

AN IMPLEMENTATION OF DEEP-CONNECTIONS FOR MULTI-LEVEL  
MODELING

BY  
CHANG XICHENG

A thesis

submitted to the Victoria University of  
Wellington in fulfilment of the requirements for  
the degree of Master of Engineering

Victoria University of Wellington

(2016)

Student: Chang Xicheng

Supervisor: Thomas Kühne

School: School of Engineering and Computer Science

## Abstract

Traditional object-oriented programming languages only support two logical domain classification levels, i.e. classes and objects. However, if the problem involves more than two classification levels, then to model a multi-level scenario within two classification levels, a mapping approach is required which introduces accidental complexity and destroys the desirable property of “direct mapping”. Therefore “Multi-level modeling” was proposed. It supports an unbounded number of classification levels, that can support “direct mapping” without introducing accidental complexity. Many supporting features have been proposed for “multi-level” modeling such as “deep instantiation”, potency, clabjects, etc. To date most of the research effort was focusing on the entities (clabjects), while the relationships between entities were receiving much less attention and remained under-explored.

The “Melanee” tool was developed to support multi-level modeling both for academics and practitioners. “Melanee” supports an unbounded number of classification levels for domain modeling and it treats relationships like clabjects. It mainly supports “constructive modeling” by creating models using a “top-down” approach, whereas “explanatory modeling”, which is creating models using “bottom-up” approach, is not well supported and lacks support to ensure the integrity of the created models. Hence, to further explore relationships in multi-level modeling and to provide a better modeling environment, there are two main focuses in this thesis: First, based on existing, I further explore relationships between entities and extend the LML (Level Agnostic Modeling Language) supported by Melanee accordingly. Second, I extend Melanee’s functionality to support “explanatory modeling”.

Considering that Melanee is an open source tool I first discuss Melanee’s structure and its principles in order contribute to future extensions to Melanee. The knowledge of Melanee is currently known by its principle developer, Ralph Gerbig, with whom I had contacts in the beginning phase of the “deep-connection” development for advices. Next

I use the work proposed in the paper “A Unifying Approach to Connections for Multi-Level Modeling” by Atkinson et al. as a foundation and stepping stone, to further explore relationships between entities. I extended Melanee to support the “Deep-connections” feature by adding potency to connections and their monikers, and further allow connections to have “deep-multiplicities”. I developed these features, as well as respective validation functions to ensure the well-formedness of models.

Then I extended LML so that user-specified type names can be used to indicate the names of types for clabjects. Instead of relying on modelers to fully manually define type-of classification relations between different levels, I introduce “connection conformance” and “entity conformance” to introduce classification support to Melanee. Potentially matching types are calculated and ordered per their matching scores. Respective suggestions to modelers including messages for each possible matching type about how to fix the current connection instance so that it matches the potential type whenever applicable. The suggestions are made available as so-called “quick-fixes” and I extended this approach with a second-stage dialog that allows modelers to select amongst many fix alternatives. Finally, I evaluate my design using model sets taken from existing papers and a systematic exploration involving 57 different scenarios.

Keywords: Multi-level modeling, metamodeling, exploratory modeling, deep-connection, Melanee, clabject, deep-multiplicities

## Acknowledgements

I would like to express my sincere gratitude to my supervisor Associate Prof. Thomas Kühne for the continuous support of my Master study and research.

My sincerest thanks also go to Dr. Peter Andrews and Dr. Alex Potanin for giving me advices and help when I was confused and lost. They helped me kindly and patiently in the process. Thank all of them for the strength they put in me in the Master Study.

Special thanks for Dr. Ma Hui, who lit a light in my darkest days, and showed me the path which I can follow to this end. Her support and advices helped me in the process.



# Contents

1 Introduction .....	9
2 Background .....	13
2.1 What is Multi-Level Modeling? .....	13
2.2 Related Notions .....	16
2.2.1 Deep Instantiation and Potency .....	17
2.2.2 Clabjects .....	18
2.2.3 Entity and Connection .....	18
2.2.4 Level Conformance and Model Validity .....	19
2.2.5 Entity Conformance .....	20
2.2.6 Connection Conformance .....	20
2.2.7 Deep-Connection Feature .....	20
2.3 Melanee .....	23
2.3.1 Eclipse .....	24
2.3.2 Eclipse Plugin .....	24
2.3.3 EMF .....	26
2.3.4 GMF .....	27
2.3.5 Melanee Validations .....	30
2.3.6 Level Agnostic Modeling Language (LML) .....	31
3. Design .....	33
3.1 Requirements .....	33
3.2 Design .....	35
3.2.1 Connection Conformance Design .....	35
3.2.2 Entity Conformance Design .....	63
4 Implementation .....	67
4.1 Deep Connection Implementation .....	69
4.1.1 Adding Feature to PLM.Ecore File .....	71
4.1.2 Connection Initialization Customization .....	73
4.1.3 Adding Multiplicity Control Tab in Properties View .....	78
4.1.4 Rendering View of Deep-Connection .....	81

4.1.5 Deep-Connection Validations .....	88
4.2 Connection Conformance .....	90
4.2.1 userSpecifiedTypeName .....	91
4.2.2 Calculating the Matching Score (Weighted Sum) .....	94
4.2.3 Connection Conformance Critique Development.....	102
4.2.4 Second-Stage Dialog .....	105
4.2.5 Other Validations .....	110
4.3 Entity Conformance .....	110
5 Evaluation .....	113
5.1 Deep-Connection Implementation Evaluation .....	113
5.1.1 Moniker and Potency Rendering.....	115
5.1.2 Deep Multiplicity Support .....	116
5.1.3 Validations Related to Deep-Connection .....	116
5.1.4 Evaluation of Moniker Potency Validation and New Multiplicity Validation ..	125
5.2 Connection Conformance Evaluation.....	127
5.2.1 userSpecifiedTypeName .....	128
5.2.2 Weight Calculation Functions Evaluation.....	128
5.2.3 Quick-Fix for Connection Conformance .....	149
5.2.4 Other Validations of Connection Conformance.....	151
5.3 Entity Conformance .....	151
5.4 Summary.....	152
6 Future Work .....	153
7 Conclusion .....	155
Reference.....	157



# 1 Introduction

Traditional object-oriented metamodeling is getting attention with respect to modern software development [11]. However, just like object-oriented programming languages, standard object-oriented metamodeling supports only two logical domain classification levels [1]. This, however, means that “direct mapping” cannot be obtained if the problem domain involves more than two classification levels [1]. “Accidental complexity” [2] is introduced to solve this problem. Several solutions are compared in [2] with respect to how much they can reduce accidental complexity. It turns out that, multi-level modeling – modeling that goes beyond two classification levels [6] –, can significantly reduce the accidental complexity of domain models. A consequence of multi-level modelling is that a modelling entity is both “instance” and “type” at the same time. Hence, the notion “clabjects” was proposed in [3]. Unlike regular classes, “clabjects” may have an arbitrary instantiation depth, i.e., their instances may have instances themselves, etc. In traditional “shallow” object-oriented metamodeling, types can only describe instances on the immediate level below [1]. The multi-level instantiation capability of “clabjects” is referred to as “Deep Instantiation” [1]. By introducing “potency” (a non-negative integer value), “Deep Instantiation” can be limited how many levels deep a clabject may be instantiated [4]. Traditional two-level programming languages without built-in notion of “Deep Instantiation” offers the ability to focus on a particular problem domain [16], and most data structures and algorithms can be implemented in DSL [17]. But traditional two-level metamodeling has problems representing multi-level-domain [2].

Melanee is an open source multi-level modeling tool developed by University of Mannheim [13] that supports “Deep Instantiation”. It is a proof-of-concept research tool for multi-level modeling as well as a usable multi-level tool that can be used for practical applications. Melanee uses the LML language (The **L**evel-agnostic **M**odeling **L**anguage) to define the PLM (the **P**an **L**evel **M**odel) [5].

Most of the research around Melanee and its implementation have been focusing on entities. The relationships between entities have been under-explored with respect to multi-level modeling in both research and implementation [7]. Most of the papers in multi-level research either did not discuss relationships at all or simply treated them as clabjects in [6] and [1]. Recently, Colin Atkinson, Ralph Gerbig and Thomas Kühne published work in which connections in multi-level metamodeling are further explored [7]. In this thesis, I will use the research conducted in [7] as a basis to explore adequate support of relationships in multi-level modelling with a view on implementing the respective concepts and ideas in Melanee. Following [7], I will refer to “relationships” as “connections” in the rest of the thesis.

Melanee hitherto mainly supported a “constructive” modeling, which is considering that a type model in higher level to be valid and consistent, and all instances in lower level are created according to their types [15]. This constructive method assumes that the type models are always correct, and type models need to be created first considering their instances are defined according to them in the system. This “constructive” modeling needs modelers to create models from topmost levels first, so that the instances can be create from this defined types, a “top-down” approach. However, sometimes types are not predefined, but extracted from the instances defined in lower levels. In other words, sometimes models are created from lower levels first, in a “bottom-up” approach. This approach is also called “exploratory” modeling, in which types are created from instances, after all instances are properly defined [15].

In the “exploratory” modeling, the types are derived from instances, and the derivation process are performed by developers or modelers [15]. Hence, to support “explanatory modeling”, validations to ensure the validity of the model and type matching information should be supported by Melanee to avoid unintentional errors or mistakes made by human element in the modeling process. So, that “clabjects” in each level are conformed (entity conformance and connection conformance) with their types in upper level to achieve “Level Conformance” [3].

Hence in this thesis, the LML and PLM in Melanee are extended so that Melanee can support deep instantiation for connections. To this end, connectionends are extended to have the ability to have monikers (connection end names) for supporting deep instantiation, but no moniker type is shown in the system. Furthermore, deep multiplicities to connections in Melanee are implemented, so that multiplicities can be described from higher up than the immediate type level above to achieve “deep- multiplicities”. Besides deep-connection features, the support for “exploratory” modeling has been implemented and validated using experiments in Melanee to support “bottom-up” construction of models. Certain features like allowing modelers specify type model names before type model defined, validations for insuring validity of the whole model, and the ability of providing type matching information are implemented to support the “exploratory” way.



## 2 Background

In this chapter, some background information and definitions of related concepts and notions in multi-level modeling are discussed. The “Melanee” tool used to implement deep-connections and “explanatory” modeling will be introduced and so are the LML (Level-agnostic Modeling Language) and PLM (Pan Level Model) that Melanee supports are discussed for extension purpose.

### 2.1 What is Multi-Level Modeling?

Before entering the modeling world, it is necessary to figure out the key question of “What is multi-level modeling”. Multi-level modeling is an extension of standard modeling, and it can reduce complexity in the models by introducing as many modeling levels as it needs. According to the definition of [5], the definition of multilevel-modeling is:

*multi-level modeling covers any approach to modeling that aims to provide systematic support for representing multiple (i.e. more than two) ontological classification levels within a single body of model content.*

To understand the definition presented above, it is necessary to distinguish between classification and generalization in modeling. Classification and generalization can both be regarded as abstraction mechanisms and they are all mechanisms to abstract common characteristics from numerous elements for reducing complexity of specifications [9]. Both classification and generalization aim at reducing complexity, but they are different in nature and cannot be replaced by one another:

*“Classification is used to create types from instances, giving rise to type models, abstracting away from the identity of instances and their different property values” [8].*

The type-instance relationship is known from object-oriented programming languages. Let me take a Java class as an example: “Car” can be viewed as an abstraction of many “car objects”, and so it carries certain attributes like “name”, “PRP” etc. The instance “myCar ” is an instance of “Car”, and can be created by using the type “Car” as follows:

```
Car myCar = new Car (“My Car”);
```

This is a typical classification example, and typical object-oriented programming languages have two levels:

1. A type level, in which the types reside; and
2. an instance level, in which the instances reside.

Entities in each level only have one role, as types or as instances, but never as a type and as an instance simultaneously.

