is moderate but relatively high with respect to other pairs of metrics, for which we did not find any significant correlation. This can be considered as a positive result since it gives insights that the

generated recommendations tended to be both accurate and serendipitous. Note that, as reported in Section 6.1, in our dataset, the average number of items per user, i.e., attributes/methods per class in the UML diagrams, was 3.15, 2.94 and 3.17 for the *Banking*, *Literature* and *Education* domains, respectively. Hence, it is likely that the correctly recommended test items are popular attributes/methods for the classes at hand.

7.3.3 *Redundancy results*

We propose using the redundancy metric to assess to what extent the recommended items (attributes and operations) are unnecessary, given the current items of the target (class). It is related to diversity, which considers the absence of duplicated or very similar items within a recommendation list. In general, redundant recommendations should be avoided. Specifically for class modelling, designers should not get suggestions of attributes and operations that exist or are too similar to those in a class.

Redundancy is the percentage of recommendations that the participants perceived as *redundant*. Table 7.5 shows the redundancy values of the recommendation methods by domain. Low values (close to zero) indicate that the recommendations were sufficiently distinct from the attributes and operations of the target class, being consequently useful. We observe no significant differences between the redundancy values of the different methods and domains, except for the *Banking* domain. Redundancy was slightly higher in this domain, meaning that, compared to the other domains, a higher percentage of the recommended items were alike to existing items. This is consistent with the serendipity values that *Banking* obtained (cf. Subsection 7.3.2), as redundant values reflect less serendipity.

Table 7.5: User study: Redundancy@k by domain.

Method	Domain	#Evals	r@1	r@2	r@3	r@4	r@5	Avg.
ItemPop	All domains	3,375	0.044	0.156	0.000	0.022	0.022	0.049
CosineCB			0.000	0.022	0.089	0.111	0.022	0.049
CBUB			0.044	0.089	0.067	0.067	0.022	0.058
ItemPop	Banking	1,125	0.133	0.467	0.000	0.000	0.000	0.120
CosineCB			0.000	0.000	0.133	0.267	0.000	0.080
CBUB			0.133	0.267	0.200	0.133	0.000	0.147
ItemPop	Education	1,125	0.000	0.000	0.000	0.000	0.000	0.000
CosineCB			0.000	0.000	0.000	0.000	0.000	0.000
CBUB			0.000	0.000	0.000	0.067	0.067	0.027
ItemPop	Literature	1,125	0.000	0.000	0.000	0.067	0.067	0.027
CosineCB			0.000	0.067	0.133	0.067	0.067	0.067
CBUB			0.000	0.000	0.000	0.000	0.000	0.000

This result could be due to several factors: the input dataset itself, in which classes have been designed with "redundant" attributes and operations; the frequency distribution of items in the *Banking* domain follows a more pronounced heavy tail than the other domains, and thus there is a greater bias towards recommending popular (and therefore redundant) items; the item neighbourhoods in collaborative filtering are smaller, i.e., there are fewer items per target. An exhaustive study should be done to provide a well-argued explanation. Nonetheless, according to the values reported in Table 7.5, it seems that the last two aspects may apply; the ItemPop and CBUB methods show higher redundancy than CosineCB.

7.3.4 *Contextualisation results*

Contextualisation is a metric we propose to analyse the relation of the recommended items to the target's context. In the class completion task, the context of a target class refers to the set of other classes (together with their attributes and operations) that surround the target class. In our study, we measure contextualisation as the percentage of recommendations that the participants marked as *contextualised*.

Table 7.6 shows the average and contextualisation @k values of each recommendation method and domain. As expected, CBUB generated the most contextualised recommendations since it exploits content-based information from related classes in a collaborative filtering fashion. By contrast, CosineCB was the worst-performing method in terms of contextualisation since it only exploits information of the target class to generate recommendations.

More in detail, Table 7.6a reports the results when considering all cases evaluated by the participants (3,375), where CBUB achieved an average contextualisation value of 0.280. Tables 7.6b and 7.6c show contextualisation results when filtering out cases with little context information. In particular, discarding diagrams with only the target class (Table 7.6b), and discarding diagrams with only the target class or where the target class had just one attribute/operation (Table 7.6c). In these scenarios, the contextualisation values increased to 0.308 and 0.320.

For all the above scenarios, as with previous metrics, if we limit the analysis to those assessments with which participants felt completely confident, the contextualisation values of all methods increased, being 0.473 the maximum one, achieved by CBUB. However, when doing so, the computation of such values came from a significantly smaller number of participants and evaluations.

Table 7.6: User study: Contextualisation@k by confidence level in three scenarios: (a) all target classes; (b) target classes with context; (c) target classes with context and items (attributes or operations).

(a) Contextualisation@k by confidence level

Method	Confidence	#Part.	#Evals	c@1	c@2	c@3	c@4	c@5	Avg.
ItemPop	All	40	3,375	0.289	0.222	0.222	0.200	0.267	0.240
CosineCB				0.156	0.156	0.200	0.244	0.222	0.196
CBUB				0.333	0.200	0.289	0.311	0.267	0.280
ItemPop	Completely confident	9	225	0.429	0.214	0.143	0.357	0.286	0.286
CosineCB				0.143	0.286	0.214	0.214	0.214	0.214
CBUB				0.286	0.357	0.429	0.357	0.500	0.386

(b) Contextualisation@k by confidence level (classes with context)

Method	Confidence	#Part.	#Evals	c@1	c@2	c@3	c@4	c@5	Avg.
ItemPop	All	40	2,775	0.324	0.243	0.243	0.243	0.297	0.270
CosineCB				0.162	0.162	0.216	0.297	0.243	0.216
CBUB				0.378	0.189	0.324	0.324	0.324	0.308
ItemPop	Completely confident	8	180	0.545	0.182	0.182	0.364	0.273	0.309
CosineCB				0.182	0.364	0.273	0.273	0.273	0.273
CBUB				0.364	0.455	0.455	0.455	0.636	0.473

(c) Contextualisation@k by confidence level (classes with context and items)

Method	Confidence	#Part.	#Evals	c@1	c@2	c@3	c@4	c@5	Avg.
ItemPop	All	40	2,250	0.300	0.233	0.267	0.200	0.267	0.253
CosineCB				0.133	0.133	0.167	0.200	0.233	0.173
CBUB				0.367	0.200	0.367	0.333	0.333	0.320
ItemPop	Completely confident	6	150	0.444	0.111	0.111	0.222	0.222	0.222
CosineCB				0.111	0.333	0.222	0.222	0.111	0.200
CBUB				0.333	0.444	0.444	0.333	0.556	0.422

7.3.5 *Generalisation results*

In the studied class modelling task, generalisation is a metric we propose to measure whether recommendations are applicable (valuable) to classes "related" to the target one. It represents the possibility of using recommended attributes and operations with classes linked to the class that is being defined, e.g., via the subclass relation.

In our study, generalisation was measured as the percentage of recommendations that the participants deemed as *generalisable*. Table 7.7 shows the generalisation values of the methods, in the upper part considering all assessments, and in the bottom part considering only those assessments for which the participants were completely confident. In average, CosineCB was the method that generates less generalisable recommendations. This can be expected since it tends to be (over)specialised in the target's profile, that is, in the attributes and operations of the target class. ItemPop and CBUB achieved the same generalisation values when taking just completely confident assessments into account, but ItemPop slightly outperformed CBUB when considering all assessments. However, checking the most pop-

ular items that ItemPop recommends uncovers that they correspond to generic attributes like *name*, *title* and *address*.

Table 7.7: User study: Generalisation by confidence level.

Method	Confidence	#Part.	#Evals	g@1	g@2	g@3	g@4	g@5	Avg.
ItemPop	All	40	3,375	0.089	0.111	0.044	0.022	0.067	0.067
CosineCB				0.022	0.022	0.044	0.089	0.000	0.036
CBUB				0.089	0.022	0.178	0.044	0.044	0.076
ItemPop	Completely confident	9	225	0.429	0.286	0.357	0.571	0.286	0.386
CosineCB				0.500	0.286	0.214	0.429	0.286	0.343
CBUB				0.429	0.500	0.286	0.286	0.429	0.386

7.4 DISCUSSION

To answer RQ3 (*How do users perceive the recommendations of* Droid *recommenders?*), we focus on the metrics used in the user study.

Regarding precision, on individual domains, we observe average values ranging from 0.373 to 0.493 for the hybrid recommendation method CBUB (cf. Table 7.1). If we limit our attention to those evaluations in which participants stated a complete confidence with their assessments, the global average precision value of CBUB rises to 0.543 (cf. Table 7.2). Likewise, in contextualised cases, i.e., in diagrams with more than one class and more than one attribute/operation in the target class, CosineCB achieves precision values as high as 0.600, followed by CBUB with 0.556 (cf. Table 7.3). Comparing these results with others from related user studies [36, 52, 90], we can claim that the recommendations provided by Droid are perceived as highly precise, even considering that we did not use ad hoc recommendation methods for the task at hand, and we did not exploit very large datasets.

Moreover, the participants evaluated the generated recommendations as predominantly serendipitous, with average values ranging from 0.640 to 0.987 (cf. Table 7.4). In this sense, having both high precision and high serendipity makes Droid recommendations very valuable for users.

When it comes to the perceived redundancy of the recommendations, we observe significant differences between domains (cf. Table 7.5). Whereas the participants perceived almost no redundancy on the *Education* domain, recommendations on the *Literature* domain were evaluated as slightly redundant, and recommendations on the *Banking* domain seemed the most redundant ones. We argue that these results may depend on certain characteristics of the input datasets, and not on the recommendation method used. Nonetheless, specific recommendation methods to avoid redundancy (e.g., based on diversification techniques) could be researched [4].

As expected, the more context the class diagrams have, the more contextualised the recommendations are perceived (cf. Table 7.6). For

instance, average contextualisation values varied from 0.280 to 0.473 for the CBUB method, in non-contextualised and contextualised cases, respectively. The impact of these results in practice should be studied carefully. However, in our opinion, they are promising results that encourage further research on specific context-aware recommendation approaches for the addressed task.

Finally, participants did not feel the recommendations to be particularly generalisable to other classes in the context (cf. Table 7.7). Looking only at the evaluations coming from participants completely confident in their assessments, more recommendations were found to be generalisable, although in the case of the ItemPop method, they were also overly generic.

To answer RQ4 (*How do the offline experiment results compare to the ones of the user study?*), we compare the precision values of the recommendation methods in the offline experiment (Table 6.3) and the user study (Table 7.1). We observe that the precision achieved in both evaluations is high, being values of the user study results slightly greater. Specifically, the precision of CBUB was particularly similar in both evaluations, respectively achieving average values of 0.224 and 0.373 on the three addressed domains together. Likewise, considering each domain separately, the best performing method in both evaluations was CBUB on the *Banking* domain, with average values of 0.312 and 0.493.

When it comes to the other methods, the precision achieved in the user study is considerably higher for the CosineCB and ItemPop methods, except for CosineCB on the *Education* domain, where the method performed slightly better in the offline experiment (0.053 vs. 0.059 in average).

General higher precision values in the user study could be attributed to the fact that users may evaluate as correct valuable recommendations (e.g., an attribute called `account_number` for a class `Account`) that nonetheless are flagged as incorrect by the offline method, which may perform a stricter comparison (e.g., it may flag `account_number` as incorrect, while `account_id` would be correct). Still, we cannot always expect those results, since the quality of recommendations – as perceived by users – can only be proportional to the quality of the data the RS was trained with. Still, even with low-quality data, an offline evaluation may yield good results.

Overall, all the above results confirm that hybrid recommendations exploiting both content-based and collaborative filtering information are the most accurate since, as reported in the RSs literature, they help mitigating the particular weaknesses of the two types of approaches [12, 30].

Our experiment shows good correlation between the user study and the offline experiment, which is an important result, since most RSs that are built today are only evaluated with offline experiments,

or not at all [10, 28, 43, 44, 167]. Even if this may suggest that offline experiments could be a reasonable surrogate for user studies — much more costly to perform –-, it needs to be taken with caution. We argue that user studies – like the one presented here – are advisable to better understand the weak and strong points of a recommender, providing an assessment of the real value of recommendations.

7.5 THREATS TO VALIDITY

In this section, we examine threats to the validity of the reported study.

External validity. For the user study, we utilised the same domains and datasets as those in the offline experiment (cf. Chapter 6). Therefore, similarly to the offline experiment, the generality of the conclusions of the study could be strengthened if more domains and a bigger dataset were taken into account. Secondly, the participants in the user study performed the evaluation online from their respective work/study locations. We assumed that they had correctly understood their assignment, and did not monitor how long they took to complete it (counting from the moment they started doing it). To assess the participants' engagement, we inspected their responses, and found no fixed pattern in their assessments (e.g., always marking the first five recommendations as precise). However, to reduce the likelihood of participants giving random scores, and to make sure that everyone understood the task at hand, a more controlled user study in person could be conducted. Finally, the study involved 40 participants, who evaluated 3,375 cases. This represents a substantial amount of data, but further evidence could be obtained with a larger set of participants.

