Reuse and Automated Integration of Recommenders for Modelling Languages

Anonymous Author(s)*

Abstract

Many recommenders for modelling tasks have recently appeared. They use a variety of recommendation methods, tailored to concrete modelling languages. Typically, recommenders are created as independent programs, and subsequently need to be integrated within a modelling tool, incurring in high development effort. Moreover, it is currently not possible to reuse a recommender created for a modelling language with a different notation, even if they are similar.

To alleviate these problems, we propose a methodology to reuse recommenders and integrate them into modelling tools. It considers four orthogonal dimensions: the target modelling language, the modelling tool, the recommendation source, and the recommended items. To make the access to arbitrary recommenders homogeneous, we propose a reference recommendation service that enables indexing recommenders, investigating their properties, and obtaining recommendations likely coming from several sources. Our methodology is supported by IronMan, an Eclipse plugin that automates the integration of recommenders within Sirius and tree-based modelling editors, and can bridge recommenders created for a modelling language for their reuse with a different one. We evaluate the power of the tool by reusing 2 recommenders for 4 different modelling languages, and integrating them into 6 existing modelling tools.

CCS Concepts: • **Software and its engineering → Integrated and visual development environments**; • **Information systems → Recommender systems**.

Keywords: Model-driven engineering, recommender systems, language engineering, modelling tools

ACM Reference Format:

Anonymous Author(s). 2023. Reuse and Automated Integration of Recommenders for Modelling Languages. In *Proceedings of International Conference on Software Language Engineering (SLE'23).* ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SLE'23, October 22–27, 2023, Cascais, Portugal

© 2023 Association for Computing Machinery.

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Recommender systems (RSs) are increasingly being used to assist developers in all sorts of software engineering tasks [49]. Modelling is no exception, as we are recently witnessing the proposal of numerous recommenders for modelling languages [3]. Most of them help in creating models or meta-models by recommending, e.g., new attributes or references for classes, or new classes related to existing ones [2, 10, 16, 19, 53, 59]. They use a variety of methods – each with their own strengths and weaknesses – ranging from classical recommendation algorithms like collaborative filtering [2, 16] or content-based recommendations [2], to knowledge graphs [53], natural language processing [10], pre-trained language models [59] or graph kernels [19].

Given this growing plethora of modelling recommenders, the natural question is "*Can I reuse these RSs for my modelling notation, and integrate them within my modelling tool?*". However, the reuse and integration of RSs pose a number of practical challenges. Firstly, existing RSs may have been developed for a different (albeit perhaps similar) modelling language, such as an existing RS for Ecore models that one may like to reuse for UML class diagrams. Moreover, it can be useful to combine several RSs because they suggest different types of items (e.g., attributes, operations) for different target elements (e.g., classes, interfaces). Further, 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 need to be integrated within heterogeneous modelling tools (e.g., graphical, textual, tree-based).

In this paper, we address the challenges of integrating and reusing RSs for modelling languages. To accomplish this goal, we propose deploying the RSs as services, on the basis of a standard API and a recommender server protocol. This also facilitates the recommenders' integration within arbitrary modelling tools. In addition, we enable the reuse of RSs tailored to a modelling language for other notations via a structural mapping. The combination of recommenders of the same type of item (e.g., attributes) relies on mechanisms for the *aggregation* of their recommendation lists [44], and our approach is flexible to accommodate several aggregation methods. Technically, we provide support for the automated integration of the assembled RSs within Eclipse modelling editors based on Sirius [55], and EMF tree editors [57].

Our approach is realised as an Eclipse plugin called IronMan (Integrating RecOmmeNders for Modelling lANguages),

which guides in all steps of the integration task, including RS discovery and selection, adaptation to the modelling language (if needed), configuration of the aggregation method, and integration of the recommender within the (Sirius- or tree-based) modelling tool. To evaluate its usefulness, we assess the reuse and integration of two existing RSs into six third-party tools of the Eclipse ecosystem.

Paper organisation. Sec. 2 provides background on RSs for modelling languages and analyses the relevant dimensions for their reuse and integration. Sec. 3 presents the components of our approach: the reference recommendation service and its protocol, the adaptation of the RSs to the modelling language, the recommendation aggregation mechanism, and the integration of the RSs into modelling tools. Sec. 4 describes our tool and Sec. 5 reports on its evaluation. Sec. 6 compares with related work and Sec. 7 concludes.

2 Background and Integration Dimensions

Next, we overview RSs for modelling languages (Sec. 2.1) and present the dimensions for their integration (Sec. 2.2).

2.1 Background on Recommender Systems

RSs have become ubiquitous software tools that assist in decision-making tasks in situations of information overload. They are key components of a wide range of applications, including e-commerce sites (e.g., Amazon), social networks (e.g., Facebook), and music (e.g., Spotify), video (e.g., Netflix) and streaming platforms (e.g., Twitch) [48].

RSs suggest items that align with the preferences of a particular user. The term *item* refers to what is suggested to the user. RSs usually focus on a particular type of item (e.g., videos), using filtering and ranking algorithms to provide valuable recommendations for that item type. The recommendations are computed based on data about three entities: *target users*, *items*, and *user-item interactions* (often unary or numeric ratings) that express personal preferences [48]. When applied to modelling tasks, these entities are sometimes reinterpreted. As an example, in a RS suggesting attributes for classes, the recommended items are the *attributes*, the target users are the *classes*, and the user-item interactions are given by the inclusion of the attributes in each class and its superclasses. To avoid confusion, we use the term *target* to refer to the target users (classes in this example).

RSs can be classified into three main categories based on how they compute the recommendations: *content-based* systems recommend items similar to the ones that the user preferred in the past; *collaborative filtering* systems recommend items preferred by like-minded users; and *hybrid* systems combine the previous two techniques to overcome their limitations. The three approaches return a list of recommended items, which is often *ranked*. In addition, some recommendation methods provide a *rating* for each item, which quantifies the likelihood of the item to be relevant for the user.

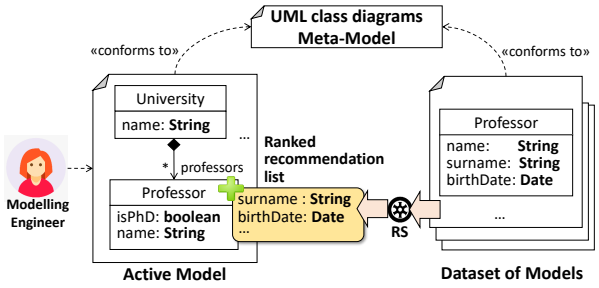


Figure 1. Working scheme of a modelling recommender.

Fig. 1 shows the working scheme of a RS for UML class diagrams. A RS for a modelling language is typically built on the basis of a dataset of models conformant to the language meta-model. Then, when a modelling engineer is working on a model conformant to the same meta-model, the RS can provide sensible recommendations. In the figure, the RS suggests new attributes to incorporate to a given class.

2.2 Dimensions of Integration of Modelling RSs

The integration and reuse of RSs for modelling tasks requires the consideration of several dimensions, summarised in Fig. 2 as a feature model [29].

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 example, a RS for meta-modelling languages like Ecore [57] may be reused for UML class diagrams, and vice versa [4]. 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. We describe our approach to adapt recommenders in Sec. 3.3.