Generalization, however, is:

*Generalization is used to create genus (hypernyms) from species (hyponyms), for example creating supertypes from subtypes [10].*

Even though generalization is also an abstraction, it is different from classification by that it forms super type/subtype relationships instead of type-instance relationship [9]. For example, the regarding the models depicted in Fig.1, “Car” on the right-hand side is a Java class and it has a subtype which is “Sports Car”, and they are both types, so they do not form a type-instance relationship, as opposed to a classification relationship in the left. A super type is a generalization of its subtypes; it does not classify them [9].

Fig. 2 is a 3D diagram to illustrate the difference of transitivity between classification and generalization: “myCar” is a member of the super type (Car) of its type (SportsCar) in (a), but not a member of the type of its type (CarModel) in (b).

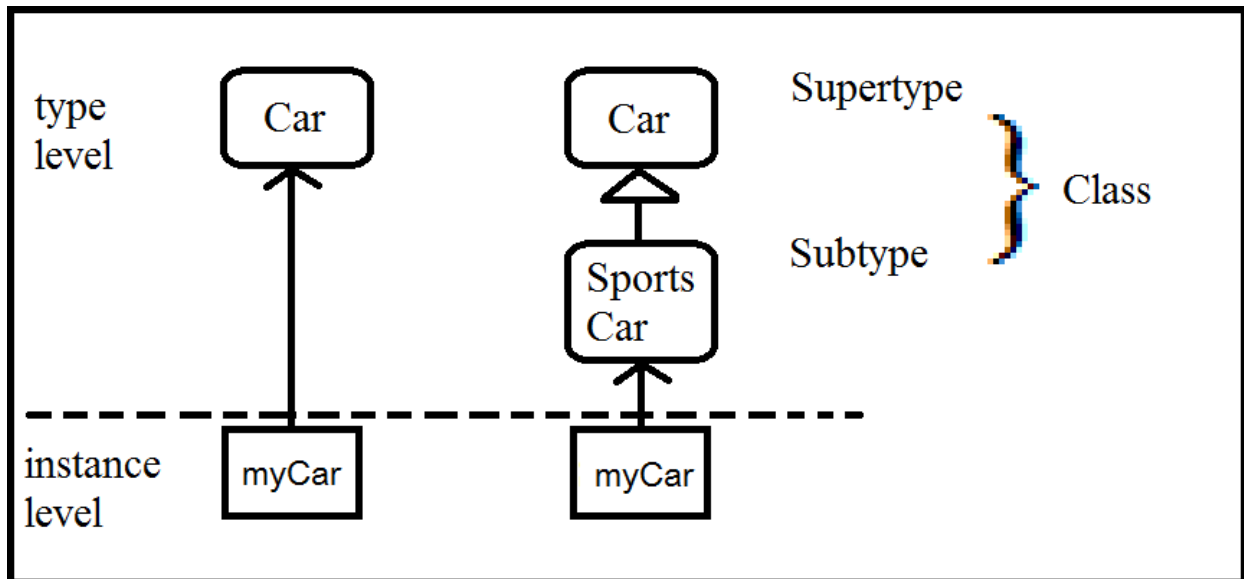


Fig. 1 Differences between type-instance relationship and sub/ super type relationship.

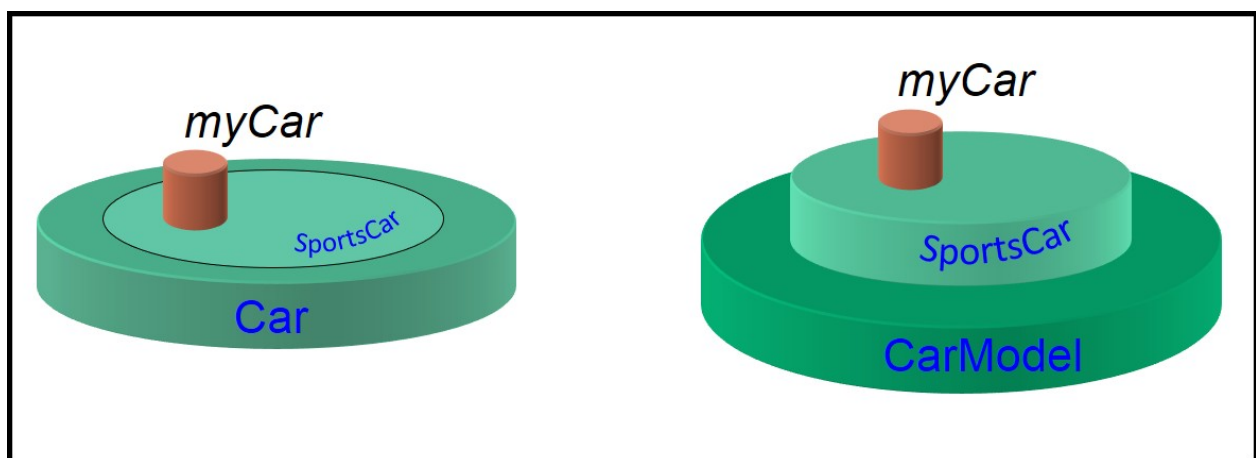


Fig. 2 Differences in transitivity.

The above definitions in multi-level modeling are concise but not detailed enough, here I adopt a more comprehensive description of description: Multi-level modeling, or so-called deep-modeling:

- mirrors classification relationships in the domain with explicit relationships that are subject to well-formedness constraints.*
- recognizes type and instance facets of model elements and views them as inseparable.*
- provides a mechanism for deep characterization [11].*

Multi-level modeling can reduce “accidental complexity” in models [2]. According to the discussions performed in [2], strategies similar to “Description Patter” are usually employed in situations where new types need to be introduced dynamically. The root of the problem is that additional work needs to be done to represent “instance-of” relationship within a model [2]. Multi-level modeling goes beyond traditional “two level” modelling and solves the ‘level mismatch” problem in modeling [2]. By applying the notion “clabject” and simple “instance-of” relationship, less accidental complexity is introduced. According to [2], the workarounds need to be implemented to represent the “instance of” relationship in traditional modeling scenario are removed in multi-level modeling.

## 2.2 Related Notions

In this subchapter, some the core features found in multi-level modelling (and Melanee) are introduced and discussed.



### 2.2.1 Deep Instantiation and Potency

In traditional two-level modeling, like object-oriented programming languages, the type only has to describe instances in one level below, and this mechanism is called “shallow- characterization” [1]. In multi-level modeling, it is necessary to describe instances in far beyond one immediate level below, and the description cross more than one level boundary will be the responsibility of modelers. Relying on human to implement such feature is not reliable, especially in large models, errors or mistakes can be made unintentionally. Hence a more systematic way of describing instances in many levels below is required. “Deep Instantiation”, which allowing the description of instances beyond immediate one level below, can solve this problem [1]. The definition of “Deep Instantiation” is:

*Deep instantiation conservatively extends traditional shallow instantiation by letting potency values range over integer values  $\{0, \dots, n\}$  where  $n$  corresponds to the maximum characterization depth required [3].*

The ability of allowing description of instances to be specified from two level above is called achieving “Deep Instantiation” [1]. In order to achieve Deep Instantiation in multi-level modeling, notion “Potency” is proposed [1].

Potency [1] can be owned by both fields and objects, to indicate how many levels the type can describe its instances. Technically speaking it a non-negative value. For instance, the example shown in Fig. 3, attribute “RRP” with potency 2 can be instantiated by elements in as far as two levels below, but not further. Attribute “designer” with potency 1 can only be instantiated by elements in one level below.

### 2.2.2 Clabjects

In multi-level modeling, molders can specify arbitrary number of levels, hence there is a question regarding the role of the elements in each level: the elements in the middle level can be instance and type simultaneously. Hence notion “clabject” is proposed to solve this problem [3]. Clabjects have two facets at the same time [3], like shown in Fig. 4. On the left-hand side is the instance view, this facet contains attribute instances and method instances described in the type; on the right-hand side is the type view which describes attribute and method types that its instance will instantiate. Both entities and connections can be viewed as clabjects.

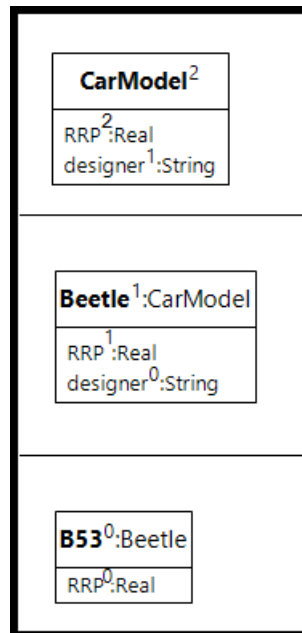


Fig. 3 Different potency values describe different levels below.

### 2.2.3 Entity and Connection

Entity is a subtype of clabject in Melanee PLM (the Pan Level Model), and entity adds no extra definitions to clabject. Its sole purpose is to have a name for concrete clabjects that are not connections [5]. The elements shown in Fig. 3 are all entities.

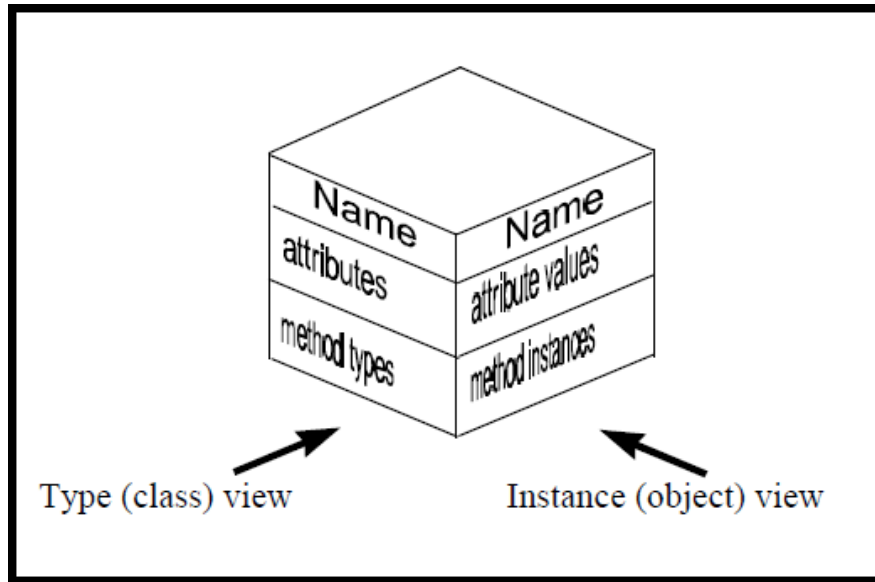


Fig. 4 3D visualization of a clabject, this picture is taken from [3].

Connection is also a subtype of clabject in Melanee. Connections in Melanee are representing relationships between entities, and they can also have classification and generalization relationships. Melanee hitherto mainly supports binary connections (connections that connects to 2 entities).