Internal validity. As mentioned in Chapter 6, in the offline experiment, we sought unbiased data by utilising a third-party search engine to collect the dataset and employing model search keywords instead of manual selection. Likewise, we attempted to reduce any possible bias in the results of the user study by choosing the targets for the experiment, the evaluation cases for each participant, and the order of the recommendation lists at random. Moreover, to avoid spurious effects caused by subjective individual assessments, five participants evaluated each recommendation. Fleiss Kappa's coefficients [59] for inter-rater agreement show statistically significant ($p < 0.001$), moderate agreement among raters in their assessments of correctness, serendipity and redundancy; and reveal fair agreement among raters regarding the assessments of contextualisation and generalisation[30]. This is reasonable since the latter aspects are more challenging to interpret

30 Discarding *"not confident at all"* and *"slightly confident"* assessments, we obtain agreement values of 0.467 for feature correct, 0.522 for obvious, 0.504 for redundant, 0.356 for contextualised and 0.288 for generalisable, with p-values lower than 0.001.

and evaluate without direct involvement in a class diagram modelling task.

Construct validity. In the user study, we applied established metrics in the RSs community. Specifically, the user study used precision, serendipity, redundancy, contextualisation and generalisation. However, the last two of them had to be slightly reformulated for the modelling domain. In addition, the fact that we considered these five indicators to assess the perception of users about the recommendations (rather than a single indicator) mitigates the possibility of construct under-representation.

Conclusion validity. The user study revealed no significant correlation between the metrics, except between precision and serendipity (0.545). In this case, to avoid unreliability in our measures, two authors of the paper carefully revised that they were computed correctly. Also note that, while using a small sample size can lead to low statistical power (i.e., finding no correlation when there is one), the somewhat homogeneous background of our participants mitigates this problem. In any case, further studies with more participants could be performed to strengthen our conclusions.

7.6 SUMMARY

We conducted a user study to assess how users perceive recommendations generated by DROID recommenders. Participants of the study, recruited via email, assessed recommendations from three recommendation methods (ItemPop, CosineCB, and CBUB) across three domains (*Banking*, *Education*, and *Literature*). A total of 40 participants evaluated the recommendations based on criteria such as correctness, obviousness, redundancy, contextualisation, and generalisability.

Regarding precision, which measures the accuracy of the recommendations, CBUB obtained the highest precision across all domains, consistent with the findings from the offline experiment. In terms of serendipity, which measures the unexpectedness of the recommendations, ItemPop generated slightly more serendipitous recommendations than the other methods. The Pearson correlation between precision and serendipity was moderate but positive. With respect to redundancy, the recommendations were generally distinct, with the *Banking* domain showing slightly higher redundancy. Regarding contextualisation, CBUB generated the most contextualised recommendations. Finally, the generalisation results indicated that CosineCB tended to be specialised in the target's profile, while ItemPop and CBUB showed better generalisation.

Overall, the precision results show that CBUB performed well on individual domains, particularly with high-confidence assessments and in contextualised cases. Comparing these results with related user studies suggests that DROID recommendations are perceived as

highly precise, even without using ad hoc recommendation methods or large datasets. Moreover, the recommendations were more contextualised when class diagrams had more context. This suggests promising results for further research on context-aware recommendation approaches for the task.

When compared to the offline experiment results, precision values in the user study were slightly higher. CBUB consistently performed well in both evaluations, and the general trend showed higher precision values in the user study. This could be attributed to users evaluating valuable recommendations that might be flagged as incorrect by the offline method, which performs a stricter comparison.

8

VALIDATION OF TOOLS

This chapter presents a validation of the tools developed during the thesis (cf. Chapter 5). The validation is conducted through two approaches. In the first approach, described in Section 8.1, we validate the manual integration of RSs generated with Droid into third-party modelling technologies. In the second approach, described in Section 8.2, we validate the automatic integration of RSs within existing Sirius and tree-based modelling editors. This second validation also serves to assess that the created tools facilitate the bridge between recommenders created for a particular modelling language and their reusability with other different languages. The chapter concludes with a summary in Section 8.3.

8.1 VALIDATION OF DROID

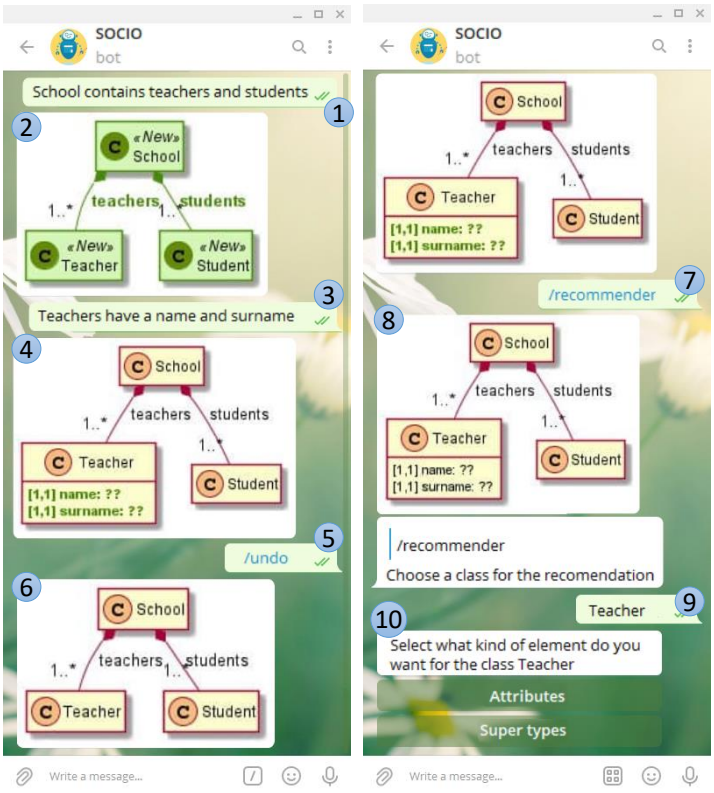
In this section, we validate that the RSs generated with Droid can be integrated with third-party modelling technologies. We aim to assess the usefulness of our approach in terms of its capacity to reuse RSs by their manual integration within existing modelling tools. Hence, we focus on the following research goal:

> **RG1:** *Exploring the integration of* Droid *generated recommenders into third-party modelling tools.*

For such purpose, we describe a case study on the manual integration of a RS specified with Droid into an external third-party modelling chatbot called Socio [123]. Socio 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 NL (Natural Language) as the modelling interface.

Figure 8.1(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) in order to buils a class diagram model interactively. 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).



a) SOCIO modelling examples b) Recommender command

Figure 8.1: Example of Socio interaction in Telegram.

Users usually do not provide all requirements in a single message, and so, Socio allows 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 *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, and download the model, as well as to undo and redo modelling actions. In Telegram, these commands start by a backslash followed by a keyword. Labels 5 and 6 in Figure 8.1(a) show an example of the undo command.

For this validation, we extended Socio with a RS specified with Droid and thus available as a service. Figure 8.2 shows a scheme of the integration of the RS within Socio, where the new components created for the integration are highlighted in green. Socio has a front-

end provided by Telegram and a back-end. The latter is the main component of the architecture since it handles all the functionalities of Socio: information and model storage, NL processing, and modelling actions. The Telegram client connects the user interaction in Telegram with the back-end.

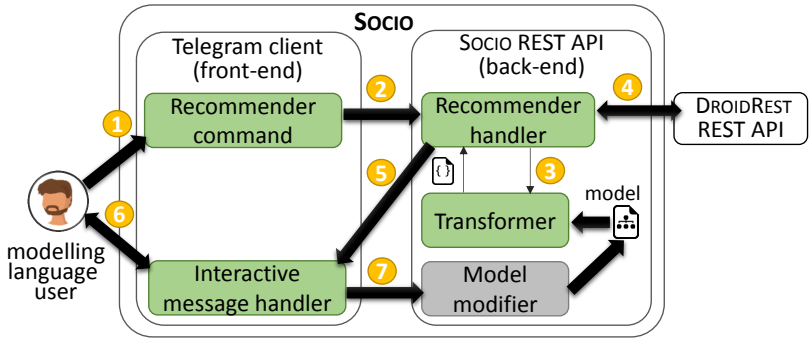


Figure 8.2: Architecture of Droid integration with Socio

For the integration, we created a *Recommender command* on the client side (label 1 in Figure 8.2). When the user types this command to obtain recommendations (label 1), the Telegram client sends a request to the back-end, which is handled by the *Recommender handler* (label 2). Since the model is internally represented with EMF, a *Transformer* converts the context element of the recommendation into the JSON format required by Droid (label 3). Then, the *Recommender handler* requests recommendations to the Droid service (label 4), and sends the returned recommendations back to the client (label 5). In the client, an *Interactive message handler* transforms the recommendations into an interactive message containing one button per recommended item (label 6). When the user selects one of these buttons, the handler sends a request to the back-end for adding the selected item to the model (label 7). Then, the selected button is deleted, while the other buttons remain available to enable applying further recommendations.

Figure 8.1(b) shows the usage of the `/recommender` command in Telegram. When a user types the command (label 7), Socio displays the current model and prompts the user to select a class (label 8). Once the user selects a class (label 9), Socio asks for the kind of items to be recommended for the class (label 10). Since Socio models do no support methods, the user can choose the recommendation of attributes and supertypes.

Figure 8.3 illustrates the recommendations provided by Droid. It shows the recommended supertypes (label 1) and attributes (label 2) for the class *Teacher*. In the former case, 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*. In the latter case, when the user presses the button with the recommen-

dation *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.

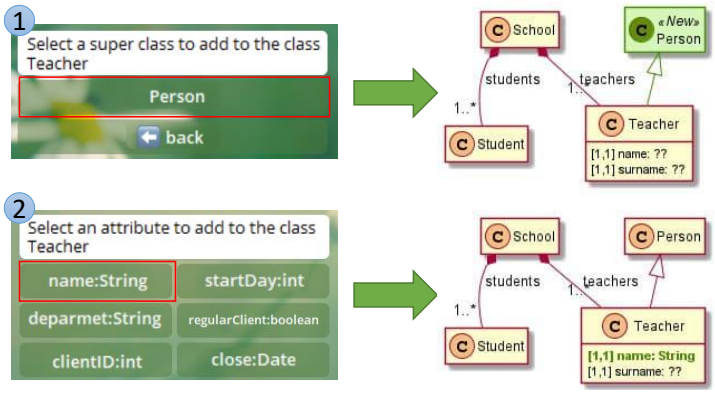


Figure 8.3: Droid recommendations in Socio.

Table 8.1 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 8.1: Metrics for integrating Droid with Socio.

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

We determine that the research goal RG1 has been successfully achieved. In the presented case study, the integration involved incorporating a RS generated with Droid into the third-party modelling tool Socio. This integration was accomplished by creating a `/recommender` command on the client side, resulting in the generation of less than 800 LoC. The low number of LoC indicates that Socio required minimal modifications to perform the integration.

8.2 VALIDATION OF IRONMAN

In this section, we validate the usefulness of IronMan in terms of its capacity to integrate Droid-generated RSs within existing Sirius and tree-based modelling editors, and its facility to reuse RSs created for a particular modelling language with other languages. Hence, we target the following research goals:

RG2: *Exploring the adaptability of* IronMan *in facilitating the adaptation of existing RSs to different languages.*

RG3: *Investigating the potential of* IronMan *for seamlessly integrating existing RSs into third-party modelling tools.*

For this purpose, we used two RSs for Ecore and UML reported in [9] and [11], which suggest attributes and operations for classes. Both RSs were created with Droid, and so, they were already deployed as services and are conformant to the API described in Section 4.4.

The aim for this validation is twofold. On the one hand, we wanted to assess if the RSs can be adapted to other modelling languages. In particular, we checked if the RSs can be adapted to UML, Ecore, Entity-Relationship (ER) diagrams, and the Interaction Flow Modelling Language (IFML) [31]. The three first languages are widely-used structural notations to define software systems, modelling languages, and databases. IFML is an OMG standard to define the content, user interaction and behaviour of the front-end of software applications. On the other hand, we wanted to assess the integration of the RSs into existing tree and Sirius editors built by third parties.

Table 8.2 summarises the experiment set-up. Each RS (e.g., for Ecore) was adapted to the other three languages (e.g., UML, ER and IFML) and integrated in all the six tools. This resulted in twelve integrations, covering environments based both on Sirius and the tree editor.

Table 8.2: IronMan validation: Set-up.

RS	Modelling language	Modelling tool
Ecore meta-models	Ecore	UML designer (Sirius)
UML class diagrams	UML	UML tree editor
	ER diagrams	Ecore tools (Sirius)
	IFML	Ecore tools (tree)
		ISD designer (Sirius)
		IFML editor (Sirius)

Table 8.3 summarises the results of the integrations, including the number of mappings needed to adapt the RS to the modelling language, and the synthesised LoC by the code generator. We did not need any mapping when using a RS for the same language (e.g., Ecore for Ecore), while for the other cases, we required from 5 to 9 mappings. Since the original RSs recommend both attributes and operations, the number of mappings depended on whether the language had a notion akin to operations (absent both in ER and IFML).