Recommendation sources. The recommendations may be produced *locally*, if the RS is deployed on the computer where the modelling tool is running [10, 16, 19, 59]. In addition, recommendations may come from services deployed on a *remote* server [4, 53]. The latter option is more flexible, as it permits reusing recommenders within different tools, and aggregating recommendations from several sources. We propose a reference recommendation service in Sec. 3.2 for this purpose.

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

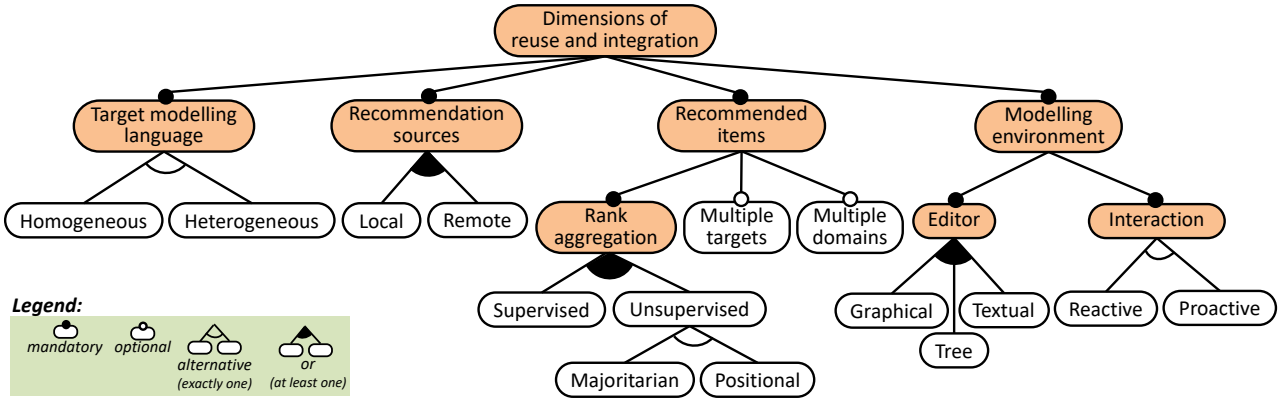


Figure 2. Dimensions of reuse and integration of RSs for modelling languages.

be aggregated. According to [44], the approaches to recommendation *rank aggregation* are broadly classified into *unsupervised* and *supervised*. The former can be further divided into *majoritarian* and *positional*. When calculating 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 [44]. Unsupervised methods are generally simple, efficient and flexible. However, if ground truth data are available, supervised methods may be more effective. These methods may use a variety of techniques, like learning to rank [37] or genetic programming [58]. Sec. 3.4 details some of these methods, recasted for modelling tasks.

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 [38]). Sec. 3.5 explains our approach to integrate RSs into graphical and tree modelling editors.

3 Approach

This section presents our proposal to reuse and integrate RSs for modelling. First, Sec. 3.1 provides an overview. Then, Sec. 3.2 describes the recommendation service. Sec. 3.3 explains our approach to bridge RSs to modelling notations. Sec. 3.4 recasts existing aggregation methods for ranked recommendations to modelling RSs. Finally, Sec. 3.5 introduces our support for integrating RSs into modelling environments.

3.1 Overview

Fig. 3 shows the scheme of the methodology we have created for RS integration and reuse. It covers all dimensions of integration depicted in Fig. 2.

Our approach relies on deploying the RSs as services conformant to the reference REST API described in Section 3.2.

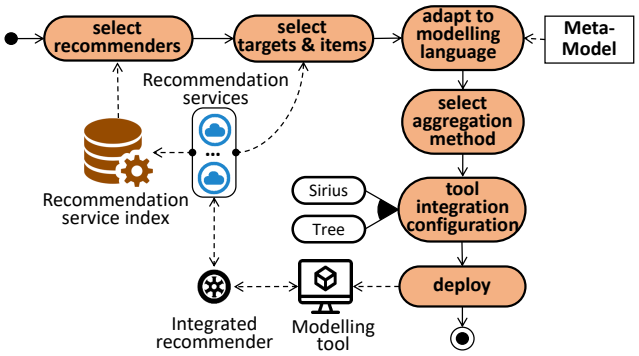


Figure 3. Overview of our methodology for RS integration.

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 user 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 user will have to adapt the RSs via a mapping. As the last step, the user configures the integration with the modelling tool. Currently, we support the adaptation of EMF-based modelling languages, and the integration with Sirius graphical editors [55] and tree editors. However, our extensible architecture facilitates future integration with other technologies, like Xtext [7, 60].

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.

Table 1. Endpoints of the recommender indexer API.

Endpoint	`/register?urlName=`⟨*url*⟩
Desc.	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`
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).

3.2 Recommendation service

We have developed a recommendation service consisting of two components: the recommendation service indexer API, and the recommendation service API. On the one hand, the *indexer* 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. On the other hand, the *recommendation* service offers a uniform mechanism for *requesting recommendations* and accessing the *features* of the RSs registered in the indexer. This approach simplifies the integration of arbitrary RSs into modelling tools, avoiding the need to build custom, heterogeneous integrations.

Table 1 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 2. In particular, upon registering a recommendation service, the indexer invokes its `/features` endpoint (explained below) 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,

Table 2. Endpoints of the recommendation service API.

Endpoint	`/features`
Desc.	Returns the features of all RSs within the recommendation service.
Method	GET
Output	Features of the recommendation service (JSON).
Endpoint	`/recommend/`⟨*name*⟩
Desc.	Returns a list of recommendations for a given target (within a context, if required).
Method	GET
Body	Target and its context, if required (JSON).
Param 1	`newMaxRec` (integer) Maximum number of recommendations to retrieve
Param 2	`threshold` (double) Threshold for the ranking
Param 3	`itemType` ([string]) Type of recommended items
Output	List of recommendations (JSON).

and `/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 [57]. This way, the API returns a JSON list with all recommenders with the given name or defined over a meta-model with the provided *nsURI*.

Table 2 shows the endpoints of the recommendation service. They allow accessing the features of the registered services and requesting recommendations.

The `/features` endpoint allows clients to retrieve the metadata of all RSs within the service. Fig. 4 shows a conceptual model of the metadata. 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 containing the target element), or `Targets` (requires all objects with the same type as the target element).

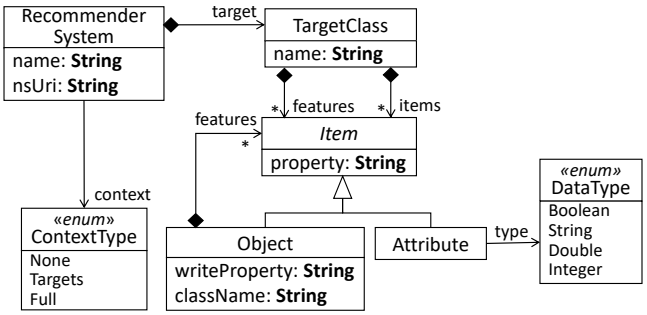
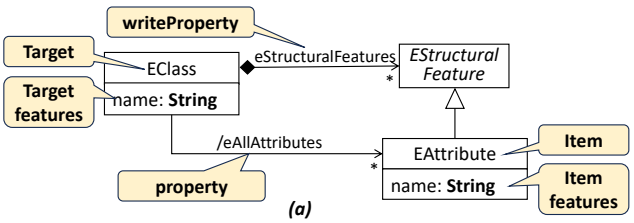


Figure 4. Conceptual model of RSs assumed by our approach.