#### 2.2.4 Level Conformance and Model Validity

In each of the ontological classification levels, there can be numerous number of entities and connections. Using a strict metamodeling approach, each level is an instance of the level above (except for the top level) [3]. Tools like UML [18] adopt type-instance relationship of models in a loose way [3]. In order to have a level conformance, it is necessary for all the entities and connections in each level to conform to corresponding entities and connections in the levels above. Here I adopt a multi-level framework based on the doctrine of strict metamodeling [12] -

*Every element of an  $M_n$  level model is an instance\_of exactly one element of an  $M_{n+1}$  level model [12].*

Except for the topmost levels which has no “type” level for them, all other levels should form conformance to archive the validity of the whole model.

The goal of achieving “Level Conformance” can be divided to into two smaller goals: achieving “**Entity Conformance**” and “**Connection Conformance**” [7].

#### 2.2.5 Entity Conformance

Achieving entity conformance means that all the entity instances can be described by their corresponding types in one level above in multi-level environment. The attributes defined in the entity instances comply with what is defined in the types, e.g. potency in the instance is smaller than that of its type. The entity instances satisfy all constraints defined by their types.

#### 2.2.6 Connection Conformance

Connection conformance means that all connection instances must satisfy all constraints defined by their respective types in one level above [7]. Achieving connection conformance indicates that all connection instances can match with the description of their types with no violation of any kind, e.g. connection name potency should not be larger than that of its type.

#### 2.2.7 Deep-Connection Feature

Most of the research focus are concentrated on the “entities” in models. However, in order to achieve level conformance in models, not only the entities need to be conformed with their corresponding types, but also the connections need to be conformed to their types as well. In most approaches relationships are treated as “clabjects” [1], [6], [22], but some

researchers [5], [7] and [23] gave more consideration exploring the feature of relationships. And “A Unifying Connections Design” was proposed in [7] by Kuehne et al.

The detail of “A Unifying Connections Design” is shown in Fig 5. In the paper terminologies are discussed and a conceptual model was proposed to implement deep-connections feature. According to [7], three conceptual models were compared and discussed, and the composite model is the best for implementing deep connections [7].

In [7], connections are regarded as clabjects so they can have potency value, attributes and method as well. Each connection must conform with its type in upper level to comply with strict meta-modeling notions. Each connection also has two “connectionends” as part of the connections [7], and “connectionends” are dependent on connections: they cannot be created or removed independently for they are part of the deep-connection design [7]. An example of a deep-connection is shown in Fig. 6.

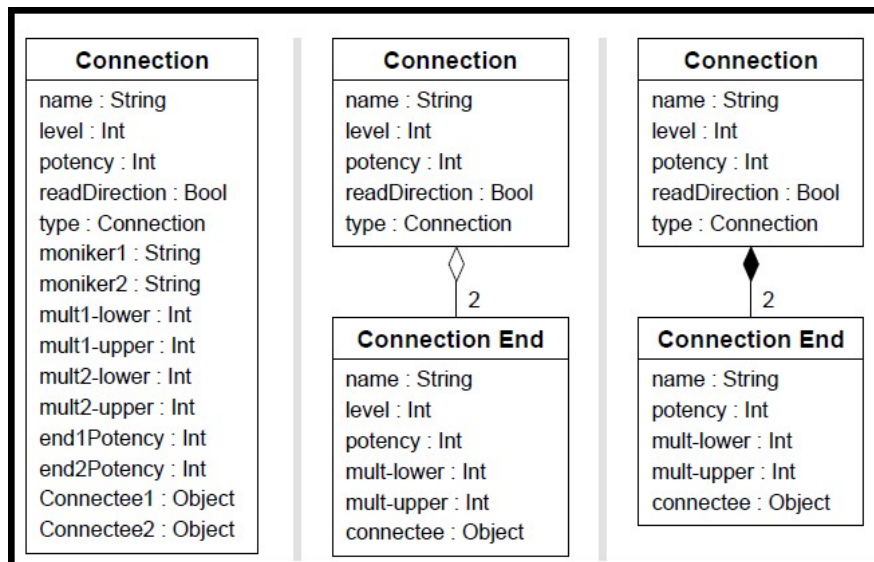


Fig. 5 Three conceptual model for deep-connections taken from [7].

The description of connectionends in [7] are:

*Connectionend name is referred as moniker, this is because they are essentially origin-specific aliases to entities that may be known through other names to other entities [7].*

Connections can also have “deep multiplicities”. Traditional multiplicity affects the immediate level below by defining boundaries of how many connections instances there should be. With deep multiplicities, modelers can specify many multiplicities with different potency value so that the number boundary of connections instances not only the level immediate below but as many levels as it is specified to be can be defined.

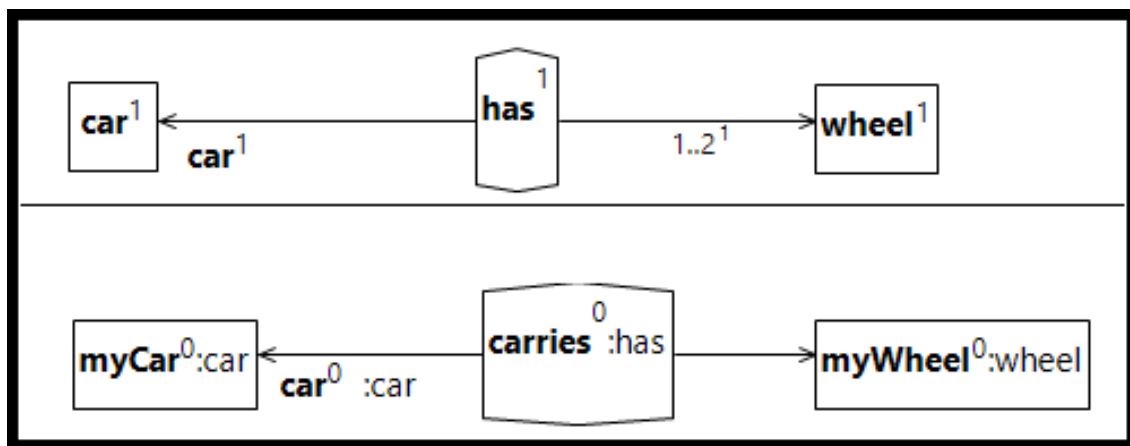


Fig. 6 Two deep connections example: connection “carries” conforms to connection “has”.

In [7] there are also discussions related how to identify connections using the “five components” to achieve connection conformance. Details related to the “five components”, connection identification and connection conformance will be described in Design Chapters of the thesis.

## 2.3 Melanee

*Melanee is a workbench for creating domain-specific languages which occupy an arbitrary number of ontological levels [13].*

It is an open-source software. In this thesis, Melanee is adopted and used to implement deep-connection, level conformance (including entity conformance and connection conformance) to support “explanatory” modeling in Melanee. Melanee is built on Eclipse-platform and GMF frameworks, and it is difficult to locate proper documents or tutorials in depth. The structure and code of Melanee are complex and lacks of proper comments to explain the code. And considering it is an open-source software, here I will introduce and explain Melanee’s structure and the key code in certain plugins so that it will be much easier for modelers and developers to further contribute to Melanee without having to spend a lot of time to get to know the structure and its details.

Melanee is built on the Eclipse platform and GMF (Graphical modeling framework) framework. It is split into multiple components; the introduction of compotes is from Melanee website:

*Melanee-core: The foundation of the deep-modeling environment. Provides Meta-model, GMF Editor, Emendation Service, Proximity Indication Service, Visualization Definition Editor, Autolayout [13].*

*Melanee-graphdsl: Graphical domain-specific language capabilities of Melanee. Allows to model graphical domain-specific language notations and use these embedded into the default Melanee graphical editor [13].*

These components are the ones that relate to the content of this thesis, and there are other components in Melanee as well, but they are unrelated to this thesis hence they will not be discussed in this paper.

Melanee is designed to support multi-level modeling. Modelers can specify as many levels as it is needed, and from the palette on the right operations like new entity adding, drawing connections and other operations can be chosen. Changing properties of a specific element can be performed by selecting the element and conduct changes in properties sheet. An example of Melanee modeling user interface is shown in Fig. 7.

In the following of this chapter, technologies and the platforms Melanee is built on will be introduced and discussed.

### 2.3.1 Eclipse

Eclipse is an open-source, Java based, extendable platform. Eclipse itself is just framework and a series of services, and its IDE (Integrated Development Environment) is constructed through plugins. Many different plugins contribute to what Eclipse and how it works today.

### 2.3.2 Eclipse Plugin

Every function in Eclipse platform is constructed by numerous plugins. Each plugin can play two roles at the same time: user of other plugins and service provider to other plugins.

The detailed introduction of eclipse plugin is not the focus of this thesis, but in Implementation Chapter the more relevant detail will be covered.



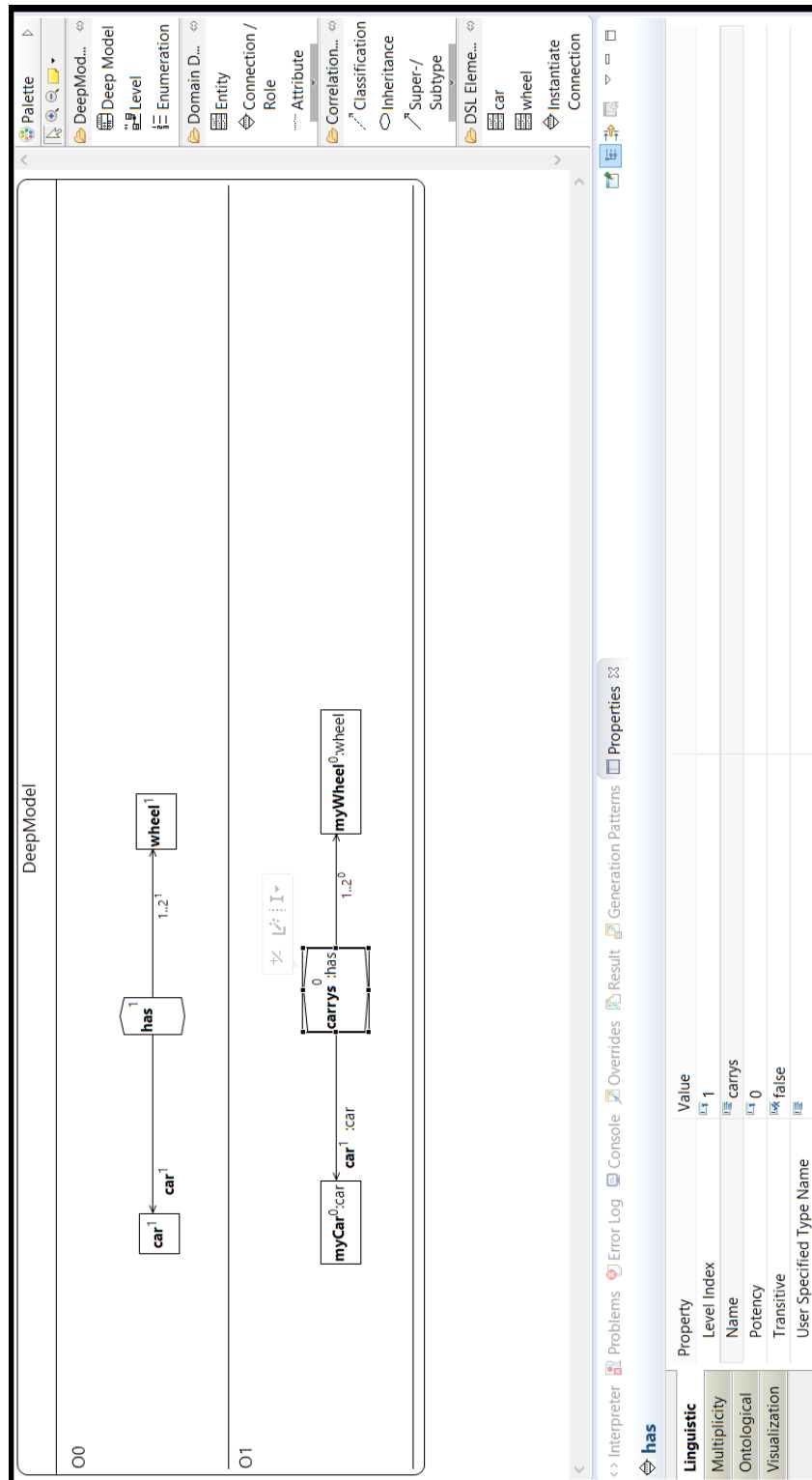


Fig. 7 Melanee modeling environment.