31 https://www.ifml.org/

Table 8.3: IronMan validation: Summary of the experiment results.

Integration	RS	Language	Editor	Maps	LoC
1	Ecore	Ecore	Ecore-tree	0	455
2	Ecore	Ecore	Ecore-Sirius	0	528
3	Ecore	UML	UML-tree	9	457
4	Ecore	UML	UML-Sirius	9	530
5	Ecore	ER	ISD-Sirius	5	414
6	Ecore	IFML	IFML-Sirius	5	414
7	UML	Ecore	Ecore-tree	9	451
8	UML	Ecore	Ecore-Sirius	9	525
9	UML	UML	UML-tree	0	452
10	UML	UML	UML-Sirius	0	525
11	UML	ER	ISD-Sirius	5	411
12	UML	IFML	IFML-Sirius	5	411

The built plugins used the EMF reflective API, and in average, 464 LoC were generated per plugin. This number does not include the implementation code to communicate with the recommender services. Additionally, in the case of Sirius, IronMan generates an *odesign* model automatically, which is the file that stores a description of the modelling environment, including the recommendation layer.

Figure 8.4 has some screenshots of the resulting integrations. Labels 1–3 show the integration of the Ecore RS with the Sirius editor provided by Ecore tools. Label 1 shows the menu to activate the recommendation layer, label 2 the menu contribution of the recommender, and label 3 the dialog from which to choose the recommendations. Label 4 displays the integration of the UML RS with the UML tree editor. Finally, label 5 displays the integration of the UML RS within the Information System Designer (ISD) [32]. The resulting plugins are available at: `https://github.com/ironman/generated-projects`.

Overall, both research goals (RGs) were successfully addressed.

For RG2, we could reuse RSs defined for Ecore or UML, and adapt them to other three languages (Ecore, UML, ER, IFML). The only requirement for this reuse was to map (subsets of) the meta-model of the RS and the meta-model of the target language, by defining between 0 and 9 declarative mappings.

For RG3, we could automatically integrate each RS into six existing modelling tools based on Sirius and EMF tree editors. Remarkably, all the tools were built by third parties, and we did not need their source code.

In summary, our experiment provides evidence that IronMan is able to reuse existing RSs for other modelling languages, and integrate them automatically into existing tools.

32 https://www.obeosoft.com/en/products/is-designer

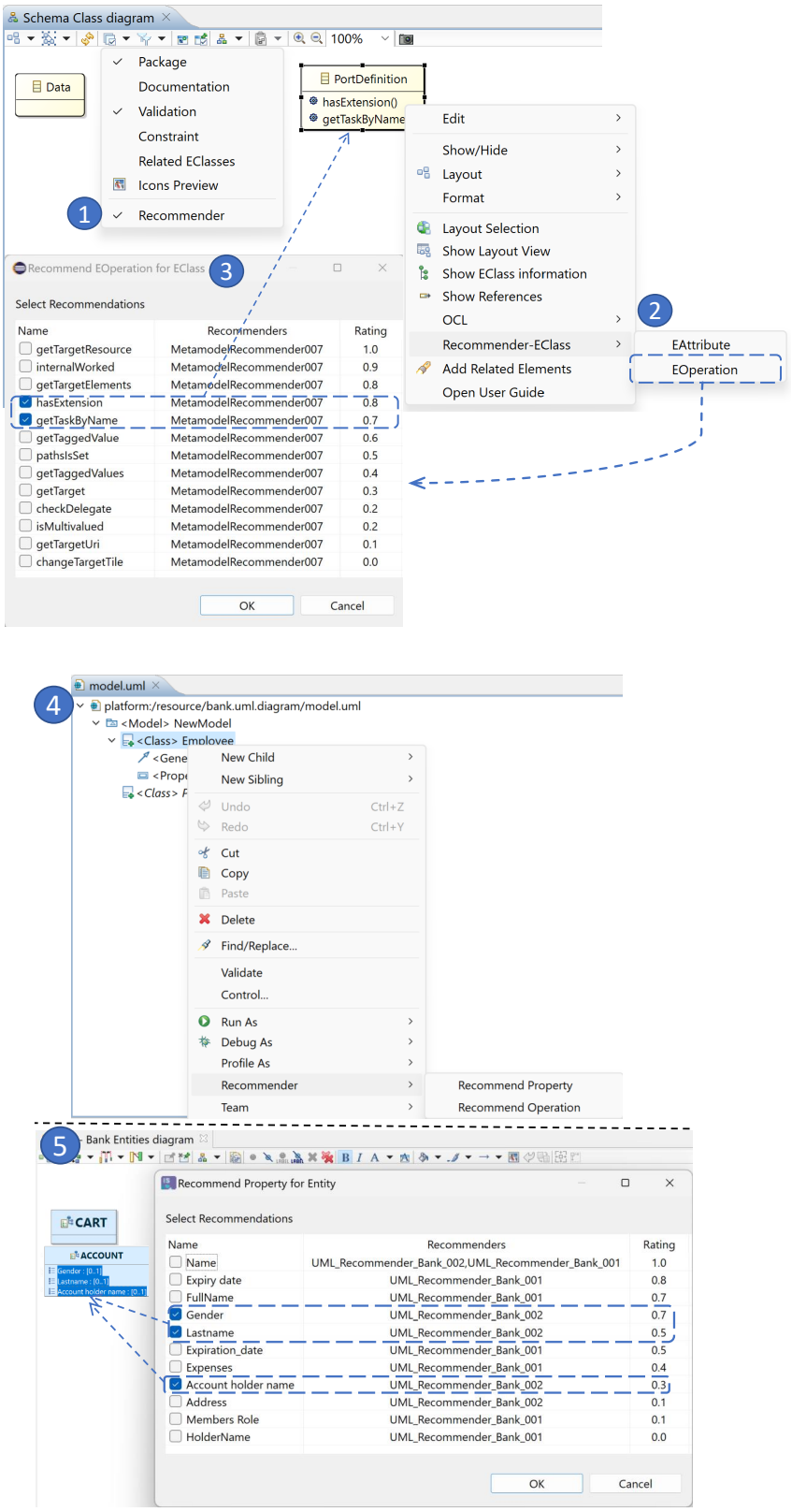


Figure 8.4: Screenshots of the integration results. (1–3) Ecore tools (Sirius), (4) UML tree editor, (5) ISD-Designer.

Regarding threats to the *external validity* of our validation, the number of RSs reused was limited, as we only considered RSs for Ecore and UML. Similarly, we reused these RSs just for four other modelling languages. A stronger validation would be obtained by considering more RSs and more languages. In particular, the languages in the experiment were somewhat similar, and we are aware that considering more structurally different notations would require from a more powerful mapping mechanism (e.g., based on OCL or Java), able to bridge the structural dissimilarities between the meta-models. This is future work. However, to mitigate this threat, we reused the RSs for well-known modelling languages developed by third parties, which are representative of structural modelling. Moreover, we also considered IFML, which is related to interaction modelling. Regarding the integration with tools, we chose existing tools built by third parties to avoid any bias. Still, integration with further tools would result in a stronger validation.

8.3 SUMMARY

We conducted two validations on the tools developed in this thesis. The first one aimed to validate the manual integration of RSs generated with Droid into third-party modelling technologies. The second one aimed to validate the automatic integration of RSs within existing Sirius and tree-based modelling editors.

The validation of Droid involved manually integrating a Droid-specified RS into the Socio modelling chatbot. The integration was achieved through the creation of a "/recommender" command, allowing users to request and receive recommendations within the Socio modelling chatbot. The integration was successful, requiring fewer than 800 LoC and minimal modifications to the architecture of Socio. This case study demonstrates the feasibility of integrating Droid-generated RSs into third-party modelling tools, emphasizing user-friendly commands and minimal code adjustments.

The second validation assessed the capability of IronMan to integrate Droid-generated RSs within existing Eclipse modelling tools and its adaptability to different modelling languages. The experiment involved adapting RSs for Ecore and UML to other languages (UML, Ecore, ER, IFML) and integrating them into six tools, covering both Sirius and tree-based editors. The RSs could be reused and adapted to different languages by using small mapping specifications. Moreover, IronMan facilitated the automatic integration into various modelling tools without requiring access to their source code.

Part IV

CONCLUSION

9 CONCLUSIONS AND FUTURE WORK

This chapter provides a comprehensive summary of the conclusions derived from the research undertaken in the thesis, and outlines potential directions for future research.

9.1 CONCLUSIONS

The main contributions of this thesis include the proposal of an MDE solution to automate the creation of RSs for modelling languages, and a solution to reuse and integrate existing recommenders within Sirius and tree-based modelling editors.

In the systematic mapping review presented in Section 3.1, we identified the recommendation tasks addressed in the literature, the prevalent recommendation techniques, and the drawbacks associated with RSs in the context of MDE. Existing RSs tailored for MDE predominantly target five key tasks: completion, creation, finding, repairing and reusing performed over models, meta-models, transformations, or code generators. The majority of approaches handle the completion and repair of models, with only a limited number focusing on the discovery of relevant artifacts and their reuse within specific contexts.

Moreover, a substantial percentage of RSs in MDE are content-based and knowledge-based, designed specifically for particular modelling languages or tools. This leaves ample room for further exploration in leveraging collaborative filtering in MDE, as well as exploring less commonly used algorithms. Given the significant development effort required for RSs, the objective of this dissertation was to develop an approach facilitating the adoption of RSs in MDE. Hence, the proposed model-driven solution aims to automate the synthesis of RSs for specific modelling languages, with a focus on the completion task for both models and meta-models. Additionally, we emphasise the reusability of such recommenders in diverse contexts. The incorporation of two extension points allows for the seamless integration of new recommendation methods and data encoding techniques.

Another challenge arises from the scarcity of data available for training the recommenders. The acquisition and preprocessing of data (models in our context) are essential steps. To address this issue, we introduced an extension point for data collection, and provided access to two well-known repositories of models and meta-models, as well as direct access to directory folders. Additionally, we incorporated a variety of data preprocessing options. This approach enables

the swift exploration and evaluation of multiple preprocessing options to identify the most effective ones for a given dataset.

Furthermore, only 71% of the approaches proposed in the literature were evaluated through offline experiments, with a primary focus on ranking accuracy metrics such as precision, recall and F-measure. For this reason, we extended our proposal to include seven metrics for offline experiments, whose values are displayed in a drop-down colour-coded view to facilitate interpretation. Our ultimate goal was to devise solutions that enable the customisation of recommendation algorithms, user preferences, or evaluation metrics to cater to the specific needs of users.

These contributions have been realised in the form of two Eclipse plugins named Droid and IronMan. Droid is a framework automating the configuration, evaluation, synthesis and deployment of RSs specifically tailored for modelling languages. Droid has an extensible architecture that enables the integration of data encoding techniques, recommendation methods, and data sources. It offers a DSL to configure every aspect of the RS, covering the type of recommended modelling items, the recommendation method, the gathering and preprocessing of training data, and the methodology and metrics for evaluating the created RS. The RSs generated by Droid are deployed as a REST service, facilitating its integration with a variety of modelling tools.

On the other hand, IronMan facilitates the reuse of recommenders developed for a modelling language with other similar notations, as well as their automated integration within modelling tools. Moreover, it supports the aggregation of recommendations coming from several RSs. Similar to Droid, IronMan features an extensible architecture that allows for the integration of additional aggregation methods and modelling editors.

The tools have been evaluated from various perspectives. Droid recommenders were assessed for effectiveness through an offline experiment using over 7,600 UML models across three domains. The outcomes show that the precision of Droid recommendations are inline, or surpass, that of RSs specifically created for the same recommendation task. Moreover, we performed a user study involving 40 participants. The results of this study indicated that the recommendations were perceived as highly precise, largely serendipitous, slightly or not redundant (depending on the domain), albeit not particularly generalisable. Interestingly, the study also highlighted that the more context the models have, the more contextualised the recommendations are perceived. The precision results of the offline experiment and the user study are consistent, just slightly better for the user study. Moreover, the hybrid recommendation methods demonstrated the highest accuracy in our evaluation.

We also conducted validation exercises for both tools. First, we manually integrated a RS generated by DROID with a third-party modelling chatbot called Socio. This integration was accomplished by creating a "/recommender" command on the client side, which required less than 800 LoC. This low number of LoC indicates that minimal modifications were required to perform the integration. Subsequently, we assessed the versatility of IRONMAN by integrating two recommenders designed for four languages into six modelling tools. We successfully repurposed RSs initially defined for Ecore or UML, adapting them to three additional languages (Ecore, UML, ER, IFML). The only prerequisite for this reuse was to map (subsets of) the meta-model of the RS and the meta-model of the target language, which required defining between 0 and 9 declarative mappings. Thus, we could seamlessly integrate each RS into six existing modelling tools based on Sirius and EMF tree editors. It ia worth highlighting that all these tools were developed by third parties, and we accomplished the integration without requiring access to their source code.

