Generic Graphical Navigation for Modelling Tools

Hyacinth Ali, Gunter Mussbacher(✉), and Jörg Kienzle

McGill University, Montreal, Canada
hyacinth.ali@mail.mcgill.ca, {gunter.mussbacher,joerg.kienzle}@mcgill.ca

Abstract. To describe the characteristics of software systems, model-driven engineering (MDE) advocates the use of different modeling languages and multiple views that modellers need to navigate in the models' editors to understand and modify the system under development. This paper introduces a generic navigation mechanism that facilitates navigation within a model, from one model to other linked models potentially expressed in a different language, as well as for feature-based development and across reuse hierarchies. Furthermore, a proposed navigation bar visually indicates to the modeller the place of a model in this structure. To make a modelling language navigable, a language designer enhances the modelling language at the metamodel level with our generic navigation capabilities, which include the ability to filter language elements based on attribute values. We present evidence that the proposed generic navigation mechanism comprehensively supports model navigation by analyzing the navigation facilities offered by popular UML modelling tools and a feature-based modelling tool.

Keywords: Navigation · Domain-specific language · Multi-view modelling · Features · Reuse · Model-driven engineering

1 Introduction

Model-driven engineering (MDE) [1] advocates the use of different modelling languages and multiple views to *describe* the characteristics of software systems as well as to *prescribe* their structure and behaviour. The software development process that is being used establishes conceptual and causal links between models, potentially crossing different levels of abstraction. While several works have focused on describing and enforcing these relationships [2–4], graphical navigation support for the user of a modelling tool within and across models has received only limited attention. This is the case even though studies have shown the importance of good visualization and navigation mechanisms in both software usage and during development [5–7].

With the proliferation of domain-specific modeling languages (DSMLs) [8], one cannot assume anymore that a fixed set of modeling languages is used to develop software systems. Rather, a flexible modeling environment needs to be

P. Fonseca i Casas et al. (Eds.): SAM 2019, LNCS 11753, pp. 44–60, 2019.
https://doi.org/10.1007/978-3-030-30690-8_3

provided that allows sets of languages to be integrated as the needs arise. Consequently, the corresponding set of models needs to be navigated. This navigation is not about generic navigation of models with the Object Constraint Language (OCL) or similar languages, but rather the navigation of models by the modeller in the models' editors.

In this paper, we present a generic approach for language designers and modelling tool developers to specify navigation mappings within a model, from one model to other linked models potentially expressed in a different language, as well as for feature-oriented development and across reuse hierarchies. We show how these mappings can be used to populate a navigation bar with navigation links that make it easy for a user to traverse models and navigate to related model elements. We use different colour highlighting to help the user find model elements within large models, and to inform the user when navigation links cross model boundaries. We explain how we encoded our generic approach in a metamodel targeting modelling languages and tools developed as part of the Eclipse Modelling Framework (EMF), and illustrate intra-model, inter-model, and inter-language navigation by means of a small example. We further demonstrate how our navigation approach can be used in a reuse-context and to filter language elements. Furthermore, we analyze popular UML modelling tools and a feature-oriented modelling tool with respect to their navigation capabilities. For each tool, we explain which navigation capabilities they support and show that the navigation concepts in our proposed metamodel are sufficient to handle them.

In the remainder of this paper, Sect. 2 presents generic language navigation by means of a running example. Section 3 elaborates our navigation metamodel and Sect. 4 discusses the navigation capabilities of several UML modelling tools. We briefly review related work in Sect. 5. The conclusion in Sect. 6 provides a summary and discusses future work.

2 Generic Language Navigation

MDE advocates the use of models expressed in different languages to capture the many characteristics of systems. This set of models needs to be navigated to understand the system under development. In this section, we first motivate our proposed generic navigation facility with the help of four examples, each representing a typical navigation situation.

2.1 Single Model Navigation

The first situation concerns the navigation of a single model, i.e., intra-model (and hence also intra-language) navigation. A complex model may consist of many model elements, and it is hence desirable to have a concise and easy-to-use way to find important model elements or navigate model element relationships.

For example, Fig. 1 depicts a class diagram of a bank system and our proposed navigation bar. Clicking the drop-down arrow under *BankClassDiagram* pops up the *Classes* of the model, listed under the tab *Classes*. Clicking on a class

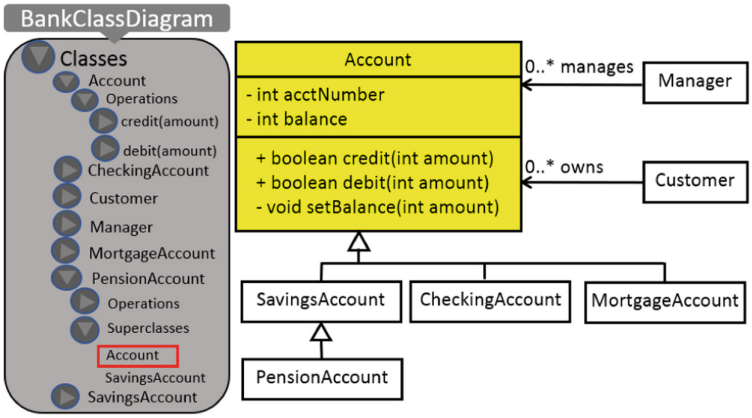

Fig. 1. Bank class diagram

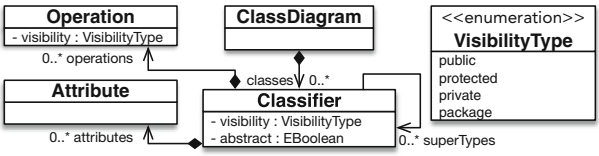

Fig. 2. Class diagram metamodel, *(an excerpt)*

reveals the operations and superclasses of the class in the navigation bar. In this example, we navigate from the class diagram to the class, `PensionAccount`, and then to its superclass, `Account`. Once a class is selected, the background of the class is highlighted in yellow in the model and centred on the screen, if needed, for easier identification.