### 2.3.3 EMF

Eclipse Modeling Framework (EMF) is a framework and code generation provider based on Eclipse platform for building modeling tools [24]. The main function of EMF is to hold the meta-model for Melanee to generate corresponding model code and interfaces for model manipulation. There are two main files in EMF of Melanee that are relevant to development in this thesis.

PLM.ecore file stores the meta-model for Melanee. It allows expressing models by defining meta-models to describe what models the modeling tool can support. Melanee defines its meta-model in PLM.ecore file in plugin “org.melanee.core.models.plm” under the model folder. Modifying meta-model of Melanee, for example adding new features in the model, needs to be performed in this file. OCL functions can be defined in the meta-model in PLM.ecore file, and these OCL functions can be called later in other OCL supported environment such as in validation script, or in Acceleo.

The file PLM.gnmodel is generated per the meta-models defined in PLM.ecore. File PLM.gnmodel is for generating corresponding code for model manipulation, for example the model initialization code when creating a clabject. Generating code instead of being hand-coded allows programmers to focus more on logic instead of implementation details. Generating code can provide straightforward way of implementing model related features, such as adding new component or modifying attribute values of an element. There are three categories of code which are relevant to this thesis:

1. Model code: three packages will be generated under src: package “org.melanee.core.models.plm.PLM”, “org.melanee.core.models.plm.PLM.impl” and “org.melanee.core.models.plm.PLM.util”. package “org.melanee.core.models.plm.PLM” contains a series of interfaces for accessing attributes of meta-model, among them there are two important interfaces: interface “PLMFactory” and interface “PLMPackage”. EMF uses “Factory Pattern” and the

interface “PLMFactory” is for creating elements defined in the meta-model. Customized code could be written into PLMFactory to achieve customized feature in model creation. Customization of generated code could allow programmers add features that are not provided by EMF default generated code.

In package “org.melanee.core.models.plm.PLM.util” there are two files: class “PLMAdapterFactory” and “PLMSwitch”. Class “PLMAdapterFactory” is for event listening. When a model changes, the view must be refreshed accordingly. A model could have multiple views, and EMF uses the Observer Pattern, and observers use adapter class to notify changes to refresh the visuals of the models.

2. Edit code: The main function of edit code is to access model definitions and prepare the models for visualization. It converts model data and provide them to editor code. It is like a layer between model and editor code.
3. Editor code: it is about visualization representation of model, based on Factory Pattern it can acquire different adapter per what kind of change has been made, and the adapter links EMF and model events together when a change is made to the model. When editing graphically, “Commands” are used to perform the actions.

#### 2.3.4 GMF

The Graphical Modeling Framework (GMF) is a framework based on Eclipse platform [26]. It provides a generative component and infrastructures for developing graphical editors based on the EMF and GEF [26]. Graphical Editing Framework (GEF) is an Eclipse Graphical Editing Framework [25]. It and EMF together can provide graphical model editing function. GMF is based on EMF and GEF, and GMF provides GEF functions more conveniently.

Melanee is built mainly on GMF and GMF glued EMF and GMF together by providing configuration files to simplify the process of development. These configuration files can generate corresponding GEF code.

PLM.gmfgraph: this file is used to define the graphical elements for a domain model. For example, it can define what shape to use to present the entity. It can also define labels required to show relevant information to modelers, such as showing the name of an entity and potency value. It can also define features like what position the label will be shown at and what font to use in the graph. This file is the place to define appearances of model elements.

PLM.gmftool : this file is used to define the palette of tools that you can use in the graphical editor. Like the example shown in Fig. 7, the palette shown on the right side. The elements inside the palette are defined in this file.

PLM.gmfmap: this file links together the domain model, the graphical model (PLM.gmfgraph) and the tooling model (PLM.gmftool). It is the glue between all other definition models: it maps EObjects with Graphical Elements and tools; and it is the file that defines the possible relationships, like connections in the editor. The labels defined in PLM.gmfgraph need to be linked to the graph in PLM.gmfmap. Not only is PLM.gmfmap able to link things together, but also it can define value of attributes. PLM.gmfmap also supports “Expression Label” that can set values for certain attributes using OCL expression. For example, some connection related features are defined in PLM.gmfmap file in Melanee, e.g. compute the connection’ type name.

PLM.gmfge: this file is used to generate the GMF graphical editor in addition to the EMF code generated by the PLM.genmodel file. In Melanee this file can generate EditPart files, commands, providers and all other relevant files under plugin

“org.melanee.core.modeleditor”. These generated files can also be customized to support more features. The changes must be written into templates under template folder in plugin “org.melanee.core.models.gmf” to make sure the customized code won’t be lost in future generation of GMF code using these configuration files.

These configuration files are key parts of GMF, many of the changing and adding of new elements are done in these files. However, what the GMF generates is fundamental code that provide basic functions only. There are two kinds of generated files that are crucial for customizing code to provide more functionalities.

The first kind of generated files are the commands. They are placed under folder “org.melanee.core.modeleditor.edit.commands” in plugin “org.melanee.core.modeleditor”. As explained earlier every action related to graphically changes of models are performed through commands. Most of the commands under this folder are “createCommand”. This is the class stores code of model initialization during model creation. For example, the default potency value could be 0 or 1, and such configurations are performed here. And they can be customized.

Another category of files are the EditPart files. They are placed under folder “org.melanee.core.modeleditor.edit.parts” in plugin “org.melanee.core.modeleditor”. Classes called “EditPart” are building blocks, they are the controller, and they tie the meta-model to their visual representations. Classes “EditPart” are responsible for making changes to the model by using commands, for example the change of values of labels is done in EditPart. EditPart controls the layout and initialization of model visualization in the graph. When a change is made in the graph, the change will be passed to corresponding EditPart, and the EditPart will use commands to notify the models for the change and then refresh the graph. Representation of models includes Figures, TreeItems and so on. The code inside commands can also be customized.

### 2.3.5 Melanee Validations

Validations in Melanee are performed using Epsilon Validation Language in file `wellformedness.evl`, which is located in plugin `org.melanee.core.models.plm.validation` under validation folder. Eclipse offers several kinds of warnings with different critical levels, from highest critical level to the lowest, the warnings are: errors, warnings and information sign. High critical level warnings can override low level warnings in the graph, but the warning messages are preserved and shown in order according to critical levels of the errors. The most commonly used kind of warning is “constraint”. Constraint can produce errors if certain conditions defined in it are not met, for example the potency is smaller than -1, the type has not been defined for a clabject etc.

Each constraint is placed under a context, which the name of the context corresponds to the models defined in file `PLM.ecore`. “Guard” section in the constraint is pre-check to see if certain conditions are met or not. If the conditions in “guard” are met, then perform the “check” section. This is precondition check in the validation can prevent unexpected situations like the feature values are not defined. In the “check” section validations can be performed to see if the elements meet certain criteria or not. Returning true means it passed validation, and returning false means otherwise, an error sign will be shown inside the graph of the correspond element. A scenario of failing the validation and showing of the error sign in the graph is depicted in Fig. 8.

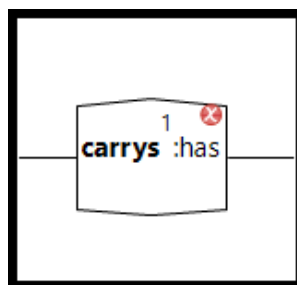


Fig. 8 Error for failing validation.

Epsilon Object Language (EOL) is supported in EVL script, hence it is possible to define operations that can be called in the context. According to [14],

*In typical object-oriented languages such as Java and C++, operations are defined inside classes and can be invoked on instances of those classes [14].*

It is possible to call a java method in EVL script. This is quite convenient to extend validations with rich functionality. The functionalities EVL provided are limited. For example, if a dialog needs to be prompted when an error occurred, this function needs to be implemented in Java code and be called in EVL, and simply using EVL this function cannot be implemented. In order to allow EVL to be able to call Java method, it is necessary to register written Java class in plugin.xml.

#### 2.3.6 Level Agnostic Modeling Language (LML)

Level Agnostic Modeling Language(LML) [5] is used by Melanee to render views of multi-level models. According to [5], The Pan Level Model (PLM) is the linguistic type model and defines all linguistic types that are used to represent ontologies [5]. Melanee's modelling notation is compatible with the UML language and supports the construction of models with more than two classification level without introducing accidental complexity [5], as it supports "clabjects" and "deep instantiation".

LML supports deep-instantiation by using potency to indicate how many levels down to describe the instances. LML represents PLM and render views of PLM [5].





### 3. Design

Since the main design for deep-connection has already been described in [7], in this Design chapter I only focus on design of entity conformance and connection conformance design.

#### 3.1 Requirements

Since an “explanatory” modeling is being introduced into Melanee, it is important that models can be constructed from lower levels to the upper levels, thus the conformance between the level and the level above needs to be guaranteed. This is because the original way of constructing models from “top-down” approach that always guarantees that the type clabjects, which includes entities and connections, in each level are the instances of the clabjects in one level above, except the ones in the topmost level, which contains clabjects that only have type facet. Thus, in original approach each clabject can be guaranteed to be constructed with their type already defined in upper levels. However, the “explanatory” modeling, in which models are created from lower levels first, when types are probably not defined yet in upper levels. When types are created from the instances, it is likely that some instances or partial of an instance are not conformed with the types, which will create inconsistencies and break the conformance between levels. Thus, to support an “explanatory” modeling, it is necessary that all clabject instances are checked against their possible types to achieve entity conformance and connection conformance, so that level conformance can be guaranteed.

To achieve level conformance in multi-level modeling supported in Melanee, and to ensure the ease-of-use of modeling tool, three requirements need to be met in LML extension and Melanee functions development.

The first requirement is that modelers should be notified with warnings if connections or entities have no types defined for them, and then calculate the penitential types and present the results to modelers. For each connection, there are five components according to [7], it is likely that not all the five components matched with a connection type at the same time. Hence along with the warnings, there are messages to modelers regarding which connection type the connection instance matched perfectly, if not all components matched, messages containing how to modify the instance to match the type need to be presented. To consider all possible scenarios, all connections in immediate level above will be acquired and calculated for each connection instance.

For entities, if the type has not been defined for them in all levels except the topmost level, warning regarding type undefined information will be presented to modelers as well. Since there is only one component in entities, the message will not be complex compared with those of the connections.