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) for a target type (e.g., UML classes). If several target types are supported, then one RS for each target needs




```
"services": [
  { "name": "EcoreRecommender",
    "details": [
      { "nsURI"   : "http://www.eclipse.org/emf/2002/Ecore",
        "context": "Targets",
        "target": { "name"    : "EClass",
                    "features": [ {"property" : "name", "type" : "String" }]},
        "items":   [ { "name"           : "EAttribute"
                       "property"       : "eAllAttributes",
                       "writeProperty" : "eStructuralFeatures",
                       "features"       : [ { "property" : "name",
                                              "type" : "String" } ] }
      }]
  }]
```

(b)

Figure 5. (a) Excerpt of the Ecore meta-model, annotated with the role of the elements in the example RS. (b) Encoding excerpt of the RS returned by the `/features` endpoint.

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 object to the target class) 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, `property` in the figure), and the reference where to store an item (i.e., enabling writing the item, `writeProperty` in the figure). In EMF, the latter are containment references. Next, we use an example to illustrate the difference between both.

Example. Let's assume a RS for the Ecore meta-modelling language, which recommends attributes for classes. Fig. 5(a) shows an excerpt of the Ecore meta-model with the relevant parts for the RS[1]. The figure identifies that `EClass` is the target element, that the feature identifying `EClasses` is their `name`, that the recommended items are of type `EAttribute`, and that the feature provided when recommending an attribute is its `name`. In addition, `EAttributes` are *read* via the `eAllAttributes` derived reference, but they are written on the `eStructuralFeatures` composition reference. The former contains all attributes owned and inherited by the class, and the latter only contains the owned ones (and is a common container for both references and attributes). The rationale for distinguishing both is that modelling tools need to provide the items that any target object already has – owned and inherited attributes in our example, available via `eAllAttributes`. However, when a recommendation is accepted, the item needs to be created and assigned to the target – in our example, `eStructuralFeatures` is used for this purpose.

[1]The meta-model is slightly modified to ease understanding.

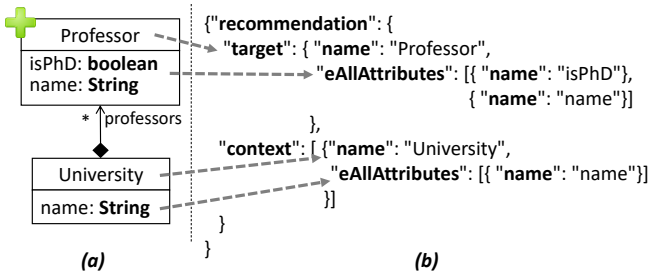


Figure 6. JSON representation for a `/recommend` request.

Fig. 5(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 *EcoreRecommender*, and the RS needs to receive all other possible targets in the model (i.e., all `EClasses`) as context.

Some well-formedness criteria are required from the roles that meta-model elements can play in a RS. In particular, if the recommended items are of type c_i, then c_i should be the destination class of a write property p_w, or a subclass of such destination class: $c_i \leq dest(p_w)$. For instance, in Fig. 5(a), `EAttribute`≤`EStructuralFeature`, which is the destination of the write property `eStructuralFeatures`. This ensures compatibility of the items with the write property, so that newly created items can be inserted in it. Conversely, the destination of a read property p_r should be the type c_i of the recommended item, or a subclass: $dest(p_r) \leq c_i$. In the example, `EAttribute`≤`EAttribute`, which is the destination of the read property `eAllAttributes`. This ensures that the content of the read property is compatible with the item.

The last endpoint in Table 2 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). Clients can customise the recommendation by means of optional query parameters such as the maximum number of recommendations to retrieve (*newMaxRec*), the minimum ranking value threshold (*threshold*), and the desired item type when several are possible (*itemType*).

Example. Fig. 6 shows a recommendation request example for the RS in Fig 5. Part (a) shows an Ecore model being edited, for which the user solicits recommendation for class *Professor*. Part (b) shows the encoding of the request. In this request, the target `EClass` is named *Professor*, and has two `EAttributes` called *isPhD* and *name*. They are encoded in the *read* feature `eAllAttributes`. As specified in Fig. 5, the only feature that identifies `EAttributes` is their *name*. Since the context of the RS is set to `Targets`, the recommendation request needs to include the name and attributes of all other `EClasses` in the model. In this case, there is just one additional class named *University*.

3.3 Adaptation of RSs to the modelling notation

Our approach to reuse a RS for a modelling notation involves establishing a structure-preserving mapping $m : RS \longrightarrow MM$ between the classes and features used by the RS, and the elements of interest in the language meta-model MM.

Example. Fig. 7 exemplifies a mapping that adapts the RS for Ecore – which recommends `EAttributes` for `EClasses` – to UML. The adapted RS will then recommend properties for UML classes. *RS* on the left shows an excerpt of the Ecore meta-model containing the elements designated as targets, features and items. The mapping maps the Ecore meta-model elements playing a role in the RS to corresponding elements in the UML meta-model *MM* to the right.

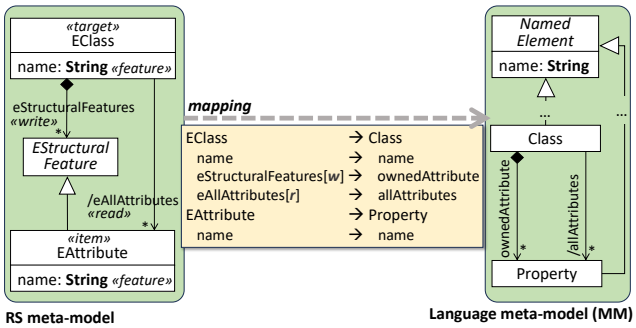


Figure 7. Adapting the RS to the modelling language.

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 [14, 52]. 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 $c_i \leq c_j$ to denote that c_i is compatible with c_j (a subclass, or c_j itself).

This way, a mapping $m : RS \longrightarrow MM$ 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 $c_1 \leq c_2 \iff m(c_1) \leq m(c_2)$.

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 $\leq m(a)$.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 $m(c_1) \leq c_1'$. 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':
$\forall c_i \in RS \cdot dest(r) \leq c_i \wedge relevant(c_i) \implies c_2' \leq m(c_i)$

$writeProperty(r)$**:** Any relevant class compatible with $dest(r)$ is mapped to a class compatible with c_2':
$\forall c_i \in RS \cdot c_i \leq dest(r) \wedge relevant(c_i) \implies m(c_i) \leq c_2'$

Composition is preserved: If r is a relevant write composition in *RS*, then $m(r)$ is also a composition.

The condition for references permits 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.

Example. The mapping of Fig. 7 is well-formed. This is so as both `EClass` and `EAttribute` are mapped to classes in the UML meta-model (`Class` and `Property`), and their attributes (`EClass.name` and `EAttribute.name`) are mapped to attributes of the target classes (actually inherited). Both references `eStructuralFeatures` and `eAllAttributes` are mapped according to the conditions, e.g.,: m(`eStructuralFeatures`) =`ownedAttribute`, the source of both references coincide (m(`EClass`) =`Class`), and for the destination, m(`EAttribute`) =`Property`, which is exactly $dest(m$(`eStructuralFeature`)), but could be a subclass. Reference `eStructuralFeatures` is a *write* feature, and a composition, and so is `ownedAttribute`.