To realize this navigation, several *navigation mappings* have to be specified on the class diagram metamodel shown in Fig. 2. The first navigation mapping has the `ClassDiagram` metaclass as its source and the `classes` reference as its target. The second and third mappings have the `Classifier` as their source and the `operations` rsp. `superTypes` reference as their target. A reference is used as the target instead of a metaclass, because one metaclass may have several relationships with another metaclass.

These three navigation mappings each consist of one *hop*. However, it may sometimes be necessary to skip intermediate elements and, e.g., define a navigation that goes from a class diagram directly to all the operations defined in the diagram without listing all the classes first. Such a navigation requires multiple hops, e.g., first from the `ClassDiagram` to the `classes` reference and then on with the `operations` reference.

The navigation mapping from `Classifier` to `superTypes` is different compared to the other mappings, because it is useful to not only show the direct

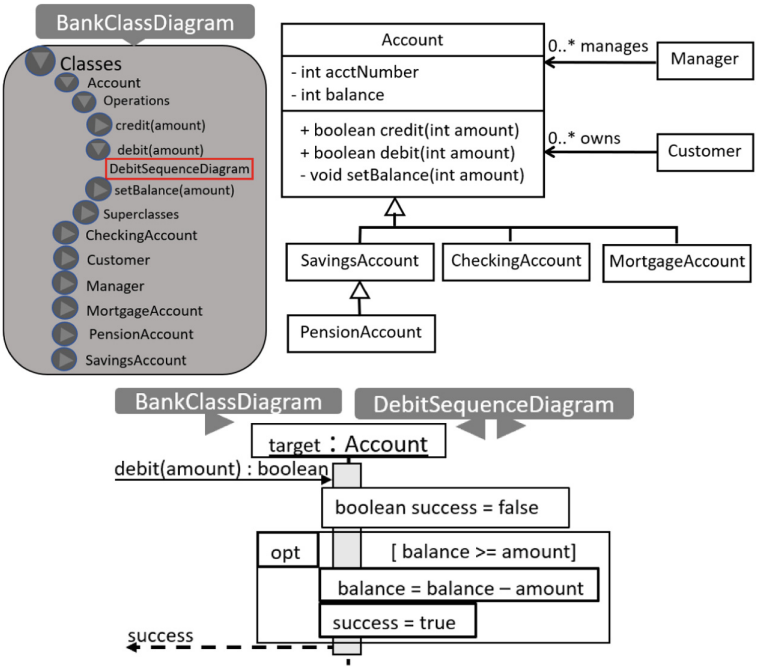

Fig. 3. Bank class diagram and sequence diagram of debit operation

superclass of a class, but instead the complete hierarchy of superclasses. For these situations we provide a *closure* flag that can be set.

2.2 Multi-view Navigation

The second situation concerns multi-view modeling, i.e., inter-model navigation. The navigation may involve models of the same type, i.e., intra-language navigation, or models from different languages, i.e., inter-language navigation. An example of inter-model, intra-language navigation is a sequence diagram that defines the behaviour of an operation, which sends messages to invoke other operations. In this case, one may want to navigate from the invocation message in the first sequence diagram to another sequence diagram showing the detailed behaviour of the invoked operation. This navigation can be handled the same way as single model navigation, with the *from* element being the message and one *hop* to its sequence diagram reference. In this case, though, the reference points to a model element in a different model.

An example for inter-model, inter-language navigation is a class diagram, where one may want to navigate from an operation declaration in a class to a sequence diagram defining the behaviour of the operation as shown in Fig. 3. In this situation, the two languages – the class diagram language and the sequence

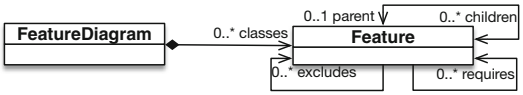

Fig. 4. Feature diagram metamodel, *(an excerpt)*

diagram language – are used together in a specific way for a purpose, which we term *perspective* in this paper.

In the navigation bar, this connection is also visualized as the "right" arrow, which opens a drop-down list similar to intra-model navigation. When *debit(amount)* under the *Operations* tab is clicked, a list of other linked models pops up. Upon clicking the *DebitSequenceDiagram* tab, as highlighted in the figure with a red box, the linked sequence diagram is opened. Because this navigation involves a different type of model, the navigation bar is extended to display the class diagram model name as well as the sequence diagram model name to the right.