The second requirement is the ability to fix the type undefined issue to achieve conformance. Especially for connections, when not all the components matched with what is defined in the connection instance, defining type for such connection instance may involve modification of different component names and their potencies as well, which is complicate and errors could be made during the process. For ease of use of Melanee and providing support for multi-level modeling environment, an easier way of defining types for both connections and entities needs to be designed to avoid possible mistakes that could be made by modelers during model construction.

The third requirement is to provide comprehensive validations in the system to avoid inconsistencies between types and their respective instances in each level, so that the validity of the whole model can be guaranteed. There should be no errors in the whole model after connection and entity conformance is achieved. Hence validations against

possible errors for connections and entities are necessary and should be developed in Melanee.

## 3.2 Design

In Design Chapter, there are two sub chapters: Connection Conformance Design and Entity Conformance Design. They are in separate chapters for even though they are all clabjects, connection is quite different and more complex in structure. Connection Conformance Design will be introduced and discussed in the following sub chapter.

### 3.2.1 Connection Conformance Design

According to the requirements discussed above, the first issue needed to be solved is how to warn modelers of type undefined warnings in Melanee. Eclipse plug-in framework offers several types of warning, and in our project, here two kinds of warning is adopted here: critique and constraint.

*Constraints is used to capture critical errors and critiques are used to capture non-critical situations that do not invalidate the model, but should nevertheless be addressed by the user to enhance the quality of the model[14].*

“Critique” is perfect for implementing connection conformance and entity conformance for potential types calculation and definition in the model is not a critical error, but a reminder to modelers that there are clabjects which their types hasn’t been defined. “Constraint” is adopted for implementing validations in Melanee, for example the destinations’ type checking and connectionend type checking after connection’s type has been defined, for constraint can produce critical error that requires modelers’ attention. “Information” is not used for it is merely for presenting information with the lowest warning level that cannot attract enough attention of modelers.

Extending validation script can also provide the advantage of using “quick-fix” section provided by Eclipse plug-in, which allows defined fixes of the corresponding errors or warnings to be performed in the “Problems Properties View”. This quite suites our second requirement here: after calculating potential connection types, quick-fix can offer a more convenient way to allow user to define types for connections and entities, so it is unnecessary for modelers to go back to palette to choose “classification” to draw the line again, which reduces the likelihood of pointing wrong types for the connection instance, especially in large models.

The first requirement needs to be fulfilled which is how to decide whether a potential connection type best matched, partially matched or doesn’t match a connection instance at all, and reorder all possible matches in a list and present the list to modelers so that they can utilize the list full of potential types to fix the type issues for a connection instance. In other words, how to identify type so that only useful information and possible matching types will be provided to modelers instead of listing all the connection types in one upper level to Melanee users. To resolve this, the idea proposed in [7] is adopted of identifying connection types using “the Five components”, which is shown in Fig. 9.

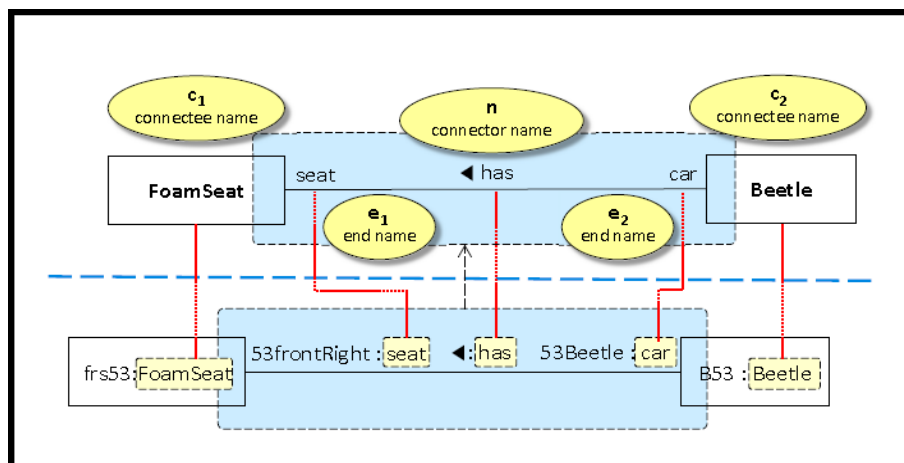


Fig. 9 Five component to identify type proposed in [7], this picture is taken from [7].

In [7], to identify a connection, 5 components are necessary, which are:

```
Type Specification = < < n : ct:name >
    < e1 : ct:end1 :name >
    < e2 : ct:end2 :name >
    < c1 : ct:end1 :connectee:name >
    < c2 : ct:end2 :connectee:name >>
```

In TS (Type Specification), n is the specified connection type name, and e1 and e2 means that specified connectionend type name respectively, and c1 and c2 means destination type name respectively. From the equation, it can be seen that in order to identify a connection type, 5 components are can be used. For example, the connection instance uses the specified type name “: FoamSeat”, “: seat”, “: has”, “: car”, and “: Beetle” to uniquely identify the connector type car–has–seat with the destination Beetle and FoamSeat in Fig.9.

TS is the foundation for deciding whether a connection matched a connection instance, partially matched or doesn’t match at all. However, [7] only suggested to consider a five-component match but left matching calculations design and heuristics as future works. In this project the idea proposed in [7] is used as foundation and stepping stone to further investigate the potentials of deep-connections in multi-level modeling. Considering that the possible matches need to be reordered and listed from highest possible matching ones to the lowest ones, the weighted sum calculation system is adopted in this thesis: each component is given a specific weight, if one component of a connection instance match against one potential type’s corresponding component, then add the component’s given weight to the weighted sum. After all potential connection types are compared and calculated, best matches and possible matches will be sorted according to the weighted sum. One virtue of using weighted sum is that it is straightforward: if a component matched, then its weight goes into the sum. In other words, if there is a component match, the match could be reflected as the number of the overall weighted sum. The larger the weighted sum value is, the more possible matching it is for the connection instance. Another virtue of adopting is that weight system is flexible in defining weight of each

component. It can allow to give more weight to particular components (such as the destination name) to produce more accurate results. In other words, that it is an approach to introduce priorities to the matching system.

In order to allow modelers to specify the type name for each component in the connections and entities, a new attribute needs to be added to the PLM of Melanee that can store user specified type name for each component. Hitherto Melanee does not support user defined type name for clabjects.

A matching score is a calculated weighted sum, but its purpose is to indicate a score for the quality of matching. It will be used for reordering purpose to obtain a list of matching types with the most likely matches in the top and less likely matches in the rear.

As weighted sum is adopted here, according to TS there are five components that need be considered during calculation of weighted sum, it can be seen that there are three categories of results from the instance's standpoint: 100% match, possible matches and incompatibles matches, these categories are listed and explained as follow:

1. **100% matches:** they are the category of types that matched with all the given type names in all the components in the connection instance. There could be multiple 100% matches at the same time.
2. **Possible matches:** these categories of matching types are the types that not all names matched with type names defined in the connection instances, but not all type names are mismatched at the same time, and they could be reconfigured to match the connection type.

3. **Incompatible matches:** they are the matches that both destinations failed to match with the type names defined in both the destination components in connection instance, or more components failed to match to result in a matching score of lower than 0.

From the type's stand point, there is another category of result, which is to **reconfigure type to match instance**. It is possible that the definition of some components in the type is incorrect, and it could be reconfigured according to the instance so that it can be a matching type. But only small differences between a type and an instance will produce this category of result. The reason for this is that it is unlikely that 3 or 4 components are wrongly specified by mistakes. So, small errors, like one destination type name wrong, one connection name wrong or both monikers wrong can produce this category of result.

Even though there are four categories of results, only **100% match**, **possible match** and **reconfigure type to match instance** are presented to modelers. Incompatible matches will be discarded.

For each component, there are three possible statuses for each value defined in the type name: right, missing or wrong. "Right" type name can match with the corresponding component in the potential connection types and a positive weight will be added to the overall weighted sum. "Wrong" means that the user specified type name failed to match with the corresponding component name in the connection type. "Missing" type name means the type name or the component name in connection types has not been defined yet, it could be assigned a right value or a wrong value, so it cannot be regarded as wrong, the weight it is assigned must be lower than that of being "right", for "missing" still have a chance of being assigned a wrong value. Hence the weight for "missing" status in each component must be between "right" and "wrong", to differentiate the three status in the matchings score.

As a starting point for weight calculation design, all the components are given weight 1 if they matched, and weight 0 if the component failed to match, which includes wrong and missing of component names in both connection instances and possible connection matching types. The scenario depicted in Fig. 10 shows two modeling levels with one connection instance of which its type has not been defined yet, and there are two possible connection types in one level above. From the Fig.10 it is clear that even though there are two possible types that could be the type for the connection instance “has”, only connection “has” in Level “O0” could be considered as the best matching type for connection instance “has”, for connection “has” in Level “O0” failed to match connection “contains” in “O0”, whereas connection “has” in “O1” matched with all type names defined in “has” in “O1”.

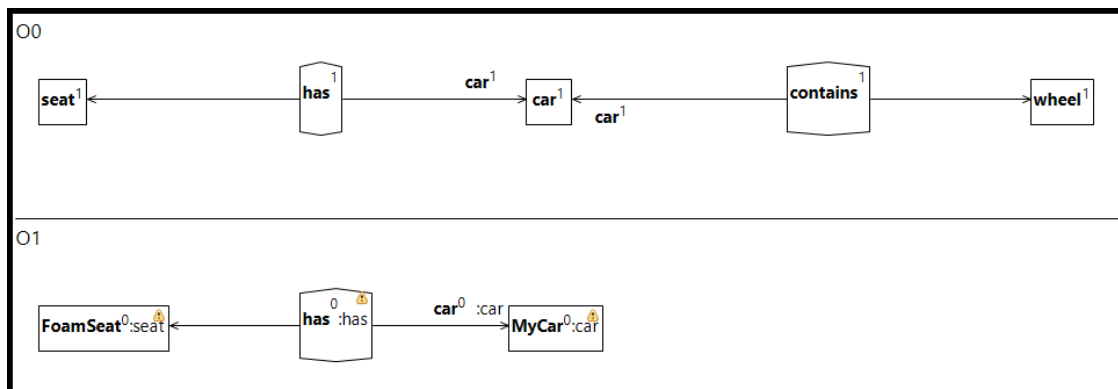


Fig. 10 Connection instance “has” matched connection type “has” by placing it in front of the list.

Another scenario is what is shown in Fig.11. There are two connection instances “has” and “contains” and two possible connection types. Considering the type names defined in the five components in the two connection instances, connections “has” and connection “contains” in “O0” should be the best matching types for connection “has” and the connection “contains” in “O1” respectively. By using design of giving weight 1 to all components, it makes the connection type “has” and connection “contains” the best matching types for connection instance “has” and “contains” respectively.