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 [27] – able to calculate derived elements in the target, or adapt attribute values. This is left for future work.

3.4 Recommendation aggregation

The RSs community has proposed different ranked item aggregation methods to combine recommendations from different RSs. A *rank aggregation method* aims to find the best permutation of recommendation lists based on an evaluation metric, such as precision. These methods can be used to provide more accurate and diverse suggestions by taking into account weaknesses or biases that specific recommenders

may have, and to reduce the impact of items incorrectly ranked in high positions by an individual recommender [44].

When combining RSs for modelling languages, two scenarios can arise. In the first one, the RSs to combine tackle *different targets* (e.g., one RS provides recommendations for classes and another one for packages) or *different kinds of items* (e.g., one 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 algorithms, or different datasets). This scenario requires rank aggregation methods to obtain a consensus ranking containing a subset of their items.

Aggregation methods can be either supervised or unsupervised [44]. The former search for the aggregated ranking that optimises a given metric computed over ground-truth data. The latter lack ground truth data and rely on metrics computed using the available rankings of items. We focus on unsupervised methods, and consider score-based positional methods, as they are very popular due to their simplicity and efficiency. These methods sort the items based on their absolute position in the individual rankings. Positional methods receive as input a set of individual rankings, and use an aggregation function $f : U \times I \rightarrow R$ and a procedure to combine the item position-based scores, with U and I the sets of users and items in the system, respectively [44].

Unsupervised positional methods, such as Borda Count (BC) [6] and Median Rank Aggregation (MRA) [24], are popular for their simplicity and efficiency. Borda Count assigns points to items based on their rank, while MRA ranks items by their median position across the individual rankings.

Fig. 8 illustrates the BC method for the aggregation of three hypothetical class attribute recommenders. ① shows the rankings of attributes that each RS suggests for a class named *Person*. 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 each RS do not recommend (e.g., *surname* in Ranking 1) receive equal portions of the remaining available points from the RS. ③ displays the aggregated score of each item, calculated as the sum of individual scores in the case of BC. 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.

3.5 Integration within modelling environments

The last step of the integration is to embed the RS into a modelling environment. This embedding will be different

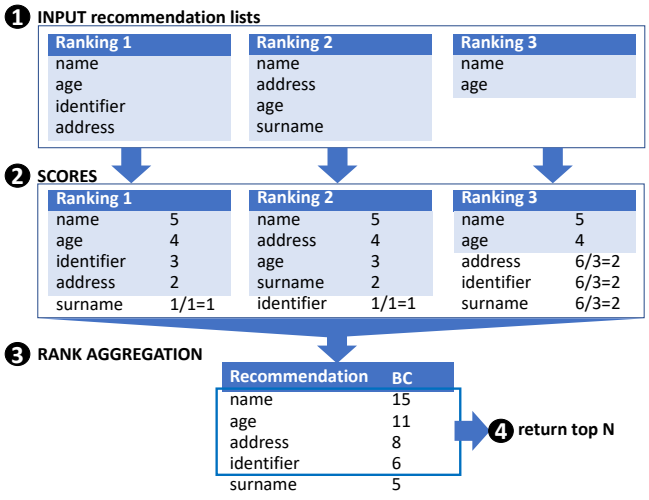


Figure 8. Rank aggregation example using Borda Count.

depending on the concrete syntax of the language. We currently consider two types: graphical syntaxes and tree-based ones. In both cases, we support a *reactive* approach by now, where the user needs to invoke the RS explicitly.

For graphical syntaxes, the integration adds an additional graphical layer in the modelling editor, which enables the option to invoke the RS when a shape corresponding to an instance object of the target class is selected. For tree-based syntaxes, a menu option becomes available when an instance object of the target class is right-clicked. In both cases, recommendations are requested to the recommender API of the selected RSs. Then, the recommended items can be applied to the model, assigned to the selected target object.

Sec. 4.2 will provide more details of this integration for the technologies we support (Sirius and EMF).

4 Architecture and Tool Support

We have realised the previous concepts on an extensible Eclipse plugin called IronMan. Its source code is available at: https://anonymous.4open.science/r/integrate-recommenders-ironman-05C9/. Next, Sec. 4.1 describes its architecture, and Sec. 4.2 reports on the tool itself.

4.1 Architecture

Fig. 9 shows a schema of the architecture of IronMan. The plugin uses the `/services` and `/discover` endpoints 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 by enabling the definition of a structural mapping between both meta-models, as described in Section 3.3.

IronMan provides two extension points. The first one is to define rank aggregation methods. IronMan currently supports Borda Count, MRA and Outranking [25], but the extension point permits adding more. The second one is

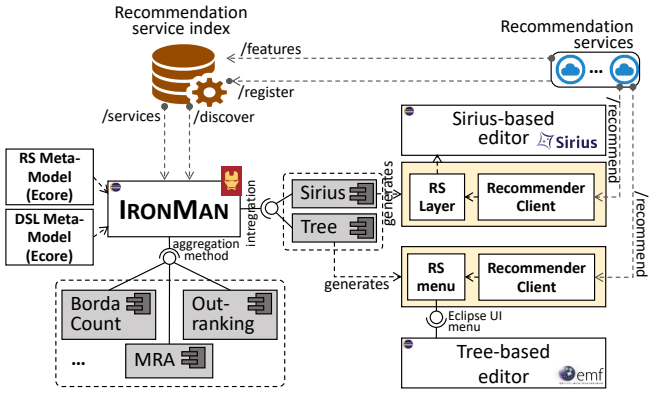


Figure 9. Architecture of IronMan.

to define code generators that can integrate the RSs with modelling tools. Currently, two implementations can generate integrations of RSs with Sirius and tree editors. For this purpose, the generated code makes use of the `/recommend` endpoint of the chosen recommendation service, and the selected aggregation method (if needed). 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 appropriate objects are selected in the tree.

4.2 Tool support

Next, we describe the parts of our solution: the plugin, the services and the generated RS clients.

4.2.1 IronMan plugin. Our tool provides a wizard to adapt RSs to a modelling language, configure the aggregation of recommendations for the same target (if needed), and integrate RSs into modelling workbenches. Fig. 10 shows 5 pages of the wizard. The first one permits selecting the available RSs from a set of recommendation service indexers. New indexers 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 indexers contain several RSs for UML and Ecore.

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, in case 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.

The fourth page is enabled only when the user selects 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. In case of Sirius, the user needs to select the viewpoint where the recommendation layer is to be inserted. The figure shows the selection dialog, where the user can select several views. 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.

4.2.2 Recommendation services. The IronMan service indexer is realised as a Java-based REST service implemented using Jersey [22], a framework for building RESTful web services and APIs. It is deployed on Tomcat [5], 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.

4.2.3 Integration with client modelling tools. The IronMan code generator synthesises code that extends externally existing (Sirius and tree) modelling editors. The generated code takes into account the defined mapping. It uses the EMF reflective API to query the relevant features of the target object of the recommendations, and to create objects corresponding to recommended items when the user applies a recommendation.

Fig. 11 shows a screenshot of the result of the integration of a RS within the Eclipse plugin of Obeo's UML Designer [41]. 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 these 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.