Navigating back to the class diagram can then simply be achieved by directly clicking on the class diagram name in the navigation bar. Furthermore, the sequence diagram has a "left" arrow which also opens a drop-down box to navigate any incoming inter-model navigation mappings in the opposite direction. For example, a workflow model may establish a mapping from one of its steps to the same sequence diagram. The "left" arrow then allows navigating from the sequence diagram to the class diagram or to the workflow model.

We also need to take into account that it is possible to directly open any model using a file browser. When the above sequence diagram is opened directly with a file browser (and not through navigation starting with a class diagram), the navigation bar should still show that the sequence diagram depicts behaviour that is best understood in the context of the class diagram or workflow model. To determine which model should be shown in the navigation bar, the boolean attribute (*default*) of one of the incoming inter-model mappings is set to true.

The main difference to the intra-language navigation mappings described in the previous section is that there exists no prior link between the metamodel of the class diagram language and the metamodel of the sequence diagram language (assuming that these two metamodels have been developed independently). Consequently, an inter-language mapping involves a *from* metaclass and a *to* metaclass instead of reference hops.

2.3 Software Product Line Navigation

The third situation involves Software Product Line (SPL) development, which groups related model artifacts with commonalities and variabilities for a given family of products [9]. In SPL, a *feature* designates a user-relevant functionality or system quality that can be present or not in a product. A feature diagram describes the relationships among features, i.e., the set of feature configurations that produce valid products.

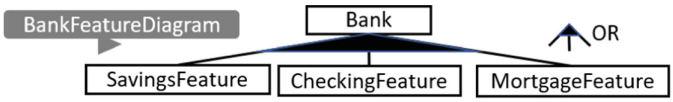

Fig. 5. Feature diagram of a bank system

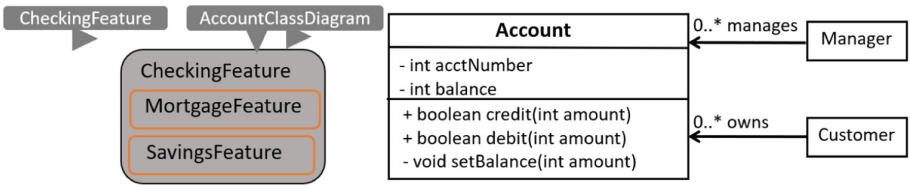

Fig. 6. Account class diagram in CheckingFeature

Figure 4 depicts a metamodel for feature diagrams. A `FeatureDiagram` basically has a list of `Features` with parent/children relationships among them. Some of these features may be optional, while others are mandatory, and define *requires* and *excludes* relationships to other features. Figure 5 shows an example feature model for a bank system supporting different kinds of bank accounts. The features *SavingsFeature*, *CheckingFeature*, and *MortgageFeature* are in an OR relationship, meaning that at least one of them must be selected in order to create a valid configuration.

In model-driven SPLs, the structural and behavioural properties of features are described with models linked to these features. In additive variability, each feature is realized by one or several models, and to derive a product the realization models corresponding to the chosen features are composed with each other. In negative variability, a so-called 150% model describes the system with all features enabled. Each feature is linked to model elements related to the feature, and to derive a product the model elements that are not linked to any chosen features are removed from the model.

While negative variability requires a highlighting feature similar to what is shown in Fig. 1, positive variability requires navigation among models. To illustrate feature-oriented navigation, we split the bank account class diagram from Fig. 1 into four smaller class diagrams. These class diagrams can then be composed (i.e., merged) to produce a bank model with the desired features.

Clicking on the "right" arrow under *BankFeatureDiagram* first shows the features and then the models realizing a feature (similar to the sequence diagrams of operations). Selecting a feature highlights the feature in the feature diagram, while selecting a model of a feature takes the modeler to the model associated with this feature as illustrated in Fig. 6.

Figure 6 shows the class diagram that contains the common structure used by *all* bank account features. At this time, though, the developer is currently working on the class diagram in the context of the *CheckingFeature*. This focus is depicted in the navigation bar by displaying the name of the *CheckingFeature* in

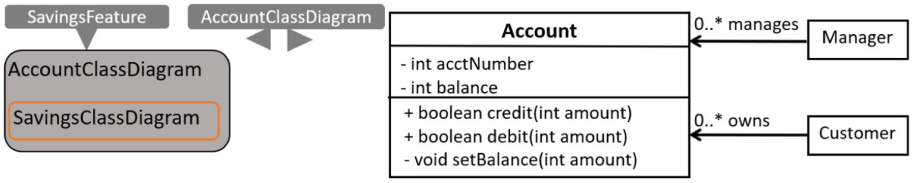

Fig. 7. Account class diagram in SavingsFeature

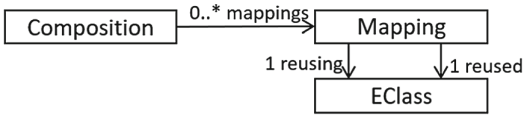

Fig. 8. Reuse metamodel, *(an excerpt)*

the navigation bar instead of the *BankFeatureDiagram*. The "right" arrow under *CheckingFeature* allows navigating to the models associated with the feature, i.e., the shared *AccountsClassDiagram* and the *CheckingClassDiagram* (which shows the generalization of the `CheckingAccount` class). The "left" arrow under the *AccountClassDiagram* shows a drop-down list with all other features that also use this class diagram. For example, when the "SavingsFeature" is clicked, the name "CheckingFeature" in the navigation bar is changed to "SavingsFeature", i.e., a context switch, and clicking the arrow under the "SavingsFeature" shows the models associated with it as shown in Fig. 7.