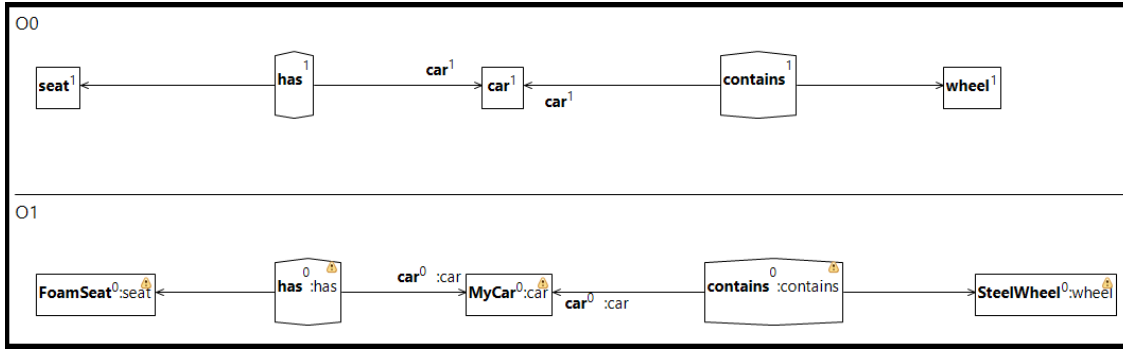


Fig. 11 Connection instance “has” and connection instance “contains” matched with “has” and “contains” in “O0” respectively.

Using weight calculation with weight 1 for a component match and 0 for a mismatch or wrong worked for these scenarios, and all best matches are listed in front of the list for each connection instances. But there is a problem for this, which could cause incorrect results listed as possible match for an instance. For instance, the scenario depicted in Fig. 12, there are only one connection instance which has not been defined with a type yet, and there are three possible connection types in Level “O0”, all named “has” but connected to different destinations. Using weight calculation design as discussed above and the types names defined in connection instance “has”, it is clear that connection type “has” between destinations “seat” and “cat” is the best matching type for connection instance “has” for all the types defined in connection instance “has” matched with their corresponding names in each component in connection type “has”. The connection type “has” between “car” and “wheel” will be listed as possible match for the type name defined in destination “FoamSeat:seat” in the connection instance failed to match with its destination “wheel”. Since there is one component failed to match, connection type “has” that connects to “car” and “wheel” will be listed after the best match connection “has”. For the last connection, which is connection “has” that connects to “truck” and “track”, it has two destinations failed to match with type names defined in both destinations in connection instance “has”, so it is listed after connection type “has” that connects to “car” and “wheel”.

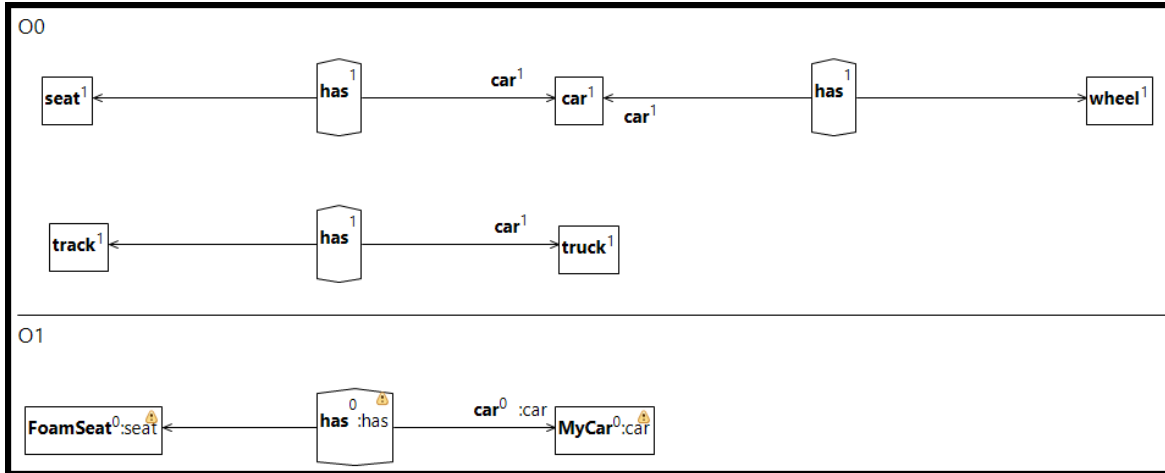


Fig. 12 Connection types with different destinations to match connection instance “has”.

However, there is an issue needs to be considered, which is that connections with different destinations are also listed as possible matches for a connection instance. In our case, connection “has” that connects to “wheel” and “car” and connection “has” that connects to “truck” and “track”, the destination names in these connections failed to match with type names defined in the connection instance “has”, but they are still listed as possible matches after best matching type “has” that connects “car” and “seat”. Connections represent relations between destinations, so if destinations failed to match, then the connection type in one level above should be considered as a mismatching connection type. But in the current weight calculation system, connections that should not be considered as matching types are still listed in the list for the connection instance.

To resolve this issue, and to differentiate the three status in the overall matching score, “penalties” is introduced in weighted sum calculation for connection conformance. It is called “penalty” is because that if a component failed to match with the type names defined in the connection instance, then a negative value is given and added to the weighted sum instead of 0. The advantage of using negative values for destinations mismatches is that the importance of the components could be reflected in the weighted sum, for example connections with two destinations failed to match, this should be

considered an incompatible match, and incompatible could be reflected in the weighted sum calculation by reducing it to a negative value. And in the final process of reordering all possible matches and 100% matches, it is convenient and clear to just remove all connection types that have weighted sum lower than 0.

Another issue that needs to be taken into consideration is that if there is only one destination failed to match, should it be considered as an incompatible match or just lower its weighted sum to allow it to be listed in the list. For the consideration of ease-of-use of the tool Melanee, it is possible that modelers could make mistakes in the system by defining type name of a destination to a wrong value, and for each connection type in the list, there are heuristic messages for each possible match to indicate how this possible matches could be redefined into 100% matches, so connections with only one destination failed to match are considered as low possible matches by reducing the weighted sum using penalty. But connections with both destinations failed to match will be considered as incompatible matches and will not be listed in the list at all.

About the “penalty” value that is defined to destinations, because if two destinations failed to match it should be given a value indicating that it is an incompatible match, the overall weight of an incompatible match should be smaller than 0. Hence both destinations mismatch will add negative values that are combined larger than the weighted sum of the rest of the three components without considering destinations. The combination of all components matched without destinations is 3, so the penalty for each of the destinations should be smaller than -1.5, and here -2 is adopted. By adopting this in the calculation, the example shown in Fig. 12 will only have two matches for connection instance “has’: one best match “has” that connects to “car” and “seat” and one possible match “has” that connects to “car” and “wheel”, the incompatible match “has” that connects to “truck” and “track” will not be listed. Thus, by adopting “penalties” for destinations, more accurate results are provided after calculation.

Fig. 13 depicts a more complicate scenario: there is one more connection added between “car” and “seat” called “contains”. In this example, connection “has” that connects to “truck” and “track” should not be considered as a match for both the destinations failed to match. Connection “contains” have the same destinations as connection “has” that connects to “car” and “seat”. These two connections have destinations that are compatible with type names defined in connection instance “has” so they should be considered as more possible matching types than connection “has” that connects to “car” and “wheel”. According to the weight calculation system design, the weighted sums for connection “has” between “car” and “seat”, connection “contains” and connection “has” that connects to “car” and “wheel” are 4, 3 and 1 respectively. And connection “has” between “truck” and “track” has weighted sum of -2, which is lower than 0, hence it is not listed in the list.

The above discussion involves the mismatch between destination name and destination’ type name. There is also another kind of mismatch, which is missing in defining type names for components. Missing in defining type names in components can still allow connections instances find matching types, according to [7], not all five components must be presented at the same time to identify a connection. But if the penalty value for connection destination type name missing be set to a negative value of -2 as the type name mismatched, there could be problems causing potential types not listed in the list. For example, the scenario presented in Fig. 14, between “car” and “seat”, there are two connections and one of the connection has no connection name. The connection instance has no type name defined in destination “FoamSeat”. Under such circumstance, the weighted sum for connection without name will be -1 if missing is also given the penalty of -2. Even though the connection without name has no name defined, it should still be considered as a possible match for they all connected to the destinations. The other destination type name is missing, which means it could be defined as the right one later. By considering so, connection without name should also be a possible matching type for connection instance “has”. Hence, missing type name defined in destinations should not be considered using the same penalties as wrong.

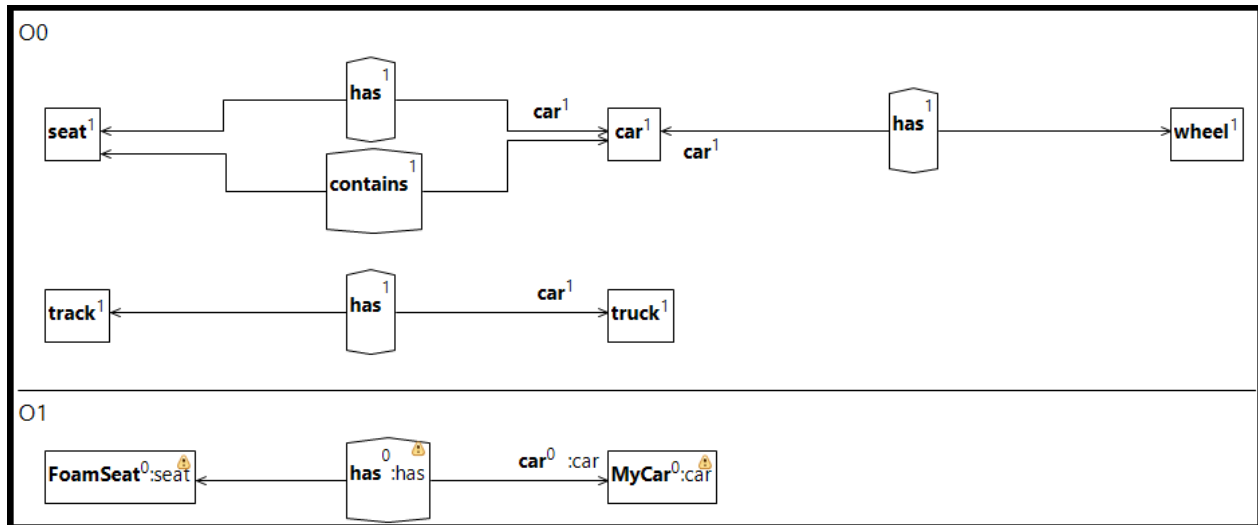


Fig. 13 More complicate scenario by adding another connection “contains” between “car” and “seat”.

As conclusion of the discussion above, a function can be formulated regarding to destinations in weighted sum calculation (x means the number of destinations mismatched):

$x = 1$  greatly reduce the overall matching score  
 $f(x) = \{x = 2 \text{ consider the matching type as an incompatible match}$   
 $x = 0$  allow the type to be further calculated

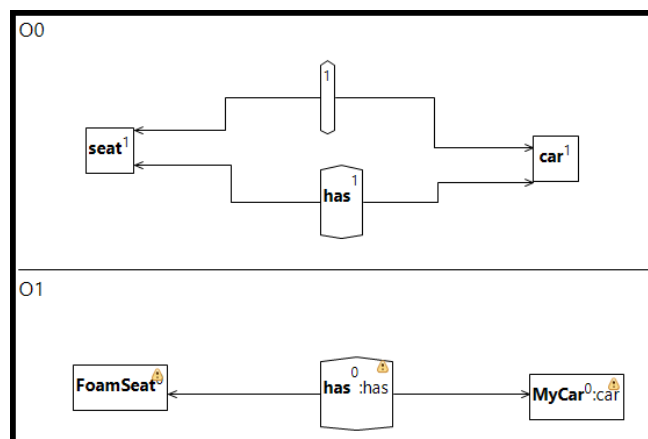


Fig. 14 Possible connection types has not connection name defined.