9.2 FUTURE WORK

This thesis has opened up several research avenues. First, concerning our framework DROID, we plan to expand the current library of extensions to integrate additional data sources, data encodings, and recommendation methods (e.g., based on neural networks or built ad-hoc). We would also like to include new extension points in DROID; in particular, to enable the inclusion of specific preprocessing options and user-defined evaluation metrics. For instance, DROID has particular metric implementations of recommendation diversity and coverage, but there are alternative ways to measure these characteristics. In addition, one could define metrics oriented to the task at hand, such as the balance of recommendations in terms of the underlying item types (e.g., classes, methods and attributes in object-oriented modelling).

DROID currently automates the offline evaluation of the created RSs. To complement this facility, we plan to incorporate an automatic generator of the infrastructure needed to conduct user studies of RSs, assisting in the selection of the evaluation cases and the recommendation methods to include in the study, and automating the analysis of the evaluation results. Moreover, we will consider extending DROID to support other modelling tasks, like (sub-)model reuse, model repair, or model optimisation and improvement.

To complement our empirical experiments (i.e., the offline experiments and user studies), we will aim at performing an online experiment in which users are requested to freely perform modelling tasks through a tool, with and without the support of a RS. In such experiment, the utility of recommendations could be defined by alternative

aspects, like the percentage of recommendations accepted/rejected by users, the time spent to perform the tasks with/without using a recommender, and the quality of the results.

With regards to the aggregation of RSs, we would like to integrate additional techniques, and investigate the scenarios where recommendation aggregation results more beneficial. While we currently support unsupervised rank aggregation techniques, we plan to extend our framework with supervised ones. These techniques require a pre-configuration step to optimise a recommendation aggregation function with respect to a given metric, typically precision.

Implementation-wise, the IronMan wizard requires that all selected RSs are defined for the same modelling language, and then adapted if needed. We will support the selection of RSs for different languages, and provide assistants to adapt them to the modelling languages (once the first mapping is defined). Besides, we would also like to include more flexible means for adapting the RSs to the modelling language (e.g., using OCL or Java snippets). We also plan to support integration with textual notations (e.g., defined using Xtext). Finally, we are interested in exploring more sophisticated types of integration of the RSs within modelling tools (e.g., a proactive approach where the RS monitors the modelling session and triggers recommendations when an opportunity is found).

10 CONCLUSIONES Y TRABAJO FUTURO

Este capítulo proporciona un resumen de las conclusiones derivadas de la investigación realizada en esta tesis, y esboza las posibles líneas de investigaciones futuras.

10.1 CONCLUSIONES

Las principales contribuciones de esta tesis incluyen la propuesta de una solución de ingeniería dirigida por modelos (MDE, por sus siglas en inglés) para automatizar la creación de SR para lenguajes de modelado, así como una solución para reutilizar e integrar recomendadores existentes en editores de modelado Sirius y en editores en forma de árbol.

En la revisión sistemática presentada en la Sección 3.1, identificamos las tareas de recomendación abordadas en la literatura, las técnicas de recomendación prevalentes, y las limitaciones de los SR en el contexto de MDE. Los SR existentes para MDE abordan principalmente cinco tareas clave: completar, crear, encontrar, reparar y reutilizar meta-modelos, modelos, transformaciones o generadores de código. La mayoría de los enfoques se centran en completar y reparar modelos, y sólo un número reducido tiene como objetivo el descubrimiento de artefactos relevantes y su reutilización en contextos específicos.

Además, un porcentaje sustancial de SR en MDE son basados en contenido y en conocimiento, y están diseñados para lenguajes o herramientas de modelado particulares. Esto deja margen para explorar el uso de técnicas de filtrado colaborativo en MDE, así como para explorar algoritmos menos utilizados. Dado el esfuerzo significativo de desarrollo que requieren los SR, el objetivo de esta tesis fue desarrollar un enfoque que facilitara la adopción de SR en MDE. Para ello, la solución dirigida por modelos propuesta busca automatizar la síntesis de SR para lenguajes de modelado específicos, centrados en la tarea de completar modelos y meta-modelos. Además, enfatizamos la reutilización de esos recomendadores en diversos contextos. Tambien proporcionamos dos puntos de extensión que permiten la integración de nuevos métodos de recomendación y técnicas de codificación de datos.

Otro desafío al que nos enfrentamos surge de la escasez de datos para entrenar los recomendadores. La adquisición y preprocesamiento de datos (modelos en nuestro contexto) son fases esenciales. Para abordar este problema, introdujimos un punto de extensión para la

recopilación de datos, y proporcionamos acceso a dos repositorios conocidos de modelos y meta-modelos, así como acceso directo a los directorios del sistema de ficheros. Además, incorporamos diversas opciones de preprocesamiento de datos. Este enfoque permite la rápida exploración y evaluación de múltiples opciones de preprocesamiento para identificar las más efectivas para un conjunto de datos en particular.

Además, cabe destacar que solo el 71% de los enfoques propuestos en la literatura se evaluaron mediante experimentos offline, centrándose principalmente en métricas de precisión de rankings como precisión, recall y F1. Por esta razón, ampliamos nuestra propuesta para incluir siete métricas para experimentos offline, cuyos resultados se muestran en una vista desplegable con distintos colores para facilitar su interpretación. Nuestro objetivo final fue diseñar soluciones que permitieran la personalización de los algoritmos de recomendación, preferencias del usuario, y métricas de evaluación para satisfacer las necesidades específicas de los usuarios.

Estas contribuciones se han materializado en forma de dos plugins de Eclipse llamados Droid y IronMan.

Droid es un framework que automatiza la configuración, evaluación, síntesis e implementación de SR para lenguajes de modelado. Droid tiene una arquitectura flexible que permite la integración de técnicas de codificación de datos, métodos de recomendación, y fuentes de datos. Ofrece un lenguaje de dominio específico para configurar todos los aspectos del SR, abarcando el tipo de elementos de modelado a recomendar, el método de recomendación, la recopilación y el preprocesamiento de datos de entrenamiento, y la metodología y métricas para evaluar el SR creado. Los SR generados por Droid se despliegan como un servicio REST, facilitando su integración con una variedad de herramientas de modelado.

Por otro lado, IronMan facilita la reutilización de recomendadores desarrollados para un lenguaje de modelado con otras notaciones similares, así como su integración automática dentro de herramientas de modelado. Además, admite la agregación de recomendaciones provenientes de varios SR. Al igual que Droid, IronMan cuenta con una arquitectura flexible que permite la integración de métodos de agregación y editores de modelado adicionales.

Las herramientas desarrolladas se han evaluado desde varias perspectivas. La efectividad de los recomendadores generados con Droid se evaluó mediante un experimento offline utilizando más de 7,600 modelos UML de tres dominios distintos. Los resultados muestran que la precisión de las recomendaciones de Droid se alinea o supera a la de SR creados específicamente para la misma tarea de recomendación. Además, realizamos un estudio de usuarios con 40 participantes. Los resultados de este estudio indican que las recomendaciones fueron percibidas como altamente precisas, en gran me-

dida fortuitas, ligeramente o no redundantes (dependiendo del dominio), aunque no particularmente generalizables. El estudio también destacó que cuanto más contexto tienen los modelos, más contextualizadas se perciben las recomendaciones. Los resultados de precisión del experimento offline y el estudio de usuarios son consistentes, siendo ligeramente mejores los del estudio de usuarios. Además, los métodos de recomendación híbridos demostraron la mayor precisión en nuestra evaluación.

También realizamos validaciones de ambas herramientas. Primero, integramos manualmente un SR generado por DROID con un chatbot de modelado llamado Socio. Esta integración se logró creando un comando "/recommender" en el lado cliente, lo que requirió menos de 800 líneas de código. Este bajo número de líneas de código indica que la integración sólo requirió modificaciones mínimas. Posteriormente, evaluamos la versatilidad de IRONMAN integrando dos recomendadores en cuatro lenguajes distintos al de su diseño original, en seis herramientas de modelado. En concreto, pudimos reutilizar con éxito SR inicialmente definidos para Ecore o UML, adaptándolos a tres lenguajes adicionales (Ecore, UML, ER, IFML). El único requisito para esta reutilización fue mapear (subconjuntos de) el meta-modelo del SR y el meta-modelo del lenguaje objetivo, lo que requirió definir entre 0 y 9 mapeos declarativos. Así, pudimos integrar sin problemas cada SR en seis herramientas de modelado existentes basadas en Sirius y editores de árboles EMF. Cabe destacar que todas estas herramientas fueron desarrolladas por terceros, y logramos la integración sin necesidad de acceder a su código fuente.

10.2 TRABAJO FUTURO

Esta tesis ha abierto varias lineas de investigación. En primer lugar, en relación con nuestro framework DROID, planeamos expandir la biblioteca actual de extensiones para integrar nuevas fuentes de datos adicionales, codificaciones de datos, y métodos de recomendación (por ejemplo, basados en redes neuronales o construidos ad-hoc). También nos gustaría incluir nuevos puntos de extensión en DROID; en particular, para permitir la inclusión de opciones de preprocesamiento de datos específicas, y métricas de evaluación definidas por el usuario. Por ejemplo, DROID tiene implementaciones específicas de métricas de diversidad y cobertura de recomendaciones, pero existen formas alternativas de medir estas características. Además, se podrían definir métricas orientadas a la tarea en cuestión, como puede ser el equilibrio de recomendaciones en términos de los tipos de elementos subyacentes (por ejemplo, clases, métodos y atributos en el modelado orientado a objetos).

DROID actualmente automatiza la evaluación offline de los SR creados. Para complementar este trabajo, planeamos incorporar un gener-

ador automático de la infraestructura necesaria para realizar estudios de usuarios de SR, que ayude en la selección de los casos de evaluación y los métodos de recomendación a incluir en el estudio, y que automatice el análisis de los resultados de la evaluación. Además, consideraremos extender Droid para admitir otras tareas de modelado, como la reutilización de (sub)modelos, la reparación de modelos, o la optimización y mejora de modelos.

Para complementar nuestros experimentos empíricos (es decir, los experimentos offline y los estudios con usuarios), planeamos realizar un experimento online en el que se solicite a los usuarios que realicen tareas de modelado de forma libre a través de una herramienta, con y sin el apoyo de un SR. En dicho experimento, la utilidad de las recomendaciones podría definirse por aspectos alternativos, como el porcentaje de recomendaciones aceptadas/rechazadas por los usuarios, el tiempo empleado para realizar las tareas con/sin el uso de un recomendador, y la calidad de los resultados.

En cuanto a la agregación de SR, nos gustaría integrar técnicas adicionales e investigar los escenarios en los que la agregación de recomendaciones resulta más beneficiosa. Si bien actualmente soportamos técnicas de agregación de clasificación no supervisada, planeamos extender nuestro marco con técnicas supervisadas. Estas técnicas requieren un paso de preconfiguración para optimizar una función de agregación de recomendaciones con respecto a una métrica específica, generalmente la precisión.

En cuanto a la implementación, el asistente de IronMan requiere que todos los SR seleccionados estén definidos para el mismo lenguaje de modelado, y luego se adapten si es necesario. En el futuro, permitiremos la selección de SR para diferentes lenguajes, y proporcionaremos asistentes para adaptarlos a los lenguajes de modelado (una vez definido el primer mapeo). Además, también nos gustaría incluir medios más flexibles para adaptar los SR al lenguaje de modelado (por ejemplo, mediante fragmentos de OCL o Java). También planeamos admitir la integración con notaciones textuales (por ejemplo, definidas mediante Xtext). Finalmente, estamos interesados en explorar otros tipos más sofisticados de integración de los SR dentro de las herramientas de modelado (por ejemplo, un enfoque proactivo donde el SR monitorea la sesión de modelado y activa recomendaciones cuando se encuentra una oportunidad).

Part V

APPENDIX

11

APPENDIX A

This appendix contains the tables related to the Chapter 3 State of the art.

Table 11.1: Purpose of recommendation vs. recommended artefacts: approaches marked with * are language-independent.