In terms of navigation mappings, feature-oriented navigation does not introduce any new kind of mapping. The mappings between a feature diagram and its features are intra-model mappings already discussed in Sect. 2.1. The mappings from features to class diagrams are inter-model, inter-language mappings already discussed in Sect. 2.2. However, since a feature is treated differently than other model element in terms of how it is displayed in the navigation bar, a `fromIsNavigationKey` flag needs to be set in its navigation mapping.

2.4 Navigation of Reusable Artifacts

The final situation discussed here concerns the use of reusable artifacts during software development. Consider the sequence diagram for `debit(amount)` in Fig. 9 and assume that a reusable artifact for authentication exists with a sequence diagram as shown in Fig. 10. When the `debit(amount)` sequence diagram reuses the *Authentication* sequence diagram, the body of the reusing sequence diagram replaces the box labeled with * in the reused sequence diagram. Consequently, the authentication check is performed before the body of the reusing sequence diagram. To specify this reuse, a composition specification needs to be provided that links the `debit(amount)` sequence diagram with the *Authentication* sequence diagram as defined in the metamodel for reuse specifications (see Fig. 8). The reuse metamodel captures the links between *reused*

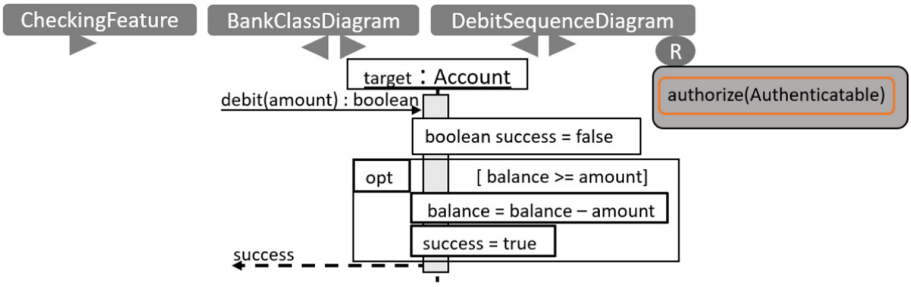

Fig. 9. Reuse of authentication

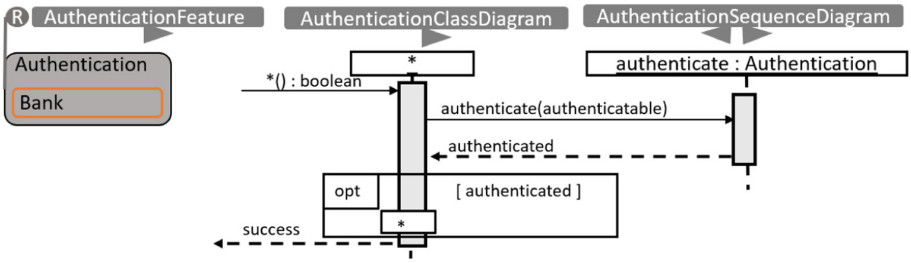

Fig. 10. Authentication reuse hierarchy

elements and *reusing* elements with a mapping. In our example, a mapping is established between the instance of the *SequenceDiagram* metaclass representing the `debit` operation to the *SequenceDiagram* metaclass instance representing the `authentication` operation. Once such a mapping in the reuse specification is established, it should be possible to navigate this composition link with the help of the proposed navigation bar.

To support this navigation, an "R" is displayed under the *DebitSequenceDiagram* in Fig. 9. Clicking on it shows all reuses of this model (or individual model elements of the model). Once a reuse is selected, the modeler is taken to the reusable artifact. This involves a context switch, which results in the navigation bar showing the reused sequence diagram with its default parent (i.e., its class diagram) and the default parent of the class diagram (i.e., its feature). As shown in Fig. 10, an "R" at the left of the navigation bar indicates the reuse hierarchy that is currently explored (e.g., the reusable artifact *Authentication* and the *Bank* that is reusing it). Clicking on an element in the reuse hierarchy results in direct navigation to that level.

In terms of navigation mappings, an intra-language mapping needs to be established (e.g., from the reusing sequence diagram to the reused sequence diagram). This navigation mapping requires a `from` element. Furthermore, two hops are required, which are references. The first hop is identified by the `reusing` reference and the second hop is identified by the `reused` reference. Note, however, that the `reusing` reference needs to be traversed in the reverse direction, because

the reference is at the side of the source element of the hop (i.e., the reusing sequence diagram). Since reuse links are treated differently than other navigation links (due to the required context switch from the reusing artifact to the reused artifact), the `reuse` flag needs to be set for this navigation mapping.

2.5 Filtering of Model Elements

A complex model diagram may have a large number of model elements, which may be overwhelming to show in the navigation bar. To streamline navigation, we support filtering of model elements. E.g., a modeler may want to find all classes in a system and show only the *public* operations of each class. We demonstrate this mechanism with the class diagram shown in Fig. 1, which depicts a bank system where the `Account` class has two public methods and one private method.