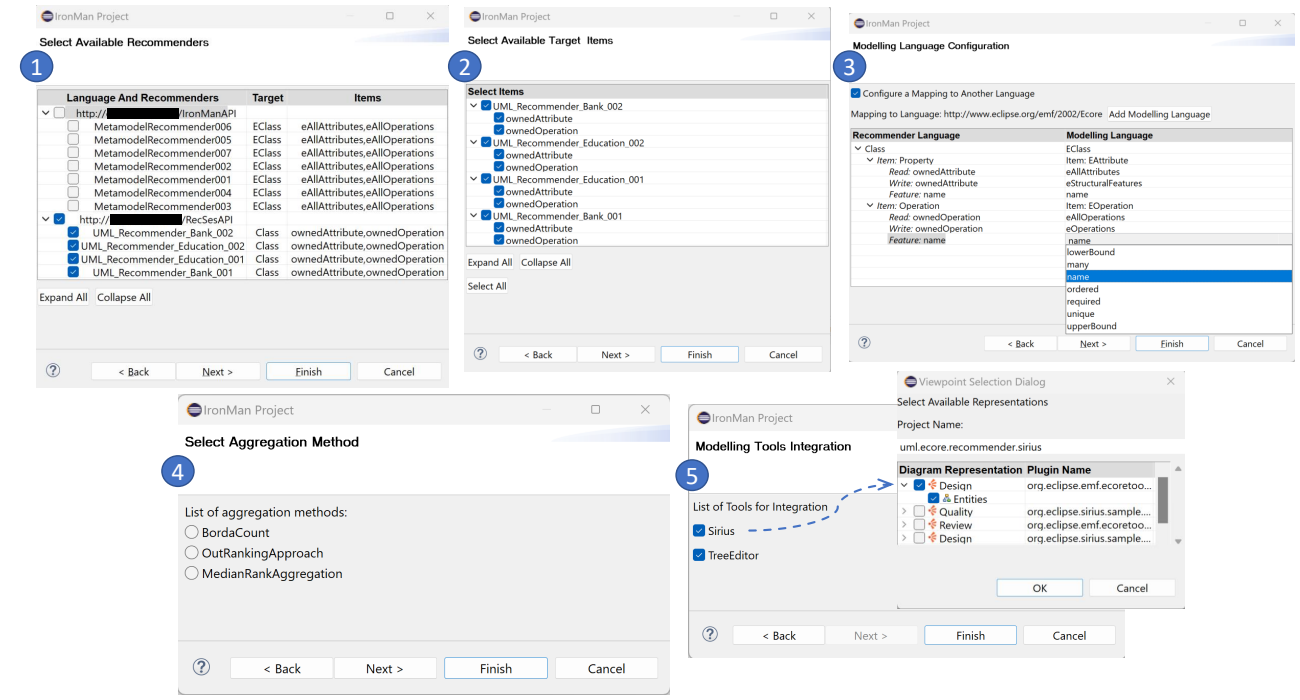


Figure 10. IronMan wizard in action: (1) Selecting recommender services (*anonymised*), (2) filtering the items to be recommended, (3) mapping the RS to the modelling language (optional step), (4) selecting the aggregation method, (5) selecting the integration with modelling tools.

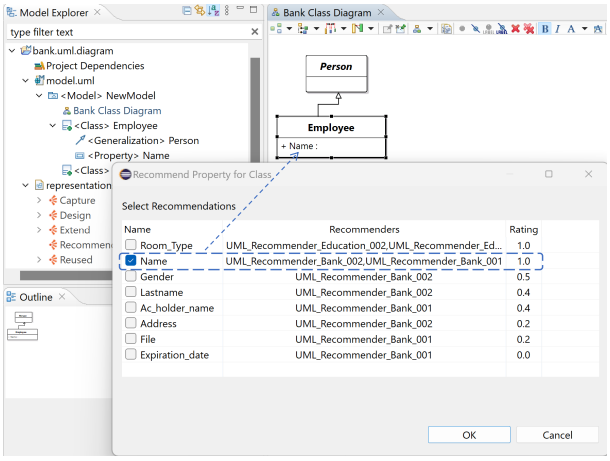


Figure 11. Integration of the RS within a Sirius editor.

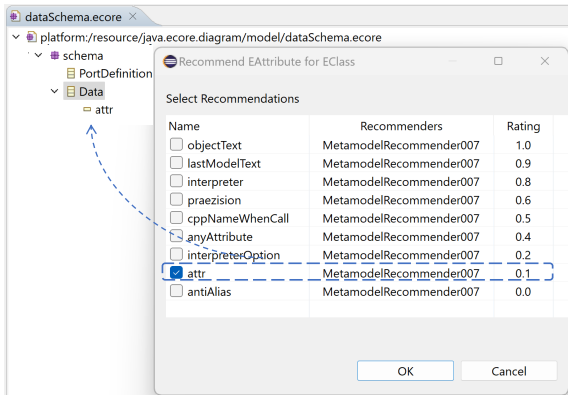


Figure 12. Integration of the RS within a tree editor.

Fig. 12 shows the integration of a RS within the standard Ecore tree editor [57]. 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`.

5 Evaluation

Next, we report on an evaluation to assess the usefulness of our approach in terms of its capacity to reuse RSs and to integrate them with existing modelling tools. Hence, we aim to answer the following research questions (RQs):

RQ1. Can IronMan be used to adapt existing RSs to different languages?

RQ2. Can IronMan be used to integrate existing RSs into third-party modelling tools?

Next, Sec. 5.1 describes the experiment set-up, Sec. 5.2 reports the results, Sec. 5.3 answers the RQs, and Sec. 5.4 discusses threats to the validity of the experiment.

Table 3. Experiment set-up.

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

5.1 Experiment set-up

To answer the RQs, we used two RSs for Ecore and UML reported in [2] and [4]. Both recommend attributes and operations for classes. Since both were already deployed as services, their adaptation to make them conformant to the API described in Sec. 3.2 was very light.

The aim of our evaluation is twofold. On the one hand, we aim to assess if the RSs can be adapted to other modelling languages. In particular, we check if the RSs can be adapted to UML, Ecore, Entity relationship (ER) diagrams, and the Interaction Flow Modelling Language (IFML) [28]. 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 want to assess the integration of the RSs into existing tree and Sirius editors built by third parties.

Table 3 summarises the experiment set-up. In the experiment, 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.

5.2 Experiment results

Table 4 summarises the results of the integrations, including the number of mappings needed to adapt the RS to the modelling language, and the synthesised lines of code (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).

The generated plugins use the EMF reflective API [57], 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 contains the description of the modelling environment, including the recommendation layer.

Fig. 13 contains 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) [40]. The resulting plugins are available at: https://anonymous.4open.science/r/integrate-recommenders-ironman-05C9/.

Table 4. Summary of the experiment results.

Integr.	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

5.3 Answering the RQs

Overall, we can answer both RQs positively.

For RQ1, we could reuse RSs defined for either 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 RQ2, 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.

5.4 Discussion and threats to validity

Our experiment shows evidence that IronMan is able to reuse existing RSs for other modelling languages, and integrate them automatically into existing tools.