It is possible that modelers intentionally or unintentionally leave the type names blank in the connections. In such cases, in order to provide better result, for connection name component and moniker component, if the type name is not specified, then the connection name or the moniker in the connection instance will be used to compare against type's connection name or moniker. Destination component does not have this function is because that it is unlikely that entity type and entity instance have the same name, but for monikers and connection names this is more likely to happen. Hence in the absence of moniker or connection type name, the moniker or the connection name itself will be utilized for comparison in the weight calculation process. And if moniker or connection name themselves mismatched with a type, 0 will added to the weighted sum.

As discussed in [7], monikers are aliases for destinations, and with the absence of destination type name, monikers are sufficient to identify connections between destinations. Monikers could also be designed to use "penalties" so that "wrong" connection types will be placed in lower possible places in the list or even removed from the list if both monikers are wrongly specified when destination type names are missing. Because if both destinations are not matched, this kind of connection types will be considered as incompatible matches and will not be listed in the list. Hence for monikers, there are two different situations that need to be considered: when destination type name has been defined, and when destination type name is not defined.

When destination type name is defined, destination penalties can rule out incompatible matches as discussed. When all destination type names are matched, moniker and connection name are used to identify the possible matches to modelers. The scenario depicted in Fig. 15, in it there are three possible matching types for connection "has" in Level "O0". If penalty for moniker is the same value as the penalty for destination, connection with monikers name "spareSeat" and "spareCar" will be ruled out for it has a matching score of -2 in total, which is not correct, the possible connection types should be preserved and presented to modelers so that they can decide which is more like a type for the connection instance.

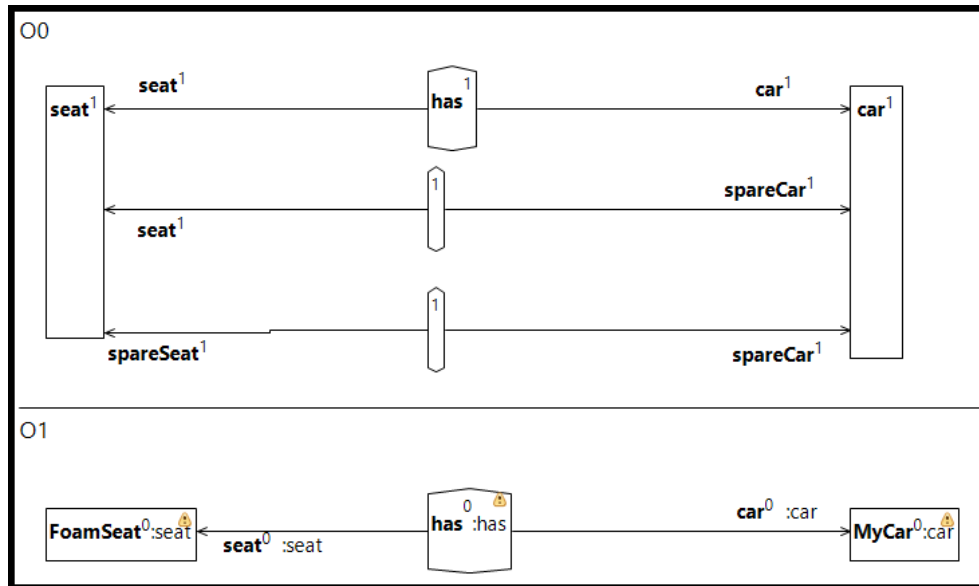


Fig. 15 Three possible matching types for connection "has".

Another scenario is what is depicted in Fig. 16. All component type names are defined in the connection type, and the moniker type names in the connection instance are wrong. In this case, if the penalty for moniker is set to -2 or even smaller value, the connection "has" will not be considered as a possible match at all. Even though moniker type names are all wrong, but the destination type names are all matched. Hence it should be presented to modelers as an option so that modelers can decide if connection "has" in "O0" is a suitable type or not.

As discussed above, penalty of -2 is too strict for moniker mismatch. But penalty -1 for moniker mismatch will not cause these problems. The connection with monikers name "spareSeat" and "spareCar" in Fig. 15 will be presented to modelers, so is the connection "has" in "O1" in Fig. 16.

The above discussion for moniker is under the circumstance destination type name is defined. However, when destination type names are not defined, monikers could work the

same way as destinations to rule out impossible matches. Hence in the absence of destination type name, type name of the moniker that connects to this destination will use penalty with the same value as destination penalties to provide better matching results.

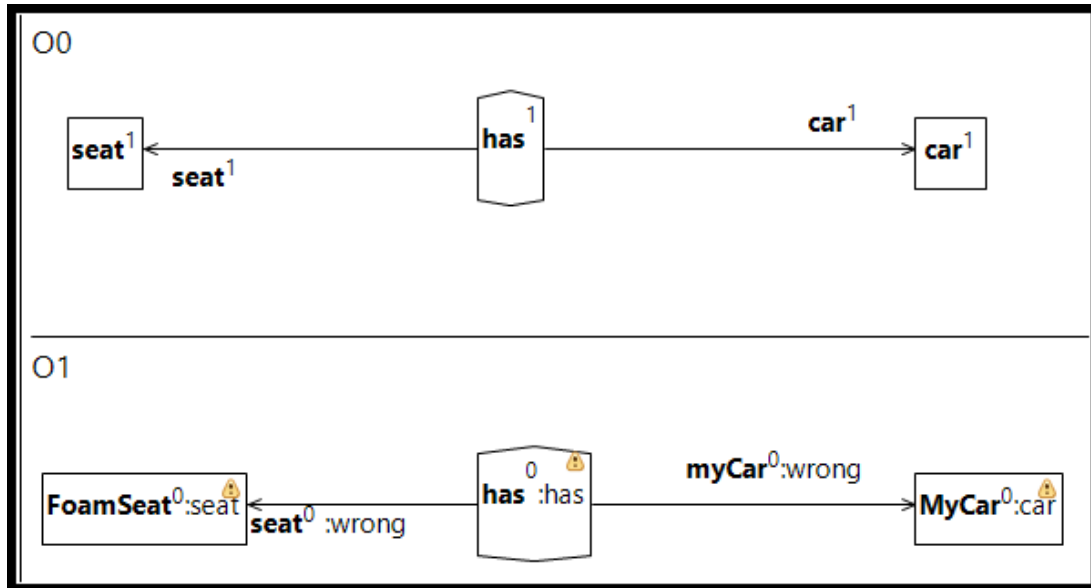


Fig. 16 Three component type name defined for connection "has" in Level "O1".

And for connection name, it can be used to identify a connection. But it is possible that modelers will rely more on monikers instead of connection name to represent the relationship between two destinations. In such cases, weight 1 for connection name could rule out some possible matches, which is like what is shown in Fig. 17. There are two connections that could be the type for connection "has" in Level "O1". If weight 1 is used for connection name matched, then connection "has" will be considered as an incompatible match for its overall weighted sum is -1. However, if modeler rely on monikers, this weight design can provide the possible matches with the right connection type, which is connection "contains"; but if modelers rely more on connection name, this design will rule out the one connection they want, which is connection "has" in "O0". Hence in situation like this, both connections in "O0" need to be presented so that modelers can decide themselves. To resolve this, weight 2 is used for connection name



matched. Thus, connection type “has” will also be listed as a possible matching option to modelers.

“Penalties” is used in for monikers and destinations. For connection name component, penalties should be adopted as well to differentiate the three status of right, wrong and missing. In the scenario depicted in Fig. 18, there are three possible connection types, all connected to the same destinations. All the type names in each component are identical except connection names. If no penalty is not adopted, connection “contains” and the connection without name will have the same matching score, so they will be treated equally. However, connection name missing means connection name has not been defined yet, if a connection name has not been defined, it should still be considered as a more matching connection than then connection with its name being wrong. In our example, considering all other components are identical, the connection type without name should have higher matching score than connection “contains”. Hence penalty is adopted for connection name component for wrong values. To lower the possibility of the connection as the type, penalty -1 is added when connection name mismatched. Thus the connection types in Fig. 18 will be listed as “has”, “” and “contains” (from more matching ones to lower matching ones).

Since the weight of connection name component matching is raised to 2, the penalty of destination mismatch should be adapted according to the current weight design. The penalty for destination is to make sure that if both destinations failed to match, then even if all other components matched, its overall matching score should be smaller than 0 to indicate it is an incompatible match. The weighted sum of all other components except destinations is 4, so the value of destination penalty should be smaller than -2. In this design, -3 is adopted so that if both destinations failed to match, this connection will not even be considered as a match at all.

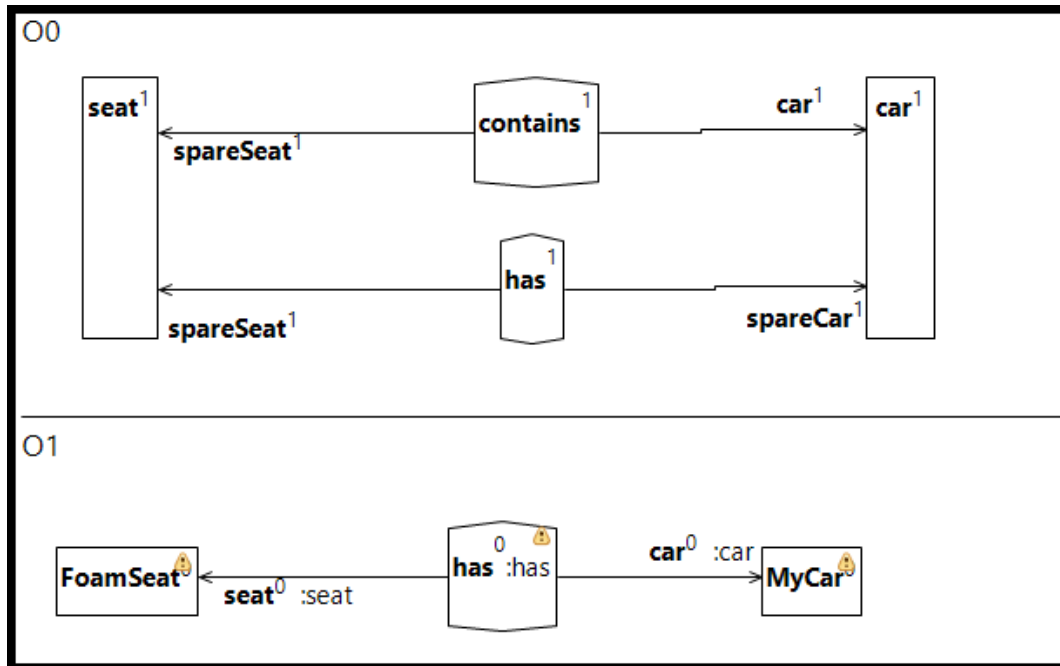


Fig. 17 Two possible matching types for connection "has" in "O1".