Purpose \ Arte	Meta-Model	Model	Transformation	Code Generator
Complete	Burgueño et al. [28] DoMoRe [7, 8] Hermes [49, 50, 51] Kögel et al. [86, 87] MemoRec [41] MORGAN* [43] NEMO [44] Refacola [157] Weyssow et al. [167]	Baya [37] Deng et al. [39] DiaGen* [101] DIG MDE [111] DoMoRe* [7, 8] Elkamel et al. [52] Heinemann [69] Hermes* [49, 50, 51] IPSE [63] Kermeta* [105] Kögel et al.* [86, 87] Koschmider et al. [71, 72, 89] MORGAN* [43] Nair et al. [15] NEMO [44] Li et al. [94] PME* [119] Rangiha et al. [129] RapMOD [90, 91, 109] Refacola* [157] Savary-Leblanc [146, 147] Sen et al.* [151, 152] SimIMA [3, 159] Shilov et al.* [154] SMART [67]	AXSM [73] CONVErT [17]	
Create	DSL-maps [124]	UCcheck [14]	OCKHAM* [160]	
Find	Extremo [148, 149, 150]	Cerqueira et al. [36] Extremo* [148, 149, 150] Matikainen et al. [100] SBPR [83, 84]		
Repair	Batot et al. [21] Clarisó et al. [38] PARMOREL [18, 19, 74] Refacola [157] Shilov et al.* [154]	AMOR [27] Anguel et al.* [13] ASIMOV* [61] BAM [168] BPMoQualAssess [80] B-repair [31] DiaGen* [101] DPF* [128] IntellEdit* [112] Mani et al.* [99] MDSafeCer [106] Nassar et al.* [110] PARMOREL* [18, 19, 74] Refacola* [157] ReVision* [115, 116] SMART [67]	anATLyzer [142, 143, 144]	Mani et al. [99]
Reuse	Hermes [49, 50, 51]	Hermes* [49, 50, 51] Koschmider et al. [71, 72, 89] Paydar et al. [120, 121] REBUILDER [64] SimIMA [3, 159]	Refactory [131]	
Other Purpose		Bobek et al. [24] MAGNET [1] ModBud [140]		

Table 11.2: Recommender systems for MDE: Tooling (part 1). We use *n.a.* as abbreviation for *not applicable* and *unk* as abbreviation for *unknown*.

Approaches	Maturity	Tool Independent	Recommender Trigger OnDemand	Recommender Trigger Proactive	Recommendation Enactment Manual	Recommendation Enactment Interact.	Recommendation Enactment Auto.	Recommendation Enactment Semi-
AMOR [27]	Plug-in		✓			✓		
anATLyzer [142, 143, 144]	System		✓			✓		
Anguel et al. [13]	Prototype		✓					✓
ASIMOV [61]	System		✓					✓
AXSM [73]	Plug-in		✓			✓		
BAM [168]	Plug-in			✓	✓			
Batot et al. [21]	Framework	✓	✓		✓			
Baya [37]	Prototype			✓		✓		
Bobek et al. [24]	Prototype		✓				✓	
BPMoQualAssess [80]	Prototype		✓		✓			
Burgueño et al. [28]	Prototype	✓	✓				✓	
B-repair [31]	System		✓			✓		
Cerqueira et al. [36]	Prototype		✓		✓			
Clarisó et al. [38]	Proposal	*n.a.*	✓		✓			
CONVErT [17]	Prototype			✓		✓		
Deng et al. [39]	Prototype		✓			✓		
DiaGen [101]	System		✓			✓		
DIG MDE [111]	System		✓		✓		✓	
DoMoRe [7, 8]	System		✓	✓		✓		
DPF [128]	Prototype		✓				✓	
DSL-maps [124]	Plug-in		✓			✓		
Elkamel et al. [52]	Prototype			✓		✓		
Extremo [148, 149, 150]	Plug-in	✓	✓			✓		
Heinemann [69]	Prototype		✓		✓			
Hermes [49, 50, 51]	Framework, Plug-in	✓	✓	✓	✓	✓	✓	✓
IntellEdit [112]	Framework		✓			✓		
IPSE [63]	Plug-in		✓			✓		
Kermeta [105]	Plug-in		✓				✓	
Kögel et al. [86, 87]	Prototype			✓	✓			
Koschmider et al. [71, 72, 89]	Prototype		✓			✓		
Li et al. [94]	Prototype		✓			✓		
MAGNET [1]	System		✓		✓			
Mani et al. [99]	Prototype		✓		✓			
Matikainen et al. [100]	Prototype			✓			✓	
MDSafeCer [106]	Plug-in		✓		✓			
MemoRec [41]	Prototype	✓	*unk*	*unk*	*unk*	*unk*	*unk*	*unk*
MORGAN [43]	Prototype	✓	*unk*	*unk*	*unk*	*unk*	*unk*	*unk*
ModBud [140]	Proposal	*n.a.*	✓	✓	✓			
Nair et al. [15]	Plug-in			✓				✓
Nassar et al. [110]	Plug-in		✓			✓	✓	
NEMO [44]	Prototype	✓	*unk*	*unk*	*unk*	*unk*	*unk*	*unk*
OCKHAM [160]	Prototype	*unk*	*unk*	*unk*	*unk*	*unk*	*unk*	*unk*
PARMOREL [18, 19, 74]	Plug-in		✓				✓	
Paydar et al. [120, 121]	Prototype		✓			✓		
PME [119]	Plug-in			✓				✓

Table 11.3: .
Recommender systems for MDE: Tooling (part 2). We use *n.a.* as abbreviation for *not applicable* and *unk* as abbreviation for *unknown*.

		Tool	Recommender Trigger		Recommendation Enactment			
Approaches	Maturity	Independent	OnDemand	Proactive	Manual	Interact.	Auto.	Semi-
Rangiha et al. [129]	Prototype			✓		✓		
RapMOD [90, 91, 109]	Plug-in			✓			✓	
REBUILDER [64]	System		✓		✓			
Refacola [157]	Prototype	✓	✓			✓		
Refactory [131]	Prototype		✓		✓			
ReVision [115, 116]	System		✓			✓		
Savary-Leblanc [146, 147]	Plug-in			✓			✓	
SBPR [83, 84]	System		✓		✓			
Sen et al. [151, 152]	System		✓		✓			
Shilov et al. [154]	Proposal	*n.a.*	*unk*	*unk*	*unk*	*unk*	*unk*	*unk*
SimIMA [3, 159]	Prototype		✓			✓		
SMART [67]	System		✓			✓		
UCcheck [14]	System		✓			✓		
Weyssow et al. [167]	Prototype	✓	*unk*	*unk*	*unk*	*unk*	*unk*	*unk*

Table 11.4: Recommender systems for MDE: Recommendation method (part 1).

Approaches	Acquisition Type	Temporality	Recommended Item	Recommendation Degree	
				Cardinality	Ranking
Content-based					
AXSM [73]	Implicit, Explicit	Short-term	Artefact fragment	All	✓
Cerqueira et al. [36]	Implicit, Explicit	Short-term	Complete artefact	N	✓
CONVErT [17]	Implicit, Explicit	Short-term	Artefact fragment	N	✓
DoMoRe [7, 8]	Implicit	Short-term	Artefact fragment	All	✓
Elkamel et al. [52]	Implicit	Short-term	Artefact fragment	All	
Extremo [148, 149, 150]	Explicit	Short-term	Artefact fragment	All	✓
MORGAN [43]	Implicit	Short-term	Artefact fragment	N	✓
Paydar et al. [120, 121]	Implicit	Short-term	Complete artefact	N	✓
Collaborative filtering					
MAGNET [1]	Implicit	Short-term	Modelling advice	All	✓
Matikainen et al. [100]	Implicit	Short-term	Complete artefact	N	✓
MemoRec [41]	Implicit	Short-term	Artefact fragment	N	✓
ModBud [140]	Implicit, Explicit	*unknown*	Modelling advice	*unknown*	*unknown*
PARMOREL [18, 19, 74]	Implicit, Explicit	Long-term	Editing actions	One	
Refactory [131]	Implicit	Short-term	Artefact fragment	N	
Shilov et al. [154]	Implicit	Short-term	Artefact fragment	All	
Knowledge-based					
AMOR [27]	Implicit	Short-term	Editing actions	All	✓
Anguel et al. [13]	Implicit	Short-term	Editing actions	All	
ASIMOV [61]	Implicit	Short-term	Editing actions	N	
BAM [168]	Implicit	Short-term	Editing actions	All	✓
Baya [37]	Implicit	Short-term	Artefact fragment	N	✓
Bobek et al. [24]	Implicit	Short-term	Artefact fragment	N	✓
BPMoQualAssess [80]	Implicit	Short-term	Modelling advice	All	
Burgueño et al. [28]	Implicit, Explicit	Short-term, Long-term	Artefact fragment	N	✓
Deng et al. [39]	Implicit	Short-term	Artefact fragment	N	✓
DiaGen [111]	Implicit	Short-term	Editing actions	All	
DIG MDE [111]	Implicit, Explicit	Short-term	Artefact fragment	All	
DPF [128]	Implicit	Short-term	Artefact fragment	All	
DSL-maps [124]	Implicit	Short-term	Artefact fragment	All	✓
IPSE [63]	Implicit	Short-term	Artefact fragment, modelling advice	One	
Kermeta [105]	Implicit	Short-term	Artefact fragment	All	
Li et al. [94]	Implicit	Short-term	Artefact fragment	N	✓
Mani et al. [99]	Implicit	Short-term	Editing actions	All	
MDSafeCer [106]	Implicit	Short-term	Modelling advice	All	
Nassar et al. [110]	Implicit	Short-term	Editing actions	All	
RapMOD [90, 91, 109]	Implicit	Short-term	Artefact fragment	N	✓
REBUILDER [64]	Implicit	Short-term	Complete artefact, artefact fragment	All	✓
Refacola [157]	Implicit	Short-term	Editing actions	N	
ReVision [115, 116]	Implicit	Short-term	Editing actions	All	✓
Savary-Leblanc [146, 147]	Implicit	Short-term	Artefact fragment	All	✓
Sen et al. [151, 152]	Implicit, Explicit	Short-term	Artefact fragment	N	
SimIMA [3, 159]	Implicit, Explicit	Short-term	Complete artefact, artefact fragment	All	✓
UCcheck [14]	Implicit	Short-term	Modelling advice	All	
Weyssow et al. [167]	Implicit	Short-term	Artefact fragment	N	✓

Table 11.5: Recommender systems for MDE: Recommendation method (part 2).

Approaches	Acquisition Type	Temporality	Recommended Item	Recommendation Degree	
				Cardinality	Ranking
Hybrid: Content-based, collaborative filtering					
Heinemann [69]	Implicit	Short-term	Artefact fragment	N	✓
Kögel et al. [86, 87]	Implicit	Short-term	Editing actions	N	
Koschmider et al. [71, 72, 89]	Implicit, Explicit	Short-term	Complete artefact, artefact fragment	N	✓
Hybrid: Content-based, knowledge-based					
B-repair [31]	Implicit	Short-term	Editing actions	N	✓
Hybrid: Content-based, social-based					
Rangiha et al. [129]	Implicit, Explicit	Long-term	Artefact fragment	All	✓
SBPR [83, 84]	Implicit	Short-term	Complete artefact	All	✓
Other Method					
anATLyzer [142, 143, 144]	Implicit	Short-term	Editing actions	All	✓
Batot et al. [21]	Implicit	Short-term	Editing actions	N	✓
Clarisó et al. [38]	Implicit, Explicit	Short-term	Editing actions	N	✓
IntellEdit [112]	Implicit	Short-term	Editing actions	All	✓
Nair et al. [15]	Implicit	Short-term, Long-term	Editing actions	All	✓
NEMO [44]	Implicit	Short-term	Editing actions	N	✓
OCKHAM [160]	Implicit	Short-term	Editing actions	N	✓
PME [119]	Implicit	Short-term	Artefact fragment	All	✓
SMART [67]	Implicit	Short-term	Editing actions	All	
Any Method					
Hermes [49, 50, 51]	Implicit, Explicit	Long-term	Editing actions	N	✓

Table 11.6: Recommender systems for MDE: Evaluation.

Approaches	Application Independent				Application Dependent
	Ranking Accuracy	Other Measure	System Performance	Usage Satisfaction	
Offline experiment					
AMOR [27]					✓
anATLyzer [144]					✓
Batot et al. [21]					✓
Baya [37]	✓		✓		
Burgueño et al. [28]	✓	✓	✓		
B-repair [31]	✓				
CONVErT [17]	✓				
Deng et al. [39]	✓		✓		
DiaGen [101]			✓		
DIG MDE [111]			✓		
Extremo [148, 150, 159]			✓		✓
Heinemann [69]	✓				
IntellEdit [112]	✓				✓
Kögel et al. [86, 87]	✓				
Li et al. [94]	✓		✓		
Mani et al. [99]					✓
Matikainen et al. [100]					✓
NEMO [44]	✓	✓	✓		
MemoRec [41]	✓	✓	✓		
MORGAN [43]	✓	✓			
OCKHAM [160]	✓				
PARMOREL [18, 19, 74]			✓		
PME [119]			✓		✓
Refacola [157]	✓		✓		✓
Refactory [131]					✓
Savary-Leblanc [146, 147]		✓			
SBPR [84]	✓				
SimIMA [159, 3]	✓				
Shilov et al. [154]		✓			
Weyssow et al. [167]	✓				
Online experiment					
ASIMOV [61]			✓		✓
DoMoRe [7, 8]				✓	
User study					
anATLyzer [144]					✓
AXSM [73]				✓	
Baya [37]			✓	✓	
Cerqueira et al. [36]	✓			✓	
CONVErT [17]				✓	
DSL-maps [124]				✓	
Elkamel et al. [52]	✓				
IPSE [63]				✓	
Koschmider et al. [71, 72, 89]			✓	✓	✓
MAGNET [1]				✓	
Nair et al. [15]	✓			✓	
OCKHAM [160]				✓	
Paydar et al. [121, 120]	✓				
RapMOD [90, 91, 109]	✓		✓		✓
ReVision [115, 116]			✓		✓
No evaluation					
[13, 14, 24, 38, 49, 50, 51, 64, 67, 80, 105, 106, 110, 128, 129, 140, 151, 152, 168]					

Table 11.7: Recommender systems for MDE: Evaluation vs metrics.