Clicking the drop-down arrow under *BankClassDiagram* in the navigation bar pops up the *Classes* of the model. Clicking on a class reveals the operations and superclasses of the class in the navigation bar. In this example, we navigate from the class diagram to the class, `Account`, and then only to its public operations, `credit(amount)` and `debit(amount)`.

To realize this filtering mechanism, a *filter condition* has to be encoded for the class diagram metamodel shown in Fig. 2. We filter based on an attribute value of the relevant model element. For example, the filter condition could be *abstract classes*, *public classes*, etc. In Fig. 1, the result based on filtering *public operations* is shown. To achieve this, the filter condition specifies the attribute of the metaclass that the filter should consider (i.e., the `visibility` attribute of the `Class` metaclass), the comparison value (i.e., the enumeration literal `public`), and a comparison operator (i.e., `EqualTo`).

To allow a modeler to dynamically configure which navigation mappings and associated filters the navigation bar uses to populate its content, it is possible to activate navigation mappings at runtime through preference settings.

3 Navigation Metamodel

This section describes our navigation metamodel that the designer of a language or modelling tool can use to define navigation mappings that configure our generic navigation bar. We elaborate our metamodel in the context of the Eclipse Metamodelling Framework (EMF), in which all metamodels are expressed using the metametamodelling language ECore. As such, any model element that is part of a language metamodel and could be selected as the source of a navigation link is encoded as an instance of the class `EClass`.

As explained with the examples above, for each `Perspective`[1] there are two broad categories of navigation, namely intra-language and inter-language navigation, which are indicated by two metaclasses (`IntraLanguageMapping`

[1] Recall that a *perspective* represents a purpose for using models expressed in one or several modelling languages during software development.

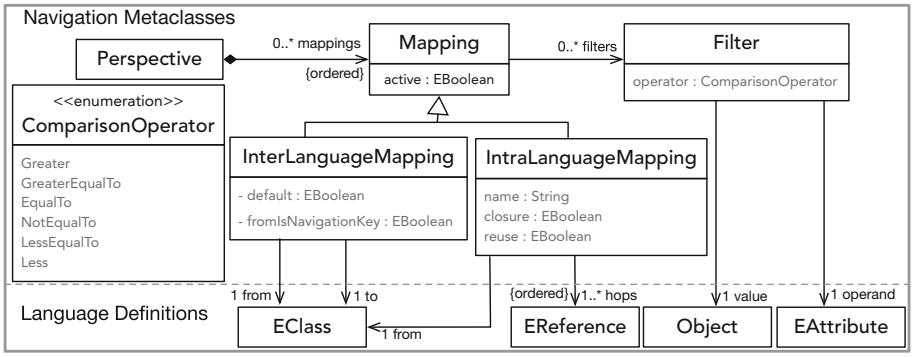

Fig. 11. Navigation metamodel

and `InterLanguageMapping`) in Fig. 11. In intra-language, we navigate *from* a model or one of its model elements (represented as `EClass`) to one or several elements of the same language by following references. In language metamodels defined with Ecore, these references are instances of `EReference`. Since navigation might involve traversing several references, every `IntraLanguageMapping` therefore defines an ordered collection of `EReference` called *hops*.

Furthermore, each intra-language mapping has the following three attributes: `name`, `closure`, and `reuse`. The string attribute `name` allows the tool designer to specify the text that should appear in the navigation bar for this navigation. The boolean `closure` attribute can be set for any `IntraLanguageMapping` where the *from* `EClass` is identical to the model element referred to by the last *hop*. In this case, the navigation bar will traverse this mapping recursively and display all reached target model elements. In our example, `closure` is set when navigating from a class to its superclasses in order to display the entire superclass hierarchy in the navigation bar. The boolean `reuse` identifies an intra-language navigation mapping that requires a context switch.

In case of inter-language mappings, the navigation involves models of different software languages, e.g., navigating from an operation definition in a class diagram to the sequence diagram specifying the behaviour of the operation. Hence, for `InterLanguageMappings`, the *from* and *to* are always instances of `EClass`, and each mapping is a 1-to-1 relationship. Finally, the `default` attribute specifies whether the source of an inter-language navigation mapping identifies the default parent of a target model. The `fromIsNavigationKey` attribute identifies key model elements (e.g., a feature) that need to be shown in the navigation bar instead of their model name.

To support filtering of language elements, we attach a `Filter` to the `Mapping` metaclass, which is the superclass of the `InterLanguageMapping` and `IntraLanguageMapping` navigation mappings. Filtering is always applied on the `to` elements in the case of inter-language filtering, or to the elements designated by the `EClass` referred to by the last `hop` in the case of intra-language filtering. The `operator` attribute specifies the comparison operator for the filtering using

pre-defined enumeration values as shown in Fig. 11. A filter then compares the attribute value of the `operand EAttribute` with the `value Object` designated by the filter. When several filter conditions are specified for a mapping, they are combined by an implicit logical AND.

Last but not least, the `active` attribute in the metaclass `Mapping` allows the navigation bar to be customized at runtime. For example, a modeller can toggle the active attribute to false if at some point he does not wish the operations of classes to show up in the navigation bar.