However, regarding *external validity threats*, the number of RSs reused was limited, as the experiment only considered RSs for Ecore and UML. Similarly, we reused these RSs just for four other modelling languages. A stronger evaluation 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

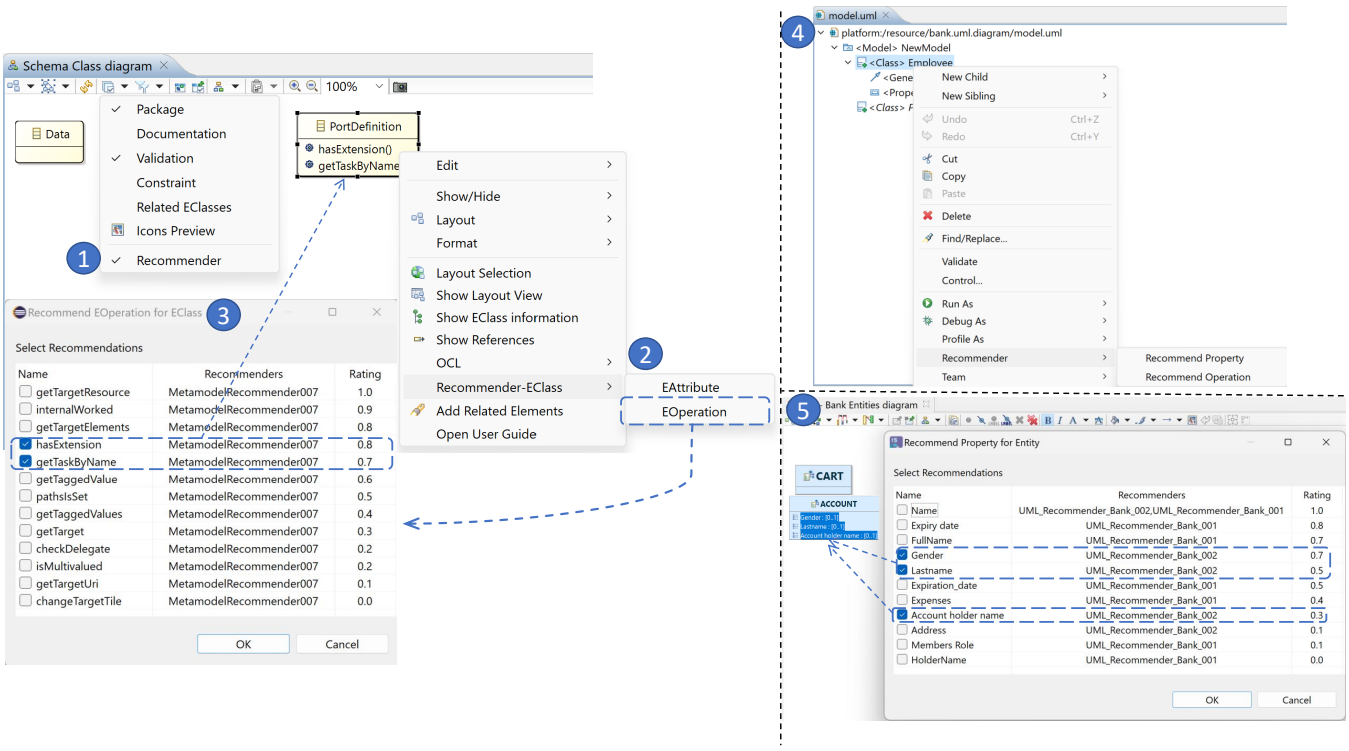


Figure 13. Screenshots of the integration results. (1–3) Ecore tools (Sirius), (4) UML tree editor, (5) ISD-Designer.

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 evaluation.

6 Related Work

Next, we review three lines of works related to our proposal: RSs for modelling tasks (Sec. 6.1), servitisation of RSs (Sec. 6.2), and aggregation of recommendations (Sec. 6.3).

6.1 RSs for modelling

As reported in a recent mapping review [3], the modelling community is showing a growing interest in RSs for modelling tasks. According to this review, the most common purposes of RSs in model-driven engineering (MDE) are the completion, finding, repair, reuse and, to a lesser extent, creation of modelling artefacts.

RSs for modelling have been normally developed ad-hoc for a specific modelling language, most frequently UML (e.g., class diagrams [9, 23, 26, 35, 38, 53] and sequence diagrams [12]), process modelling notations [15, 30, 31, 34, 36], or meta-models [1, 16]. Only a few language-independent approaches [1, 2, 21, 32, 45, 56] enable the definition of RSs for any language defined by a meta-model. Among them, Almonte et al. [2] propose the DSL *Droid* to facilitate the construction and subsequent evaluation of RSs for any modelling language. This DSL supports the selection and configuration of the recommendation algorithm. In this paper, we have evaluated our proposal using some RSs created with *Droid*. Given that RSs are typically fixed for a particular language, a tool like IronMan, which can adapt a RS for different notations, becomes useful in practice.

Regarding the techniques to generate recommendations in MDE, the most popular ones stem from classical recommendation methods, most prominently knowledge-based techniques (i.e., systems that exploit the domain knowledge to produce recommendations, such as *AMOR* [9], *Baya* [13], *IPSE* [26], *RapMOD* [35], *Refacola* [56], *ReVision* [42] and Savary-Leblanc's recommender [53]), followed by content-based (e.g., *DoMoRe* [1] and *Refactory* [47]), hybrid (e.g., SBPR [30] and the approaches by Kögel et al. [32] and Koschmider et al. [34]), and based on collaborative filtering (e.g., *MemoRec* [16] and *ModBud* [51]). Some recent approaches apply machine learning to build the RSs. For example, Burgueño et al. [10] propose a RS for class diagrams based on natural language processing; Weyssow et al. [59] apply a deep learning model to recommend meta-model concepts; Di Rocco et al. [18] use an encoder-decoder network to aid modellers in executing model editing operations, and graph neural networks (GNNs) to assist in the specification

of (meta-)models [17]; and Shilov et al. [54] use GNNs to assist in enterprise modelling processes.

Altogether, we observe a wide variety of RSs for modelling languages, targeting diverse modelling notations using different methods. Our contribution in this line of works is a proposal to homogenise the access to all existing approaches behind a common API. This enables the combination of the approaches by aggregating their results, and facilitates the access from arbitrary clients. Finally, our mappings permit adapting existing RSs to other modelling notations.

6.2 Deployment of RSs via web services

The idea of deploying RSs as web services is not new, but it has been adopted by both researchers and companies due to its benefits. A recommendation API is a (REST) service providing recommendation functionalities akin to those of recommendation software libraries, but hosting the data in the cloud. Going beyond, the RSs community has coined the term *Recommendations-As-a-Service* (RaaS) [50] to refer to cloud platforms that enable the creation of RSs using a few clicks or LoC, by automating the steps of the recommendation generation process, from data indexing to recommendation generation and display. As an example, the engine *Recombee* [46] allows building recommendation services for any domain that has a catalogue of items and is interacted by users. While the engine only supports content-based recommendation, its recommendation model is customisable and permits 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* [11], *BibTip* [8] and *Mr. DLib* [39] 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-specific tasks becomes cumbersome.

Regarding RSs for modelling, most proposals are deployed locally and integrated ad-hoc in a specific modelling tool or IDE. One of the few exceptions are *Droid* [2] and Savary-Leblanc's recommender for UML [53]. The latter is a recommender for UML class diagrams, deployed as a service, and integrated within Papyrus. *Droid* permits the creation of RSs for modelling languages and their deployment as REST services. In [4], the authors illustrate the integration of this service in both EMF tree editors and a modelling chatbot. Instead, our approach aims to be more general by unifying any modelling recommendation service under a common API. This will facilitate the reuse and aggregation of existing RSs by the community, hence contributing to better modelling tools augmented with recommendation capabilities.