Metrics \ Evaluation	Offline Experiment	Online Experiment	User Study
Ranking Accuracy	Baya [37] Burgueño et al. [28] B-repair [31] CONVErT [17] Deng et al. [39] Heinemann [69] IntellEdit [112] Kögel et al. [86, 87] Li et al. [94] NEMO [44] MemoRec [41] MORGAN [43] OCKHAM [160] Refacola [157] SBPR [84] SimIMA [3, 159] Weyssow et al. [167]		Cerqueira et al. [36] Elkamel et al. [52] Nair et al. [15] Paydar et al. [121, 120] RapMOD [90, 91, 109]
Other Measure	Burgueño et al. [28] NEMO [44] MemoRec [41] MORGAN [43] Savary-Leblanc [147] Shilov et al. [154]		
System Performance	Baya [37] Burgueño et al. [28] Deng et al. [39] DiaGen [101] DIG MDE [111] Extremo [148, 149, 150] Li et al. [94] NEMO [44] MemoRec [41] PARMOREL [18, 19, 74] PME [119] Refacola [157]	ASIMOV [61]	Baya [37] Koschmider et al. [71, 72, 89] RapMOD [90] ReVision [115, 116]
Usage Satisfaction		DoMoRe [7, 8]	AXSM [73] Baya [37] Cerqueira et al. [36] CONVErT [17] DSL-maps [124] IPSE [63] Koschmider et al. [71, 72, 89] Nair et al. [15] MAGNET [1] OCKHAM [160]
Application-Dependent	AMOR [27] anATLyzer [144] Batot et al. [21] Extremo [148, 149, 150] IntellEdit [112] Mani et al. [99] Matikainen et al. [100] PME [119] Refacola [157] Refactory [131]	ASIMOV [61]	anATLyzer [144] Koschmider et al. [71, 72, 89] RapMOD [90, 91, 109] ReVision [115, 116]

Table 11.8: Public datasets used in the evaluations.

Dataset	URL	Paper
Alps Furniture meta-model	https://backus1.uniandes.edu.co/~enar/dokuwiki/doku.php?id=asimovevaluation	[61]
anATLyzer quick fix website (ATL transform.)	http://sanchezcuadrado.es/projects/anatlyzer/quickfixes.html	[144]
ATL transformations Zoo	https://www.eclipse.org/atl/atlTransformations/	[143]
AtlanMod Ecore Meta-model Zoo	https://web.imt-atlantique.fr/x-info/atlanmod/index.php?title=Ecore	[157]
Eclipse Git repositories	https://git.eclipse.org/c/	[115]
Ecore Metamodels and EcoreBERT Pre-trained Language Model	https://zenodo.org/record/5579980#.ZHRExO9Bzrc	[167]
Extremo website (meta-models)	https://github.com/angel539/extremo/wiki/Performance-Evaluation	[148]
Extremo website (model instances)	https://github.com/angel539/extremo/wiki/Case-Studies	[148]
Google SketchUp (3D models)	https://www.sketchup.com/products/3d-warehouse	[100]
Illinois Semantic Integration Archive	http://pages.cs.wisc.edu/~anhai/wisc-si-archive/	[17]
Matlab Central File Exchange (Simulink files)	https://www.mathworks.com/matlabcentral/fileexchange/	[69]
Model versioning benchmarks	http://www.modelversioning.org/index3899.html?option=com_content&view=article&id=54&Itemid=68	[27]
PARMOREL github (models)	https://github.com/MagMar94/ParmorelRunnable	[74]
PARMOREL website (models)	https://ict.hvl.no/project-parmorel/	[19]
ProB Public Examples Repository	https://www3.hhu.de/stups/downloads/prob/source/	[31]
Refactory website (generic model refactorings)	http://www.modelrefactoring.org	[131]
Simulink Intelligent Modeling Assistant Repository	https://zenodo.org/record/5123570#.ZHTqm-9Bzrcl	[3]
State machine execution contract	http://ecariou.perso.univ-pau.fr/contracts/exec-contract.html	[21]
State machine model and OCL queries	https://github.com/atlanmod/LazyOcl_StateMachineExample	[21]
UML-based Web Engineering (UWE) website	https://uwe.pst.ifi.lmu.de/examples.html	[121, 120]
Yahoo! Pipes	http://www.pipes.digital/pipes	[37]

BIBLIOGRAPHY

[1] S. bin Abid, V. Mahajan, and L. Lucio (2019). "Machine Learning for Learnability of MDD tools". In: *31st International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pp. 355–468.

[2] S. Abid, S. Shamail, H. A. Basit, and S. Nadi (2021). "FACER: An API usage-based code-example recommender for opportunistic reuse". In: *Empir. Softw. Eng.* 26.5, p. 110.

[3] B. Adhikari, E. J. Rapos, and M. Stephan (2023). "SimIMA: A Virtual Simulink Intelligent Modeling Assistant". In: *Software and Systems Modeling*, pp. 1619–1374.

[4] G. Adomavicius and Y. Kwon (2011). "Improving aggregate recommendation diversity using ranking-based techniques". In: *IEEE Transactions on Knowledge and Data Engineering* 24.5, pp. 896–911.

[5] G. Adomavicius and A. Tuzhilin (2005). "Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions". In: *IEEE Transactions on Knowledge and Data Engineering* 17.6, pp. 734–749.

[6] G. Adomavicius and A. Tuzhilin (2011). "Context-aware Recommender Systems". In: *Recommender Systems Handbook*, pp. 217–253.

[7] H. Agt-Rickauer, R. Kutsche, and H. Sack (2018a). "Automated Recommendation of Related Model Elements for Domain Models". In: *6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), Revised Selected Papers*. Vol. 991, pp. 134–158.

[8] H. Agt-Rickauer, R. Kutsche, and H. Sack (2018b). "DoMoRe - A Recommender System for Domain Modeling". In: *6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pp. 71–82.

[9] L. Almonte, E. Guerra, I. Cantador, and J. de Lara (2022a). "Building Recommenders for Modelling Languages with Droid". In: *37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1–4.

[10] L. Almonte, E. Guerra, I. Cantador, and J. de Lara (2022b). "Recommender Systems in Model-Driven Engineering". In: *Softw. Syst. Model.* 21.1, pp. 249–280.

[11] L. Almonte, S. Pérez-Soler, E. Guerra, I. Cantador, and J. de Lara (2021). "Automating the Synthesis of Recommender Systems for Modelling Languages". In: *14th ACM SIGPLAN In-*

ternational Conference on Software Language Engineering (SLE), pp. 22–35.

[12] X. Amatriain, A. Jaimes, N. Oliver, and J. M. Pujol (2011). "Data Mining Methods for Recommender Systems". In: *Recommender Systems Handbook,* 1st *Edition*, pp. 39–71.

[13] F. Anguel, A. Amirat, and N. Bounour (2015). "Hybrid Approach for Metamodel and Model Co-evolution". In: *5th IFIP TC 5 International Conference on Computer Science and its Applications (CIIA)*, pp. 563–573.

[14] E. R. Aquino, P. de Saqui-Sannes, and R. A. Vingerhoeds (2020). "A Methodological Assistant for Use Case Diagrams". In: *8th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pp. 227–236.

[15] N. Arvind, X. Ning, and J. H. Hill (2021). "Using Recommender Systems to Improve Proactive Modeling". In: *Software and Systems Modeling* 20, pp. 1159–1181.

[16] J. A. Aslam and M. H. Montague (2001). "Models for Metasearch". In: *International Conference on Research and Development in Information Retrieval (SIGIR)*, pp. 275–284.

[17] I. Avazpour, J. Grundy, and L. Grunske (2015). "Specifying Model Transformations by Direct Manipulation Using Concrete Visual Notations and Interactive Recommendations". In: *Journal of Visual Languages and Computing* 28, pp. 195–211.

[18] A. Barriga, R. Heldal, A. Rutle, and L. Iovino (2022). "PARMOREL: A Framework for Customizable Model Repair". In: *Softw. Syst. Model.* 21.5, pp. 1739–1762.

[19] A. Barriga, A. Rutle, and R. Heldal (2020). "Improving Model Repair through Experience Sharing". In: *Journal of Object Technology* 19.2, 13:1–21.

[20] F. Basciani, J. Di Rocco, D. Di Ruscio, A. Di Salle, L. Iovino, and A. Pierantonio (2014). "MDEForge: An Extensible Web-Based Modeling Platform". In: *Proceedings CloudMDE@MoDELS*. Vol. 1242, pp. 66–75.

[21] E. Batot, W. Kessentini, H. A. Sahraoui, and M. Famelis (2017). "Heuristic-Based Recommendation for Metamodel - OCL Co-evolution". In: *20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pp. 210–220.

[22] M. Bałchanowski and U. Boryczka (2023). "A Comparative Study of Rank Aggregation Methods in Recommendation Systems". In: *Entropy* 25.1.

[23] L. Bettini (2016). *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd.

[24] S. Bobek, M. Baran, K. Kluza, and G. J. Nalepa (2013). "Application of Bayesian Networks to Recommendations in Business

Process Modeling". In: *Workshop AI Meets Business Processes co-located with AI*IA*. Vol. 1101, pp. 41–50.

[25] M. Brambilla, J. Cabot, and M. Wimmer (2017a). "MDSE Principles". In: *Model-Driven Software Engineering in Practice*, pp. 7–24.

[26] M. Brambilla, J. Cabot, and M. Wimmer (2017b). *Model-Driven Software Engineering in Practice*. 2nd. Morgan and Claypool Publishers.

[27] P. Brosch, M. Seidl, and G. Kappel (2010). "A Recommender for Conflict Resolution Support in Optimistic Model Versioning". In: *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA Companion*, pp. 43–50.

[28] L. Burgueño, R. Clarisó, S. Gérard, S. Li, and J. Cabot (2021). "An NLP-Based Architecture for the Autocompletion of Partial Domain Models". In: *Advanced Information Systems Engineering*, pp. 91–106.

[29] R. Burke (2000). "Knowledge-based Recommender Systems". In: *Encyclopedia of Library and Information Systems* 69.Supplement 32, pp. 175–186.

[30] R. Burke (2002). "Hybrid Recommender Systems: Survey and Experiments". In: *User Modeling and User-adapted Interaction* 12.4, pp. 331–370.

[31] C. Cai, J. Sun, and G. Dobbie (2019). "Automatic B-Model Repair Using Model Checking and Machine Learning". In: *Automated Software Engineering* 26.3, pp. 653–704.

[32] I. Cantador, A. Bellogín, and D. Vallet (2010). "Content-Based Recommendation in Social Tagging Systems". In: *The Fourth ACM Conference on Recommender Systems*, pp. 237–240.

[33] P. Castells, N. J. Hurley, and S. Vargas (2015). "Novelty and Diversity in Recommender Systems". In: *Recommender Systems Handbook*.

[34] P. Castells and D. Jannach (2023). "Recommender Systems: A Primer". In: *CoRR* abs/2302.02579.

[35] G. C. Cawley and N. L. C. Talbot (2010). "On Over-fitting in Model Selection and Subsequent Selection Bias in Performance Evaluation". In: *Journal of Machine Learning Research* 11, pp. 2079–2107.

[36] T. Cerqueira, F. Ramalho, and L. B. Marinho (2016). "A Content-Based Approach for Recommending UML Sequence Diagrams". In: *28th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pp. 644–649.

[37] S. R. Chowdhury, F. Daniel, and F. Casati (2014). "Recommendation and Weaving of Reusable Mashup Model Patterns for Assisted Development". In: *ACM Transactions on Internet Technology* 14.2-3, 21:1–21:23.

[38] R. Clarisó and J. Cabot (2018). "Fixing Defects in Integrity Constraints via Constraint Mutation". In: *11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pp. 74–82.

[39] S. Deng, D. Wang, Y. Li, B. Cao, J. Yin, Z. Wu, and M. Zhou (2017). "A Recommendation System to Facilitate Business Process Modeling". In: *IEEE Transactions on Cybernetics* 47.6, pp. 1380–1394.

[40] A. K. Dey (2001). "Understanding and Using Context". In: *Personal and ubiquitous computing* 5.1, pp. 4–7.

[41] J. Di Rocco, D. Di Ruscio, C. Di Sipio, P. T. Nguyen, and A. Pierantonio (2022a). "MemoRec: A Recommender System For Assisting Modelers In Specifying Metamodels". In: *Software and Systems Modeling* 22, pp. 203–223.

[42] J. Di Rocco, D. Di Ruscio, C. Di Sipio, P. T. Nguyen, and R. Rubei (2021). "Development of recommendation systems for software engineering: the CROSSMINER experience". In: *Empir. Softw. Eng.* 26.4, p. 69.