Our prototype implementation of the navigation bar ensures that the navigation information is always up-to-date by registering as a listener to all model elements that are instances of `EClass` involved in navigation mappings. Whenever a model is changed, the navigation bar is notified and the navigation links are adjusted according to the occurrences of the mappings in the model.

4 Evaluation

The Unified Modelling language (UML) [10] is a widely accepted standard for modelling software intensive systems. In its current version it defines 13 different diagrams. UML modelling tools facilitate the specification of systems at different levels of abstraction and from different points of view.

In this section, we analyse the navigation facilities of several popular modelling tools and evaluate whether our navigation metamodel covers them. We performed a Google search for "most popular UML tools". From the obtained list we investigated the top 4, namely: ArgoUML (free), StarUML (free), Visual Paradigm Enterprise (commercial), and MagicDraw (commercial). We also selected Papyrus, as a representation of a popular modelling tool based on EMF, and finally TouchCORE [11], as a representative of a UML modelling tool that explicitly supports software product line modelling and model reuse. In each tool, we specified a class diagram, and defined the behavior of some operations using sequence diagrams or state machines. We then explored how the tools support navigation. We organize our findings under the topics of intra-language and inter-language navigation, filtering, element highlighting, navigation of inheritance hierarchy, feature-oriented navigation, and navigation across reuse boundaries.

ArgoUML is an open source tool supporting all UML 1.4 diagrams [12]. *Intra-Language Navigation* in ArgoUML is done with the model explorer, which shows the list of diagrams and their contained elements. *Inter-Language Navigation* is limited, but clicking on an element in a diagram in the model explorer opens the corresponding diagram and highlights the selected element. ArgoUML supports different kinds of *filtering* using their own notion of *perspective*. Each perspective specifies the kind of model elements to be shown in the explorer. The tool allows modelers to define their own perspectives using existing rules by combining existing filter conditions from a provided library. When a model element is selected in the model explorer, the element is *highlighted* in blue in the editor, if it is currently visible on screen. The model explorer can also list all the model classes and their subclasses to explore the *inheritance hierarchy*.

Our proposed generic navigation approach can support all the navigation facilities that ArgoUML offers. The perspectives of ArgoUML can be represented as a filter condition in our generic mechanism. For example, the Class-Centric perspective lists only diagrams and classes in the model explorer. With our approach, this can be done by setting the `active` flag of `Mapping` for all instances of `Class` (see Figs. 2 and 11), and deactivating all other mappings.

StarUML is a modelling tool compatible with the UML 2.x standard and supporting 11 types of diagrams [13]. The tool partially supports *intra-language navigation* in the model explorer by right-clicking on a model element and choosing *Select In Diagram*. The tool supports *inter-language navigation* using the model explorer: clicking on a class shows the contained operations. Clicking on the operation displays the list of associated sequence diagrams, if any. StarUML has no support for filtering. Each model element in the currently displayed diagram can be *highlighted* in blue by selecting it in the model explorer. StarUML partially supports navigation of the *inheritance hierarchy* in class diagrams by navigating from a subclass to its parent class. However, it is not possible in StarUML to visualize the complete inheritance hierarchy of a given class.

Our proposed generic navigation approach can express all the navigation facilities that StarUML provides. Additionally, our approach supports filtering and displaying of the entire inheritance hierarchy.

MagicDraw [14] supports all UML diagrams. MagicDraw provides a structured containment tree which facilitates *navigation* from a model element to its related elements. Clicking on a model element displays it in the diagram editor, switching diagrams if necessary. However, just like in StarUML, the containment tree in MagicDraw displays the model element definitions separately from the diagrams in which they are used in. MagicDraw provides full support for *inter-language navigation*. A sequence diagram or activity diagram that is linked to an operation in a class diagram can be navigated to directly from the model element in the model editor. The tool has several *filter* conditions under three different categories, namely: *List*, *Inheritance*, and *Structural*. Each category has multiple options that can be turned on or off, e.g., Class, Actor, or Association. When a filter condition is enabled, the corresponding model elements are hidden in the containment tree. In the containment tree of the model explorer, a *superclass* can be navigated to by clicking a plus (+) tab before its subclass.

Our generic approach supports the navigation facilities of MagicDraw. The filtering in MagicDraw is at the granularity of model element types, i.e., every model element of a given type is either shown or not shown. Our generic mechanism supports this using the `active` flag in `Mapping` (see Fig. 11). Unlike our approach, MagicDraw does not support filtering based on attribute values, e.g., to define a filter that displays only abstract classes.

Visual Paradigm Enterprise supports UML 2 and SysML modelling [15]. Visual Paradigm has full support for *intra-language navigation* within the *diagram navigator* similar to MagicDraw. The tool also provides excellent support for *inter-language navigation*. E.g., when an operation in a class diagram has a linked sequence or state diagram, an icon is displayed with the class that can be

Table 1. Navigation support of UML tools

Tool	Intra-language	Inter-language	Attribute filtering	Activation filtering	Element highlighting	Inheritance hierarchy	SPL	Model reuse
ArgoUML	Yes	Yes	No	Yes	Yes	Yes	No	No
StarUML	Partial	Yes	No	No	Yes	Partial	No	No
MagicDraw	Yes	Yes	No	Yes	Yes	Partial	No	No
Visual Paradigm	Yes	Yes	No	No	Yes	Partial	No	No
Papyrus	Yes	Yes	No	Yes	Yes	Partial	No	No
TouchCORE	Yes	Yes	No	No	Yes	No	Yes	Yes

clicked to navigate to the linked diagrams. Visual Paradigm does not support filtering. A modeler can right-click an element in the explorer or diagram navigator and choose *Select In Diagram*. This takes the modeler to the diagram containing the element with the element being *highlighted* in bold, switching the current view if necessary. The tool only partially supports *inheritance navigation*, as a modeler can only navigate from a class to its direct superclasses.