6.3 Aggregation of RSs

Rank aggregation has been used in a wide number of fields, such as meta-search engines [20], biology [33], criticality analysis [43], or spam detection [61], to name a few. A few proposals exist in the RSs literature as well. For example, based on the observation that many top-N recommenders disagree in their returned rankings, Oliveira et al. [44] studied 19 rank aggregation methods and identified the recommendation scenarios where they performed best or worst. They concluded that rank aggregation achieves the biggest improvements in scenarios with high-quality input rankings and high diversity; unsupervised methods should be avoided in case of poor-quality input rankings; and the results of both supervised and unsupervised methods is similar in case of input rankings with high-quality but low diversity.

To our knowledge, our proposal is the first one enabling the aggregation of recommendations for modelling. We currently support unsupervised methods, and leave supervised ones as future work.

7 Conclusions and Future Work

Given the increasing number of proposals of RSs for modelling, there is a need for mechanisms to facilitate their reuse, combination and integration into modelling tools. We have presented an approach towards this goal, based on a common recommendation API, mappings bridging the RSs and the modelling notations, and algorithms for rank aggregation. The approach has been realised in IronMan, which is able to adapt and integrate RSs within modelling tools of the Eclipse ecosystem, based on Sirius and tree editors.

In the future, we would like to integrate further existing RSs, and investigate the scenarios where aggregating RSs – providing recommendations for the same target and item – are beneficial. While we currently support unsupervised rank aggregation methods, in the future, we plan to extend our proposal with supervised aggregation methods [44]. These methods require a previous customisation step to optimise the recommendation aggregation function with respect to a given metric, typically precision.

We would also like to include more flexible mechanisms for adapting the RSs to the modelling language (e.g., using OCL or Java snippets). In the wizard, all selected RSs need to be defined for the same modelling language, and then adapted if needed. In the short term, we will support the selection of RSs defined over different languages, and then provide assistants to facilitate adaptation to the modelling language (once the first mapping is defined). We also plan to support integration with textual notations, e.g., defined in frameworks like Xtext [7, 60]. Finally, it would be interesting to explore other types of integration of the RSs with modelling tools, e.g., a proactive approach where the RS monitors the modelling session and gets triggered when a good recommendation opportunity is found.

References

[1] Henning Agt-Rickauer, Ralf-Detlef Kutsche, and Harald Sack. 2018. DoMoRe - A Recommender System for Domain Modeling. In *6th*

International Conference on Model-Driven Engineering and Software Development (MODELSWARD). SciTePress, 71–82.

[2] Lissette Almonte, Esther Guerra, Iván Cantador, and Juan de Lara. 2022. Building recommenders for modelling languages with Droid. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 1–4.

[3] Lissette Almonte, Esther Guerra, Iván Cantador, and Juan de Lara. 2022. Recommender systems in model-driven engineering. *Softw. Syst. Model.* 21, 1 (2022), 249–280.

[4] Lissette Almonte, Sara Pérez-Soler, Esther Guerra, Iván Cantador, and Juan de Lara. 2021. Automating the Synthesis of Recommender Systems for Modelling Languages. In *14th ACM SIGPLAN International Conference on Software Language Engineering (SLE)*. ACM, 22–35.

[5] Apache. (last accessed in July 2023). Tomcat. http://tomcat.apache.org/.

[6] Javed A. Aslam and Mark H. Montague. 2001. Models for Metasearch. In *International Conference on Research and Development in Information Retrieval (SIGIR)*. ACM, 275–284.

[7] Lorenzo Bettini. 2016. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.

[8] BibTip. (last accessed in July 2023). http://www.bibtip.com/en.

[9] Petra Brosch, Martina Seidl, and Gerti Kappel. 2010. A recommender for conflict resolution support in optimistic model versioning. In *25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*. ACM, 43–50.

[10] Loli Burgueño, Robert Clarisó, Sébastien Gérard, Shuai Li, and Jordi Cabot. 2021. An NLP-Based Architecture for the Autocompletion of Partial Domain Models. In *33rd International Conference on Advanced Information Systems Engineering (CAiSE) (LNCS, Vol. 12751)*. Springer, 91–106.

[11] bX. (last accessed in July 2023). https://www.exlibrisgroup.com/products/bx-recommender/.

[12] Thaciana Cerqueira, Franklin Ramalho, and Leandro Balby Marinho. 2016. A Content-Based Approach for Recommending UML Sequence Diagrams. In *28th International Conference on Software Engineering and Knowledge Engineering (SEKE)*. KSI Research Inc. and Knowledge Systems Institute Graduate School, 644–649.

[13] Soudip Roy Chowdhury, Florian Daniel, and Fabio Casati. 2014. Recommendation and Weaving of Reusable Mashup Model Patterns for Assisted Development. *ACM Transactions on Internet Technology* 14, 2-3 (2014), 21:1–21:23.

[14] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. 2013. Reusable abstractions for modeling languages. *Inf. Syst.* 38, 8 (2013), 1128–1149.

[15] ShuiGuang Deng, Dongjing Wang, Ying Li, Bin Cao, Jianwei Yin, Zhaohui Wu, and Mengchu Zhou. 2017. A Recommendation System to Facilitate Business Process Modeling. *IEEE Trans. Cybern.* 47, 6 (2017), 1380–1394.

[16] Juri Di Rocco, Davide Di Ruscio, Claudio Di Sipio, Phuong T. Nguyen, and Alfonso Pierantonio. 2023. MemoRec: A recommender system for assisting modelers in specifying metamodels. *Softw. Syst. Model.* to appear (2023).

[17] Juri Di Rocco, Claudio Di Sipio, Davide Di Ruscio, and Phuong T. Nguyen. 2021. A GNN-based Recommender System to Assist the Specification of Metamodels and Models. In *24th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*. IEEE, 70–81. https://doi.org/10.1109/MODELS50736.2021.00016

[18] Juri Di Rocco, Claudio Di Sipio, Phuong T. Nguyen, Davide Di Ruscio, and Alfonso Pierantonio. 2022. Finding with NEMO: A Recommender System to Forecast the next Modeling Operations. In *25th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*. ACM, 154–164.

[19] Claudio Di Sipio, Juri Di Rocco, Davide Di Ruscio, and Phuong T. Nguyen. 2023. MORGAN: a modeling recommender system based on graph kernel. *Softw. Syst. Model.* to appear (2023), 70–81.

[20] Cynthia Dwork, Ravi Kumar, Moni Naor, and D. Sivakumar. 2001. Rank aggregation methods for the Web. In *International World Wide Web Conference, WWW*. ACM, 613–622.

[21] Andrej Dyck, Andreas Ganser, and Horst Lichter. 2014. A framework for model recommenders - Requirements, architecture and tool support. In *MODELSWARD*. IEEE, 282–290.

[22] Eclipse. (last accessed in July 2023). Jersey. https://eclipse-ee4j.github.io/jersey/.

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

[24] Ronald Fagin, Ravi Kumar, and D. Sivakumar. 2003. Efficient similarity search and classification via rank aggregation. In *International Conference on Management of Data ()*. ACM, 301–312.