[43] J. Di Rocco, C. Di Sipio, D. Di Ruscio, and P. T. Nguyen (2021). "A GNN-based Recommender System to Assist the Specification of Metamodels and Models". In: *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pp. 70–81.

[44] J. Di Rocco, C. Di Sipio, P. T. Nguyen, D. Di Ruscio, and A. Pierantonio (2022b). "Finding with NEMO: A Recommender System to Forecast the Next Modeling Operations". In: pp. 154–164.

[45] D. Di Ruscio, D. S. Kolovos, J. de Lara, A. Pierantonio, M. Tisi, and M. Wimmer (2022). "Low-code development and model-driven engineering: Two sides of the same coin?" In: *Softw. Syst. Model.* 21.2, pp. 437–446.

[46] C. Di Sipio, J. Di Rocco, D. Di Ruscio, and P. T. Nguyen (2023). "MORGAN: A Modeling Recommender System Based on Graph Kernel". In: *Softw. Syst. Model.* to appear, pp. 70–81.

[47] C. Di Sipio, J. Di Rocco, D. Di Ruscio, and P. T. Nguyen (2021). "A Low-Code Tool Supporting the Development of Recommender Systems". In: *15th Conference on Recommender Systems (RecSys)*, pp. 741–744.

[48] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett (1999). "Patterns in Property Specifications for Finite-State Verification". In: *21st International Conference on Software Engineering (ICSE)*, pp. 411–420.

[49] A. Dyck, A. Ganser, and H. Lichter (2013). "Enabling Model Recommenders for Command-Enabled Editors". In: *1st International Workshop on Model-driven Engineering By Example (MDEBE@MoDELS)*. Vol. 1104, pp. 12–21.

[50] A. Dyck, A. Ganser, and H. Lichter (2014a). "A Framework for Model Recommenders - Requirements, Architecture and Tool Support". In: *2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pp. 282–290.

[51] A. Dyck, A. Ganser, and H. Lichter (2014b). "On Designing Recommenders for Graphical Domain Modeling Environments". In: *2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pp. 291–299.

[52] A. Elkamel, M. Gzara, and H. Ben-Abdallah (2016). "An UML Class Recommender System for Software Design". In: *13th IEEE/ACS International Conference of Computer Systems and Applications (AICCSA)*, pp. 1–8.

[53] R. Espinosa, D. García-Saiz, M. E. Zorrilla, J. J. Zubcoff, and J. Mazón (2013). "Development of a Knowledge Base for Enabling Non-expert Users to Apply Data Mining Algorithms". In: *3rd International Symposium on Data-driven Process Discovery and Analysis*. Vol. 1027, pp. 46–61.

[54] R. Espinosa, D. García-Saiz, M. E. Zorrilla, J. J. Zubcoff, and J. Mazón (2019). "S3Mining: A Model-driven Engineering Approach for Supporting Novice Data Miners in Selecting Suitable Classifiers". In: *Computer Standards and Interfaces* 65, pp. 143–158.

[55] R. Fagin, R. Kumar, and D. Sivakumar (2003). "Efficient Similarity Search and Classification Via Rank Aggregation". In: *International Conference on Management of Data ()*, pp. 301–312.

[56] M. Farah and D. Vanderpooten (2007). "An Outranking Approach for Rank Aggregation in Information Retrieval". In: *SIGIR 2007: The 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 591–598.

[57] A. Felfernig, L. Boratto, M. Stettinger, and M. Tkali (2018). *Group Recommender Systems: An Introduction*. 1st. Springer Publishing Company, Incorporated.

[58] M. Fellmann, D. Metzger, S. Jannaber, N. Zarvic, and O. Thomas (2018). "Process Modeling Recommender Systems - A Generic Data Model and its Application to a Smart Glasses-based Modeling Environment". In: *Bus. Inf. Syst. Eng.* 60.1, pp. 21–38.

[59] J. L. Fleiss (1971). "Measuring nominal scale agreement among many raters". In: *Psychological Bulletin* 76.5, 378–382.

[60] J. L. Fleiss, B. Levin, M. C. Paik, et al. (1981). "The measurement of interrater agreement". In: *Statistical methods for rates and proportions*, pp. 212–236.

[61] H. Florez, M. E. Sánchez, J. Villalobos, and G. Vega (2012). "Coevolution Assistance for Enterprise Architecture Models". In:

6th International Workshop on Models and Evolution (ME@MoDELS), pp. 27–32.

[62] S. Funk (2006). *Netflix Update: Try This At Home*. URL: http://sifter.org/~simon/journal/20061211.html.

[63] H. Garbe (2012). "Intelligent Assistance in a Problem Solving Environment for UML Class Diagrams by Combining a Generative System with Constraints". In: *eLearning*.

[64] P. Gomes (2004). "Software Design Retrieval Using Bayesian Networks and WordNet". In: *7th European Conf. on Advances in Case-Based Reasoning (ECCBR)*. Vol. 3155, pp. 184–197.

[65] A. Gunawardana and G. Shani (2015). "Evaluating Recommender Systems". In: *Recommender Systems Handbook*, pp. 265–308.

[66] I. Guy (2015). "Social Recommender Systems". In: *Recommender Systems Handbook*, pp. 511–543.

[67] S. Hayashi, P. YiBing, M. Sato, K. Mori, S. Sejeon, and S. Haruna (2004). "Test Driven Development of UML Models with SMART Modeling System". In: *7th International Conference on The Unified Modelling Language: Modelling Languages and Applications (UML)*. Vol. 3273, pp. 395–409.

[68] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua (2017). "Neural Collaborative Filtering". In: *26th International Conference on the World-Wide Web (WWW)*, pp. 173–182.

[69] L. Heinemann (2012). "Facilitating Reuse in Model-based Development with Context-dependent Model Element Recommendations". In: *3rd International Workshop on Recommendation Systems for Software Engineering (RSSE)*, pp. 16–20.

[70] J. A. Hernández López and J. Sánchez Cuadrado (2022). "An Efficient and Scalable Search Engine for Models". In: *Softw. Syst. Model.* 21.5. See also http://mar-search.org/, pp. 1715–1737.

[71] T. Hornung, A. Koschmider, and A. Oberweis (2009). "A Recommender System for Business Process Models". In: *Information Technology & Systems*.

[72] T. Hornung, A. Koschmider, and G. Lausen (2008). "Recommendation Based Process Modeling Support: Method and User Experience". In: *27th International Conference on Conceptual Modeling (ER)*. Vol. 5231, pp. 265–278.

[73] J. Huh, J. C. Grundy, J. G. Hosking, K. N. Li, and R. Amor (2009). "Integrated Data Mapping for a Software Meta-tool". In: *20th Australian Software Engineering Conference (ASWEC)*, pp. 111–120.

[74] L. Iovino, A. Barriga, A. Rutle, and R. Heldal (2020). "Model Repair with Quality-Based Reinforcement Learning". In: *Journal of Object Technology* 19.2, 17:1–21.

[75] A. Isaksson, M. Wallman, H. Göransson, and M. Gustafsson (2008). "Cross-Validation and Bootstrapping are Unreliable in

Small Sample Classification". In: *Pattern Recognition Letters* 29.14, pp. 1960–1965.

[76] D. Jackson (2006). *Software Abstractions - Logic, Language, and Analysis*. http://alloytools.org/. MIT Press.

[77] D. Jannach, M. Zanker, A. Felfernig, and G. Friedrich (2010). *Recommender Systems - An Introduction*. Cambridge University Press.

[78] Y. Ji, A. Sun, J. Zhang, and C. Li (2020). "A Re-visit of the Popularity Baseline in Recommender Systems". In: *43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 1749–1752.

[79] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev (2008). "ATL: A Model Transformation Tool". In: *Science of Computer Programming* 72.1-2, pp. 31–39.

[80] F. Kahloun and S. A. Ghannouchi (2018). "Improvement of Quality for Business Process Modeling Driven by Guidelines". In: *22nd International Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES)*. Vol. 126, pp. 39–48.

[81] H. Kaindl, E. P. Wach, A. Okoli, R. Popp, R. Hoch, W. Gaulke, and T. Hussein (2013). "Semi-Automatic Generation of Recommendation Processes and Their GUIs". In: *The 2013 International Conference on Intelligent User Interfaces*, pp. 85–94.

[82] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson (1990). *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-021. Software Engineering Institute, Carnegie Mellon University.

[83] H. Khider, S. Hammoudi, A. Benna, and A. Meziane (2018). "Social Business Process Model Recommender: An MDE Approach". In: *5th International Conference on Social Networks Analysis, Management and Security (SNAMS)*, pp. 106–113.

[84] H. Khider, S. Hammoudi, and A. 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)*, pp. 65–75.

[85] B. P. Knijnenburg and M. C. Willemsen (2015). "Evaluating Recommender Systems with User Experiments". In: *Recommender Systems Handbook*, pp. 309–352.

[86] S. Kögel (2017). "Recommender System for Model Driven Software Development". In: *11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pp. 1026–1029.

[87] S. Kögel, R. Groner, and M. Tichy (2016). "Automatic Change Recommendation of Models and Meta Models Based on Change Histories". In: *10th Workshop on Models and Evolution (ME@MoDELS)*. Vol. 1706, pp. 14–19.

[88] Y. Koren and R. Bell (2015). "Advances in Collaborative Filtering". In: *Recommender Systems Handbook*, pp. 77–118.

[89] A. Koschmider, T. Hornung, and A. Oberweis (2011). "Recommendation-based Editor for Business Process Modeling". In: *Data & Knowledge Engineering* 70.6, pp. 483–503.

[90] T. Kuschke and P. Mäder (2017). "RapMOD - In Situ Auto-Completion for Graphical Models: Poster". In: *39th International Conference on Software Engineering (ICSE), Companion Volume*, pp. 303–304.

[91] T. Kuschke, P. Mäder, and P. Rempel (2013). "Recommending Auto-completions for Software Modeling Activities". In: *16th International Conference on Model-Driven Engineering Languages and Systems (MoDELS)*. Vol. 8107, pp. 170–186.

[92] J. de Lara, E. Guerra, and J. Sánchez Cuadrado (2013). "Reusable Abstractions for Modeling Languages". In: *Inf. Syst.* 38.8, pp. 1128–1149.

[93] J. de Lara and H. Vangheluwe (2002). "AToM^3: A Tool for Multi-formalism and Meta-modelling". In: *5th International Conference on Fundamental Approaches to Software Engineering (FASE)*. Vol. 2306, pp. 174–188.

[94] Y. Li, B. Cao, L. Xu, J. Yin, S. Deng, Y. Yin, and Z. Wu (2014). "An Efficient Recommendation Method for Improving Business Process Modeling". In: *IEEE Transactions on Industrial Informatics* 10.1, pp. 502–513.

[95] F. Liu, G. Li, Z. Fu, S. Lu, Y. Hao, and Z. Jin (2022). "Learning to Recommend Method Names with Global Context". In: *44th International Conference on Software Engineering (ICSE)*, pp. 1294–1306.

[96] T. Liu (2011). *Learning to Rank for Information Retrieval*. Springer.

[97] P. Lops, M. De Gemmis, and G. Semeraro (2011). "Content-based Recommender Systems: State of the Art and Trends". In: *Recommender Systems Handbook*, pp. 73–105.

[98] P. Mäder, T. Kuschke, and M. Janke (2021). "Reactive Auto-Completion of Modeling Activities". In: *IEEE Trans. Software Eng.* 47.7, pp. 1431–1451.

[99] S. Mani, V. S. Sinha, P. Dhoolia, and S. Sinha (2010). "Automated Support for Repairing Input-model Faults". In: *25th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 195–204.

[100] P. Matikainen, P. M. Furlong, R. Sukthankar, and M. Hebert (2013). "Multi-armed Recommendation Bandits for Selecting State Machine Policies for Robotic Systems". In: *2013 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4545–4551.

[101] S. Mazanek and M. Minas (2009). "Business Process Models as a Showcase for Syntax-Based Assistance in Diagram Edi-

tors". In: *12th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*. Vol. 5795, pp. 322–336.

[102] T. Mens and P. Van Gorp (2006). "A Taxonomy of Model Transformation". In: *Electronic Notes in Theoretical Computer Science* 152. The International Workshop on Graph and Model Transformation (GraMoT 2005), pp. 125–142.

[103] C. Mettouris, A. Achilleos, G. Kapitsaki, and G. A. Papadopoulos (n.d.). "The UbiCARS Model-Driven Framework: Automating Development of Recommender Systems for Commerce". In: pp. 37–53.

[104] S. E. Middleton, D. De Roure, and N. R. Shadbolt (2004). "Ontology-based Recommender Systems". In: *Handbook on Ontologies*, pp. 477–498.

[105] N. Moha, S. Sen, C. Faucher, O. Barais, and J. Jézéquel (2010). "Evaluation of Kermeta for Solving Graph-based Problems". In: *International Journal on Software Tools for Technology Transfer* 12.3-4, pp. 273–285.