Papyrus is a UML modelling tool based on EMF that supports many UML diagrams. The tool supports *navigation* of elements within a model in the model explorer, including traversing from a diagram to its elements. Papyrus uses *hyperlinks* to establish relationships between two diagrams, e.g., between a class and an activity diagram or a state diagram. The tool displays these *inter-language links* under the corresponding model elements in the model explorer. The contents for every model element shown in the diagram editor can be selectively hidden or shown by enabling or disabling *filter* options. For example in a class diagram, classes can be visualized with or without their attributes. Selected model elements in the model explorer, are *highlighted* in the diagram editor. Navigating from the model explorer to an element opens up the diagram containing the element in case it was not previously shown. Papyrus supports navigating from subclasses to direct *superclasses* only.

TouchCORE is a modelling tool for concern-oriented software design [11, 16], focussing specifically on feature-driven modularisation as required in SPLs. It also has explicit support for model reuse, and ships with a library of reusable models. The tool supports Feature Models, Goal Models, Class Diagrams, State Diagrams, and Sequence Diagrams. When selected in the model explorer, model elements in the current diagram are *highlighted* in orange. The model explorer allows the modeller to *navigate*, e.g. from an operation defined in a class diagram to an attached sequence diagram. TouchCORE does not support filtering of model elements nor navigation of the inheritance hierarchy. Since TouchCORE was designed to specifically support SPL, there is excellent support for *feature-oriented navigation*, e.g., navigating from a feature in a feature diagram to the associated realization model(s). Conversely, when visualizing a model in the model editor, the associated features are displayed and can be navigated to easily. The tool keeps track of reuse dependencies between models. A modeler can navigate from a current model to the *reused models* via the model explorer.

Evaluation Summary. Table 1 shows a summary of each tool's navigation capabilities. Each of the investigated tools has a model explorer, which corresponds to our navigation bar. Our proposed generic mechanism covers all the navigation means provided by the surveyed tools. No tool offers complete support for all navigation features provided by our proposed navigation mechanism. Only one tool supports the navigation of closures: ArgoUML supports the navigation of the entire inheritance hierarchy in a class diagram. Attribute-based filtering is not supported in any of the surveyed tools. However, we decided to include this feature in our proposed metamodel, because many development environments for programming languages have the ability to filter, e.g., by public elements. Of course, our proposed metamodel could easily employ a general query expression language for navigation purposes (e.g., OCL). However, the goal of this work is to provide the modeller with a *succinct* set of concepts needed for navigation in modelling editors instead of offering the full capabilities of languages such as OCL, which are not needed in this context according to our analysis of popular UML modelling tools. For the same reason, our proposed metamodel only supports conjuntive filters and not disjunctive filters.

5 Related Work

Navigation is an important mechanism to traverse, search, and retrieve information. Many studies have been done on how to improve navigation in software applications and web sites.

dos Santos et al. [7] investigate the effects of different types of menus in web site navigation, assessing the usability as well as performance of 8 different navigation mechanisms, each with distinctive properties. The study concludes by putting forward a horizontal menu, which is the base structure of the navigation bar presented in this paper. Burrel and Sodan [17] analyze six different types of menus contained in web pages of institutions. Considering the factors layout, ease of use, clarity of information, and ease of learning, they determine that navigation consisting of tabs, side navigation bars at the top and vertical menus on the left are the most favourite. We considered these insights when developing the navigation bar proposed in this paper. Finally, Muneo Kitajima et al. [18] present *CoLiDeS* (Comprehension-based Linked model of Deliberate Search), which is a model-based design methodology that website developers can follow to design better navigation for webpages. The main objective is to improve the user's success rate while searching for information on typical web sites.

To the best of our knowledge, there has been no prior work specifically on navigation for graphical modelling tools. Programming IDEs typically offer contextual menus that allow a developer to navigate within and across source code modules, e.g., from a method call to the method declaration. These relationships are typically inferred from static source code analysis. The following works target advanced navigation in programming IDEs, and as such can also be applied for navigation in textual modelling languages.

Mylyn is a task and application lifecycle management (ALM) framework for the Eclipse IDE [19,20]. In Mylyn, a developer can define tasks and declare which

tasks he is currently working on. Mylyn then keeps track of code elements that are being looked at, created, or modified for each task. The developer can then use this information for task-based navigation.

Similarly, the FEAT plugin for Eclipse [21] allows the developer to define a high-level conceptual unit called *concern*, e.g., a feature, a nonfunctional requirement, a design idiom, or an implementation mechanism. When coding, a developer can deliberately associate code elements to the concern, slowly building up a concern graph that relates code elements that are scattered throughout multiple source code modules. Subsequently, the developer can use the concern graph for highlighting and navigation purposes.