[25] Mohamed Farah and Daniel Vanderpooten. 2007. An outranking approach for rank aggregation in information retrieval. In *SIGIR 2007: Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 591–598.

[26] Hilke Garbe. 2012. Intelligent Assistance in a Problem Solving Environment for UML Class Diagrams by Combining a Generative System with Constraints. In *eLearning*. IADIS.

[27] Object Management Group. 2014. OCL Specification. http://www.omg.org/spec/OCL/.

[28] Object Management Group. 2015. IFML Specification. https://www.ifml.org/.

[29] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-021. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

[30] Hadjer Khider, Slimane Hammoudi, and Abdelkrim Meziane. 2020. Business Process Model Recommendation as a Transformation Process in MDE: Conceptualization and First Experiments. In *8th International Conference on Model-Driven Engineering and Software Development, MODELSWARD*. SCITEPRESS, 65–75.

[31] Krzysztof Kluza, Mateusz Baran, Szymon Bobek, and Grzegorz J. Nalepa. 2013. Overview of Recommendation Techniques in Business Process Modeling. In *9th Workshop on Knowledge Engineering and Software Engineering, KESE (CEUR Workshop Proceedings, Vol. 1070)*. CEUR-WS.org.

[32] Stefan Kögel. 2017. Recommender system for model driven software development. In *11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 1026–1029.

[33] Raivo Kolde, Sven Laur, Priit Adler, and Jaak Vilo. 2012. Robust rank aggregation for gene list integration and meta-analysis. *Bioinform.* 28, 4 (2012), 573–580.

[34] Agnes Koschmider, Thomas Hornung, and Andreas Oberweis. 2011. Recommendation-based editor for business process modeling. *Data Knowl. Eng.* 70, 6 (2011), 483–503.

[35] Tobias Kuschke, Patrick Mäder, and Patrick Rempel. 2013. Recommending Auto-completions for Software Modeling Activities. In *16th International Conference on Model-Driven Engineering Languages and Systems (MoDELS) (LNCS, Vol. 8107)*. Springer, 170–186.

[36] Ying Li, Bin Cao, Lida Xu, Jianwei Yin, ShuiGuang Deng, Yuyu Yin, and Zhaohui Wu. 2014. An Efficient Recommendation Method for Improving Business Process Modeling. *IEEE Trans. Ind. Informatics* 10, 1 (2014), 502–513.

[37] Tie-Yan Liu. 2011. *Learning to Rank for Information Retrieval*. Springer.

[38] Patrick Mäder, Tobias Kuschke, and Mario Janke. 2021. Reactive Auto-Completion of Modeling Activities. *IEEE Trans. Software Eng.* 47, 7 (2021), 1431–1451.

[39] Mr-DLib. (last accessed in July 2023). http://mr-dlib.org/.

[40] Obeo. (last accessed in July 2023). IS-designer. https://www.obeosoft.com/en/products/is-designer.
[41] Obeo. (last accessed in July 2023). UML Designer. https://marketplace.eclipse.org/content/uml-designer.
[42] Manuel Ohrndorf, Christopher Pietsch, Udo Kelter, Lars Grunske, and Timo Kehrer. 2021. History-Based Model Repair Recommendations. *ACM Trans. Softw. Eng. Methodol.* 30, 2, Article 15 (2021), 46 pages. https://doi.org/10.1145/3419017
[43] Gabriele Oliva, Annunziata Esposito Amideo, Stefano Starita, Roberto Setola, and Maria Paola Scaparra. 2019. Aggregating Centrality Rankings: A Novel Approach to Detect Critical Infrastructure Vulnerabilities. In *International Conference on Critical Information Infrastructures Security, CRITIS (LNCS, Vol. 11777)*. Springer, 57–68.
[44] Samuel E. L. Oliveira, Victor Diniz, Anísio Lacerda, Luiz H. C. Merschmann, and Gisele L. Pappa. 2020. Is rank aggregation effective in recommender systems? An experimental analysis. *ACM Trans. Intell. Syst. Technol.* 11, 2 (2020), 16:1–16:26.
[45] Tanumoy Pati, Sowmya Kolli, and James H. Hill. 2017. Proactive modeling: a new model intelligence technique. *Softw. Syst. Model.* 16, 2 (2017), 499–521.
[46] Recombee. (last accessed in July 2023). https://docs.recombee.com/.
[47] Jan Reimann, Mirko Seifert, and Uwe Aßmann. 2013. On the reuse and recommendation of model refactoring specifications. *Softw. Syst. Model.* 12, 3 (2013), 579–596.
[48] Francesco Ricci, Lior Rokach, and Bracha Shapira. 2022. *Recommender Systems Handbook* (3 ed.). Springer US.
[49] Martin P. Robillard and Robert J. Walker. 2014. An Introduction to Recommendation Systems in Software Engineering. In *Recommendation Systems in Software Engineering*, Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann (Eds.). Springer, 1–11.
[50] RS_c. (last accessed in July 2023). Recommendation-as-a-service. https://recommender-systems.com/resources/recommendations-as-a-service-raas/.
[51] Rijul Saini, Gunter Mussbacher, Jin L. C. Guo, and Jörg Kienzle. 2019. Teaching Modelling Literacy: An Artificial Intelligence Approach. In *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS*. IEEE, 714–719.
[52] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2012. Flexible Model-to-Model Transformation Templates: An Application to ATL. *J. Object Technol.* 11, 2 (2012), 4: 1–28.
[53] Maxime Savary-Leblanc, Xavier Le-Pallec, and Sébastien 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)*. IEEE, 630–634. https://doi.org/10.1109/MODELS-C53483.2021.00097
[54] Nikolay Shilov, Walaa Othman, Michael Fellmann, and Kurt Sandkuhl. 2023. Machine learning for enterprise modeling assistance: An investigation of the potential and proof of concept. *Softw. Syst. Model.* to appear (2023), 1619–1374. https://doi.org/10.1007/s10270-022-01077-y
[55] Sirius. (last accessed in April 2023). https://www.eclipse.org/sirius/.
[56] Friedrich Steimann and Bastian Ulke. 2013. Generic Model Assist. In *16th International Conference on Model-Driven Engineering Languages and Systems, MODELS (LNCS, Vol. 8107)*. Springer, 18–34.
[57] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework, 2nd Edition.* Addison-Wesley Professional. See also http://www.eclipse.org/modeling/emf/.
[58] Javier Alvaro Vargas Muñoz, Ricardo da Silva Torres, and Marcos André Gonçalves. 2015. A Soft Computing Approach for Learning to Aggregate Rankings. In *Proceedings of the 24th ACM International Conference on Information and Knowledge Management, CIKM*. ACM, 83–92.
[59] Martin Weyssow, Houari Sahraoui, and Eugene Syriani. 2022. Recommending Metamodel Concepts during Modeling Activities with Pre-Trained Language Models. *Softw. Syst. Model.* 21, 3 (2022), 1071–1089.
[60] Xtext. (last accessed in July 2023). http://www.eclipse.org/Xtext/.
[61] Zheng Zhang, Ming-Yang Zhou, Jun Wan, Kezhong Lu, Guoliang Chen, and Hao Liao. 2023. Spammer detection via ranking aggregation of group behavior. *Expert Syst. Appl.* 216 (2023), 119454.