[106] F. U. Muram, B. Gallina, and L. G. Rodriguez (2018). "Preventing Omission of Key Evidence Fallacy in Process-Based Argumentations". In: *11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pp. 65–73.

[107] K. Muslu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin (2012). "Speculative Aanalysis of Integrated Development Environment Recommendations". In: *27th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 669–682.

[108] G. Mussbacher et al. (2020). "Opportunities in Intelligent Modeling Assistance". In: *Softw. Syst. Model.* 19.5, pp. 1045–1053.

[109] P. Mäder, T. Kuschke, and M. Janke (2021). "Reactive Auto-Completion of Modeling Activities". In: *IEEE Transactions on Software Engineering* 47.7, pp. 1431–1451.

[110] N. Nassar, H. Radke, and T. Arendt (2017). "Rule-Based Repair of EMF Models: An Automated Interactive Approach". In: *10th International Conference on Theory and Practice of Model Transformation (ICMT)*. Vol. 10374, pp. 171–181.

[111] A. Nechypurenko, E. Wuchner, J. White, and D. C. Schmidt (2006). "Applying Model Intelligence Frameworks for Deployment Problem in Real-Time and Embedded Systems". In: *Models in Software Engineering, Workshops and Symposia at MoDELS'06, Reports and Revised Selected Papers*. Vol. 4364, pp. 143–151.

[112] P. Neubauer, R. Bill, T. Mayerhofer, and M. Wimmer (2017). "Automated Generation of Consistency-achieving Model Editors". In: *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 127–137.

[113] D. Nešić, J. Krüger, u. Stănciulescu, and T. Berger (2019). "Principles of Feature Modeling". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 62–73.

[114] X. Ning, C. Desrosiers, and G. Karypis (2015). "A Comprehensive Survey of Neighborhood-based Recommendation Methods". In: *Recommender Systems Handbook*, pp. 37–76.

[115] M. Ohrndorf, C. Pietsch, U. Kelter, L. Grunske, and T. Kehrer (2021). "History-Based Model Repair Recommendations". In: *ACM Trans. Softw. Eng. Methodol.* 30.2.

[116] M. Ohrndorf, C. Pietsch, U. Kelter, and T. Kehrer (2018). "ReVision: A Tool for History-based Model Repair Recommendations". In: *40th International Conference on Software Engineering (ICSE), Companion Proceeedings*, pp. 105–108.

[117] M. C. D. Oliveira, D. Freitas, R. Bonifácio, G. Pinto, and D. Lo (2019). "Finding needles in a haystack: Leveraging co-change dependencies to recommend refactorings". In: *Journal of Systems and Software* 158.

[118] S. E. L. Oliveira, V. Diniz, A. Lacerda, L. H. C. Merschmann, and G. L. Pappa (2020). "Is Rank Aggregation Effective in Recommender Systems? An Experimental Analysis". In: *ACM Trans. Intell. Syst. Technol.* 11.2, 16:1–16:26.

[119] T. Pati, S. Kolli, and J. H. Hill (2017). "Proactive Modeling: a New Model Intelligence Technique". In: *Software and Systems Modeling* 16.2, pp. 499–521.

[120] S. Paydar and M. Kahani (2015a). "A Semantic Web Enabled Approach to Reuse Functional Requirements Models in Web Engineering". In: *Automated Software Engineering* 22.2, pp. 241–288.

[121] S. Paydar and M. Kahani (2015b). "A Semi-automated Approach to Adapt Activity Diagrams for New Use Cases". In: *Inf. Softw. Technol.* 57, pp. 543–570.

[122] M. J. Pazzani (1999). "A Framework for Collaborative, Content-based and Demographic Filtering". In: *Artificial Intelligence Review* 13.5-6, pp. 393–408.

[123] S. Pérez-Soler, E. Guerra, and J. de Lara (2018). "Collaborative Modeling and Group Decision Making Using Chatbots in Social Networks". In: *IEEE Softw.* 35.6, pp. 48–54.

[124] A. Pescador and J. de Lara (2016). "DSL-maps: From Requirements to Design of Domain-specific Languages". In: *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 438–443.

[125] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson (2008). "Systematic Mapping Studies in Software Engineering". In: *12th*

International Conference on Evaluation and Assessment in Software Engineering, EASE.

[126] K. Petersen, S. Vakkalanka, and L. Kuzniarz (2015). "Guidelines for Conducting Systematic Mapping Studies in Software Engineering: An Update". In: *Information and Software Technology* 64, pp. 1–18.

[127] M. Quadrana, P. Cremonesi, and D. Jannach (2018). "Sequence-Aware Recommender Systems". In: 51.4.

[128] F. Rabbi, Y. Lamo, I. C. Yu, and L. M. Kristensen (2015). "A Diagrammatic Approach to Model Completion". In: *4th Workshop on the Analysis of Model Transformations (AMT@MoDELS)*. Vol. 1500, pp. 56–65.

[129] M. E. Rangiha, M. Comuzzi, and B. Karakostas (2015). "Role and Task Recommendation and Social Tagging to Enable Social Business Process Management". In: *BPMDS/EMMSAD@CAiSE*. Vol. 214, pp. 68–82.

[130] S. Raza and C. Ding (2019). "Progress in context-aware recommender systems—An overview". In: *Computer Science Review* 31, pp. 84–97.

[131] J. Reimann, M. Seifert, and U. Aßmann (2013). "On the Reuse and Recommendation of Model Refactoring Specifications". In: *Software and Systems Modeling* 12.3, pp. 579–596.

[132] Z. Reitermanová (2010). "Data Splitting". In: *WDS'10 Proceedings of Contributed Papers, Part I*, pp. 31–36.

[133] F. Ricci, L. Rokach, and B. Shapira (2022). *Recommender Systems Handbook*. 3rd ed. Springer US.

[134] M. P. Robillard and R. J. Walker (2014). "An Introduction to Recommendation Systems in Software Engineering". In: *Recommendation Systems in Software Engineering*, pp. 1–11.

[135] M. P. Robillard, R. J. Walker, and T. Zimmermann (2010). "Recommendation systems for Software Engineering". In: *IEEE Software* 27.4, pp. 80–86.

[136] G. Rojas, F. Domínguez, and S. Salvatori (2009). "Recommender Systems on the Web: A Model-Driven Approach". In: *10th International Conference on E-Commerce and Web Technologies (ECWeb)*. Vol. 5692, pp. 252–263.

[137] G. Rojas and C. Uribe (2013). "A Conceptual Framework to Develop Mobile Recommender Systems of Points of Interest". In: *SCCC*, pp. 16–20.

[138] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. Polack (2008). "The Epsilon Generation Language". In: *4th European Conf. on Model Driven Architecture - Foundations and Applications (ECMDA-FA)*. Vol. 5095, pp. 1–16.

[139] A. Said and A. Bellogín (2014). "Rival: A Toolkit to Foster Reproducibility in Recommender System Evaluation". In: *8th*

ACM Conference on Recommender Systems (RecSys). See also https://github.com/recommenders/rival, pp. 371–372.

[140] R. Saini, G. Mussbacher, J. L. C. Guo, and J. Kienzle (2019). "Teaching Modelling Literacy: An Artificial Intelligence Approach". In: *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS), Companion Proceedings*, pp. 714–719.

[141] J. Sánchez Cuadrado, E. Guerra, and J. de Lara (2012). "Flexible Model-to-Model Transformation Templates: An Application to ATL". In: *J. Object Technol.* 11.2, 4: 1–28.

[142] J. Sánchez Cuadrado, E. Guerra, and J. de Lara (2015). "Quick fixing ATL Model Transformations". In: *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pp. 146–155.

[143] J. Sánchez Cuadrado, E. Guerra, and J. de Lara (2018a). "AnATLyzer: An Advanced IDE for ATL Model Transformations". In: *40th International Conference on Software Engineering (ICSE), Companion Proceedings*, pp. 85–88.

[144] J. Sánchez Cuadrado, E. Guerra, and J. de Lara (2018b). "Quick fixing ATL Transformations with Speculative Analysis". In: *Software and Systems Modeling* 17.3, pp. 779–813.

[145] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl (2001). "Item-Based Collaborative Filtering Recommendation Algorithms". In: *10th International Conference on the World-Wide Web (WWW)*, pp. 285–295.

[146] M. Savary-Leblanc (2019). "Improving MBSE Tools UX with AI-Empowered Software Assistants". In: *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS), Companion Volume*, pp. 648–652.

[147] M. Savary-Leblanc, X. Le-Pallec, and S. Gérard (2021). "A Modeling Assistant for Cognifying MBSE Tools". In: *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pp. 630–634.

[148] Á. M. Segura and J. de Lara (2019). "Extremo: An Eclipse Plugin for Modelling and Meta-modelling Assistance". In: *Science of Computer Programming* 180, pp. 71–80.

[149] Á. M. Segura, J. de Lara, P. Neubauer, and M. Wimmer (2018). "Automated Modelling Assistance by Integrating Heterogeneous Information Sources". In: *Computer Languages, Systems and Structures* 53, pp. 90–120.

[150] Á. M. Segura, A. Pescador, J. de Lara, and M. Wimmer (2016). "An Extensible Meta-Modelling Assistant". In: *20th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, pp. 1–10.

[151] S. Sen, B. Baudry, and H. Vangheluwe (2007). "Domain-Specific Model Editors with Model Completion". In: *Models in Software*

Engineering, Workshops and Symposia at MoDELS'07, Reports and Revised Selected Papers. Vol. 5002, pp. 259–270.

[152] S. Sen, B. Baudry, and H. Vangheluwe (2010). "Towards Domain-specific Model Editors with Automatic Model Completion". In: *Simulation* 86.2, pp. 109–126.

[153] F. F. Shahare (2017). "Sentiment Analysis for the News Data Based on the Social Media". In: *2017 International Conference on Intelligent Computing and Control Systems (ICICCS)*, pp. 1365–1370.

[154] N. Shilov, W. Othman, M. Fellmann, and K. Sandkuhl (2023). "Machine Learning for Enterprise Modeling Assistance: An Investigation of the Potential and Proof of Concept". In: *Software and Systems Modeling* 22, pp. 619–646.

[155] T. Silveira, M. Zhang, X. Lin, Y. Liu, and S. Ma (2019). "How good your recommender system is? A survey on evaluations in recommendation". In: *International Journal of Machine Learning and Cybernetics* 10, pp. 813–831.

[156] T. Stahl, M. Voelter, and K. Czarnecki (2006). *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, Inc.

[157] F. Steimann and B. Ulke (2013). "Generic Model Assist". In: *16th International Conference on Model-Driven Engineering Languages and Systems (MoDELS)*. Vol. 8107, pp. 18–34.

[158] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks (n.d.). *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional.

[159] M. 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)*, pp. 21–24.

[160] C. Tinnes, T. Kehrer, M. Joblin, U. Hohenstein, A. Biesdorf, and S. Apel (2022). "Learning Domain-Specific Edit Operations from Model Repositories with Frequent Subgraph Mining". In: *The 36th IEEE/ACM International Conference on Automated Software Engineering*, pp. 930–942.

[161] N. Tintarev and J. Masthoff (2012). "Evaluating the Effectiveness of Explanations for Recommender Systems". In: *User Modeling and User-Adapted Interaction* 22.4-5, pp. 399–439.

[162] I. Tobías (2017). "Matrix Factorization Models for Cross-domain Recommendation: Addressing the Cold Start in Collaborative Filtering". In:

[163] S. Tuarob, N. Assavakamhaenghan, W. Tanaphantaruk, P. Suwanworaboon, S. Hassan, and M. Choetkiertikul (2021). "Automatic team recommendation for collaborative software development". In: *Empir. Softw. Eng.* 26.4, p. 64.

[164] J. A. Vargas Muñoz, R. da Silva Torres, and M. A. Gonçalves (2015). "A Soft Computing Approach for Learning to Aggregate Rankings". In: *The 24th ACM International Conference on Information and Knowledge Management, CIKM*, pp. 83–92.

[165] S. Vargas and P. Castells (2011). "Rank and Relevance in Novelty and Diversity Metrics for Recommender Systems". In: *5th ACM Conference on Recommender Systems (RecSys)*. See also http://ranksys.github.io/, pp. 109–116.

[166] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. Kats, E. Visser, and G. Wachsmuth (2013). *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org.

[167] M. Weyssow, H. Sahraoui, and E. Syriani (2022). "Recommending Metamodel Concepts during Modeling Activities with Pre-Trained Language Models". In: *Softw. Syst. Model.* 21.3, pp. 1071–1089.

[168] S. Witt, S. Feja, A. Speck, and C. Hadler (2014). "Business Application Modeler: A Process Model Validation and Verification Tool". In: *IEEE 22nd International Requirements Engineering Conference (RE)*, pp. 333–334.

[169] E. Zangerle and C. Bauer (2023). "Evaluating Recommender Systems: Survey and Framework". In: *ACM Comput. Surv.* 55.8, 170:1–170:38.