6 Conclusion

Model-driven engineering is a conceptual development framework where models of the system under development are created and manipulated using different formalisms at different levels of abstraction. Separation of concerns is further promoted when working with multi-view modelling, software product lines, and domain-specific modelling languages. While this separation into many interrelated models has many benefits, it also makes it harder for the developer to determine the relevant context when looking at a model, and to navigate from one model to related ones.

We propose a metamodel that covers two categories of navigation, intra-language and inter-language navigation. The metamodel allows the designer of a modelling tool to generically capture the relevant navigation links between model elements in a set of models manipulated for a given purpose. It is done by establishing inter-language and intra-language mappings designating the relevant metaclasses and references in the metamodels of the involved languages. We illustrate the effectiveness of our navigation metamodel by examples that involve feature models, class diagrams, and sequence diagrams, but our approach can be applied to any modelling language that is defined by a metamodel.

We furthermore show how this generic information can be used to visualize the current context of a model with a navigation bar, and how to populate the navigation bar with navigation links. When a navigation link is clicked, we either highlight the chosen model element if that element is located in the current model, or we navigate to the model that contains the model element and update the navigation bar to reflect the new context.

We validate that our generic navigation approach covers the navigation facilities provided by current modelling tools by conducting a survey of 6 popular UML modelling tools.

Our approach is not tool specific and can be applied to any language and modelling environment that uses metamodels. The main benefit is that if a modelling environment adopts our generic navigation approach, setting up navigation when adding a new language to an environment becomes greatly simplified. In that case, language designers do not have to implement intra-navigation support from scratch during language design, but can customize the navigation bar simply by

creating the appropriate intra-language mappings. To link the new models with models expressed in other languages already supported by the modelling environment, the corresponding inter-language mappings must be defined. With the increased adoption of Domain-Specific Languages (DSLs), this approach gives language designers essential support to rapidly define navigation within models expressed in the DSL as well as across model boundaries.

As future work, we are planning to examine the navigation facilities of non-UML modelling tools to ensure that our generic navigation approach can cover them. Furthermore, we will carry out an empirical user study to evaluate the usability of the navigation facilities offered by our navigation bar. Finally, we are planning to integrate our current navigation bar implementation with a modelling tool that supports language plug-ins.

References

1. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan & Claypool Publishers, San Rafael (2012)
2. Pfeiffer, R.-H., Wąsowski, A.: TexMo: a multi-language development environment. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 178–193. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31491-9_15
3. Di Ruscio, D., Lämmel, R., Pierantonio, A.: Automated co-evolution of GMF editor models. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 143–162. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19440-5_9
4. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: JTL: a bidirectional and change propagating transformation language. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 183–202. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19440-5_11
5. Beard, D.V., II, J.Q.W.: Navigational techniques to improve the display of large two-dimensional spaces. Behav. Inf. Technol. **9**(6), 451–466 (1990)
6. Mackinlay, J.D., Robertson, G.G., Card, S.K.: The perspective wall: detail and context smoothly integrated. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 173–176. ACM (1991)
7. dos Santos, E.P., de Lara, S., Watanabe, W.M., Fortes, R.P., et al.: Usability evaluation of horizontal navigation bar with drop-down menus by middle aged adults. In: Design of Communication Conference, pp. 145–150. ACM (2011)
8. Combemale, B., DeAntoni, J., Baudry, B., France, R.B., Jézéquel, J., Gray, J.: Globalizing modeling languages. IEEE Comput. **47**(6), 68–71 (2014). https://doi.org/10.1109/MC.2014.147
9. Pohl, K., Böckle, G., van Der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Heidelberg (2005). https://doi.org/10.1007/3-540-28901-1
10. OMG: Unified Modeling Language, 2.5.1, p. 802 (2007)
11. TouchCORE (2018). http://touchcore.cs.mcgill.ca/
12. ArgoUML - Free, opensource UML engineering tool. http://argouml.tigris.org/index.html
13. StarUML. http://staruml.io/

14. No Magic Inc.: MagicDraw. https://www.nomagic.com/products/magicdraw
15. Ideal Modeling & Diagramming Tool for Agile Team Collaboration. https://www.visual-paradigm.com/
16. Alam, O., Kienzle, J., Mussbacher, G.: Concern-oriented software design. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) MODELS 2013. LNCS, vol. 8107, pp. 604–621. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41533-3_37
17. Burrell, A., Sodan, A.C.: Web interface navigation design: which style of navigation-link menus do users prefer? In: Proceedings of the 22nd International Conference on Data Engineering Workshops, pp. 42–42. IEEE (2006)
18. Kitajima, M., Blackmon, M.H., Polson, P.G.: A comprehension-based model of web navigation and its application to web usability analysis. In: McDonald, S., Waern, Y., Cockton, G. (eds.) People and Computers XIV-Usability or Else!, pp. 357–373. Springer, London (2000). https://doi.org/10.1007/978-1-4471-0515-2_24
19. Kersten, M., Murphy, G.C.: Using task context to improve programmer productivity. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 1–11. ACM (2006)
20. EMF Website. Mylyn. https://www.eclipse.org/mylyn/
21. Robillard, M.P., Murphy, G.C.: Representing concerns in source code. ACM Trans. Softw. Eng. Methodol. **16**(1), 3 (2007)