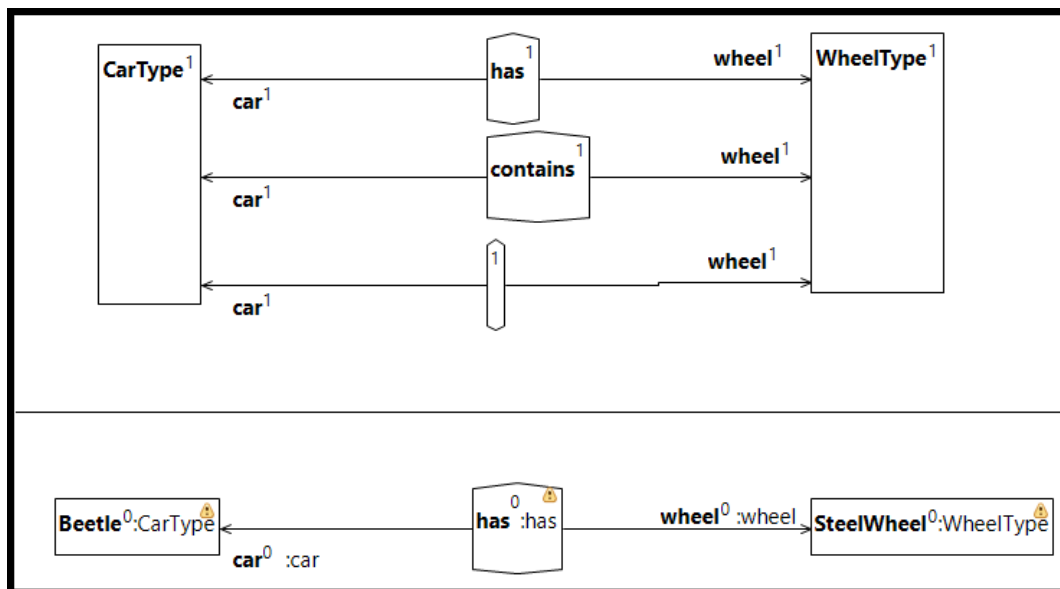


Fig. 18 "Penalty" used in connection name component.

Also for the moniker penalty, which is in the absence of destination type name, the penalty for moniker mismatch are increased to a higher value to better filter out possible matching types accordingly. So, when destination has not been specified a type name, moniker can be used to filter out incompatible matches by increasing its penalty to a lower value like - 3, which is the penalty for destination mismatch.

There is one more circumstance that needs consideration which is missing type name in connection instance or missing component name in possible connection matching types. The missing component name in possible connection matching types means that the name of the component has not been defined yet, it is possible that it will be defined as the right matching name later, like what is shown in Fig. 19. It is possible that it will be defined as “has” for the connection in “O0”. However, if the connection instance has not been defined a matching type name in a component, like what is shown in Fig. 20, the name for the connection type is defined as “has”, and for the connection instance the connection type name is missing. Though it is possible that the type name could be defined later to be the right name, it has not been defined yet, and according to the possible matching type “has” in “O0”, the type name should be presented for the connection name “has” is not missing in the possible connection matching type “has”. Hence a penalty will be given under the circumstance when the type name is defined in the possible matching types, but missing in the connection instance components. This is because the component name is defined in the connection instance, and it should be defined as type name in the corresponding component. If the component type name is missing in the connection instance and defined in the possible connection type, then a slight penalty should be given. Here in this design, a slight penalty of -0.5 will be given for missing type name in components in connection instances in such circumstance.

In pervious discussion, it is argued that if type name is missing in components except destinations, the component name itself will be used to compare against the possible matching types, but considering missing of type names in connection instances has “penalties” now, a small adjustment needs to be done: if component type name is missing

in component that are not destinations, then the component name will be used to compare first, if it matched, the according weight will be added to the sum; if not matched, then a penalty of -0.5 will be added to the weighted sum as penalty for missing component type name; if in destinations, penalty -0.5 will be added if the destination type name has not been specified, no destination name will be used compare against its type's name.

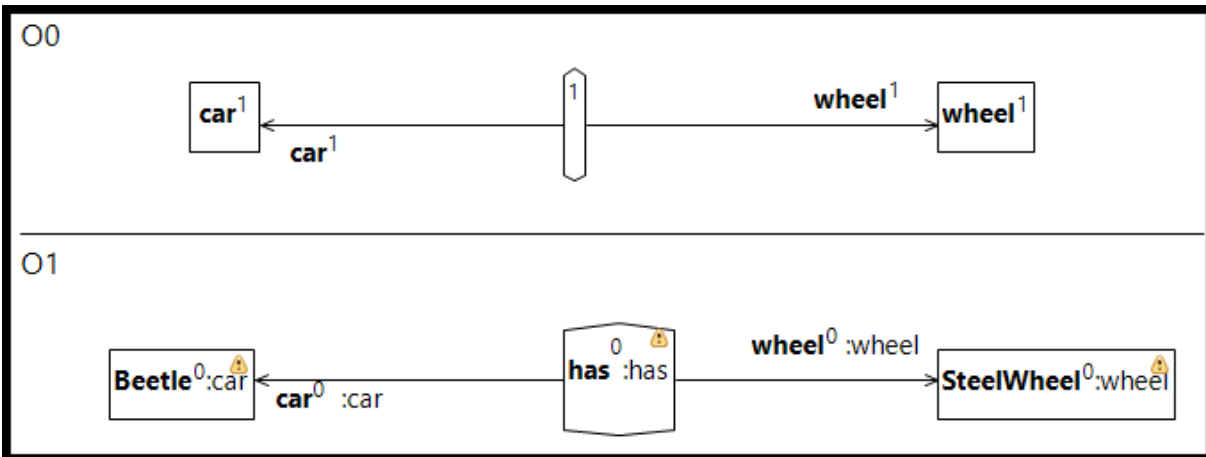


Fig. 19 Missing component name in possible connection matching type.

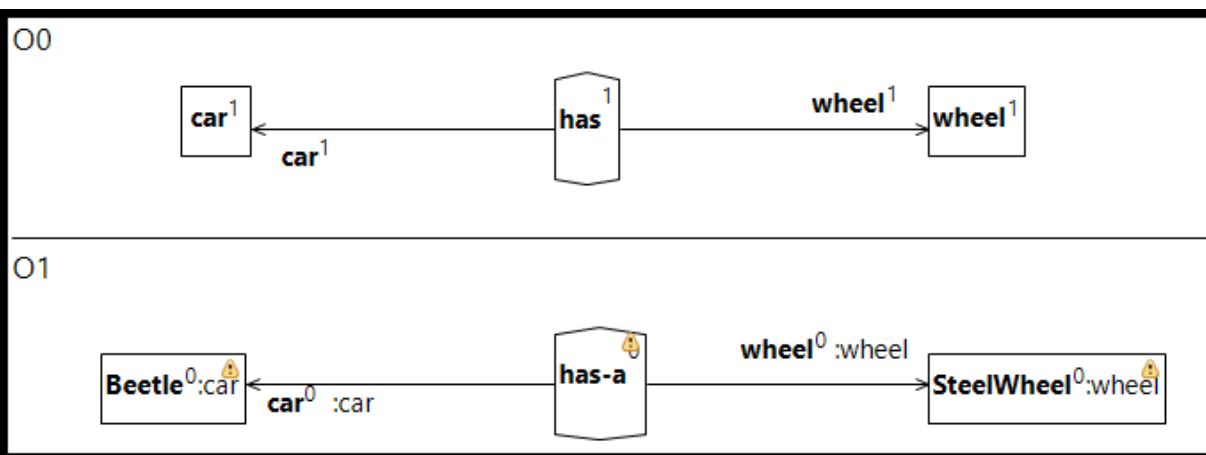


Fig. 20 Missing component name in connection instance.

To this point, the weight design of each component is listed in Table 1.

Component	Weight of Right	Penalty of Missing type name	Penalty of wrong
Connection Name (n)	2	If type's connection name is defined: -0.5; Else :0	-1
Moniker (e)	1	If type's moniker name is defined: -0.5; Else :0	If Destination type name defined: -1 If Destination type name not defined: -3
Destination (c)	1	If type's destination name is defined: -0.5; Else :0	-3

Table 1 Weight Design of each component.

Hence the matching score calculation function is:

Matching score =  $n + e_1 + e_2 + c_1 + c_2$  (n is connection name, e means "connectionend" and c means destination). The weight for n, e and c are decided as shown in Table 1.

There are some other scenarios that need to be taken into consideration, which is the sub/super type relations between destinations. For example, the two scenarios depicted in Fig 21 and Fig 22. The type names defined in connection instance "has" matched with those of connection "has" that connects to "SeatType" and "CarType" in Fig 21 except moniker type name "seat:rear", so connection "has" that connects to "SeatType" and "CarType" is considered as the most possible matching type for connection instance "has". Whereas connection "has" that connects to "RearSeatType" will be considered as possible match with low matching scores. Considering Entity "RearSeatType" is the sub

type of Entity “SeatType”, connection “has” that connects to “RearSeatType” should be given a higher matching score for “FoamSeat:SeatType” defines “SeatType” as its type, so “RearSeatType” could also be the type for “FoamSeat”, hence connection “has” that connects to “RearSeatType” can also be viewed as a matching type with high matching score for connection instance “has”. Hence if a destination is the sub type of the destination defined in the connection instance, it will be given a weight that can represent this relationship. And in order to differentiate the sub type and the type defined in the destination of the connection instance, value 0.5 is given to sub type destinations. The same also applies for super type relationship, if a destination is the super type of the destination defined in the connection instance, it will be given a weight of 0.5, the situation is like what is shown in Fig. 22.

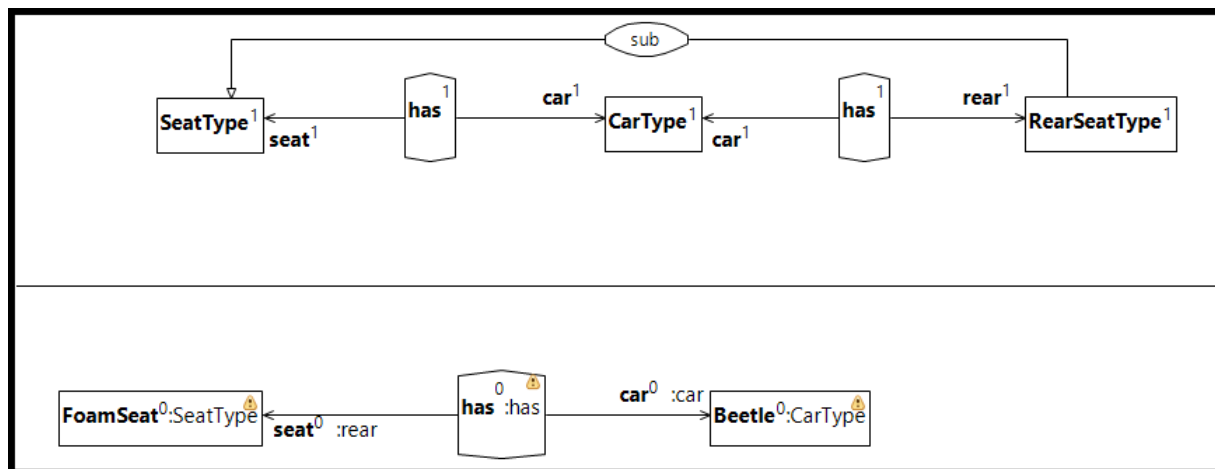


Fig. 21 Sub type relationship between “RearSeatType” and “SeatType”.

A more complicate scenario is shown in Fig. 23. In this scenario, there are five possible connection types, three of them connect to the same destinations of “seat” and “car”. According to the type names defined in the connection instance, the three connections connect to “seat” are listed as more matching ones in front of the list. Connection “has” between “car” and “wheel” is considered as a low possible matching type for penalties received because of one destination and one moniker failed to match.

There are other scenarios examined using the design as shown in Table 1. and the more thorough evaluations are conducted and discussed in Chapter 5.

After defining the weight for each component, the equation of calculating weighted sum, or called the matching score, is as follow:

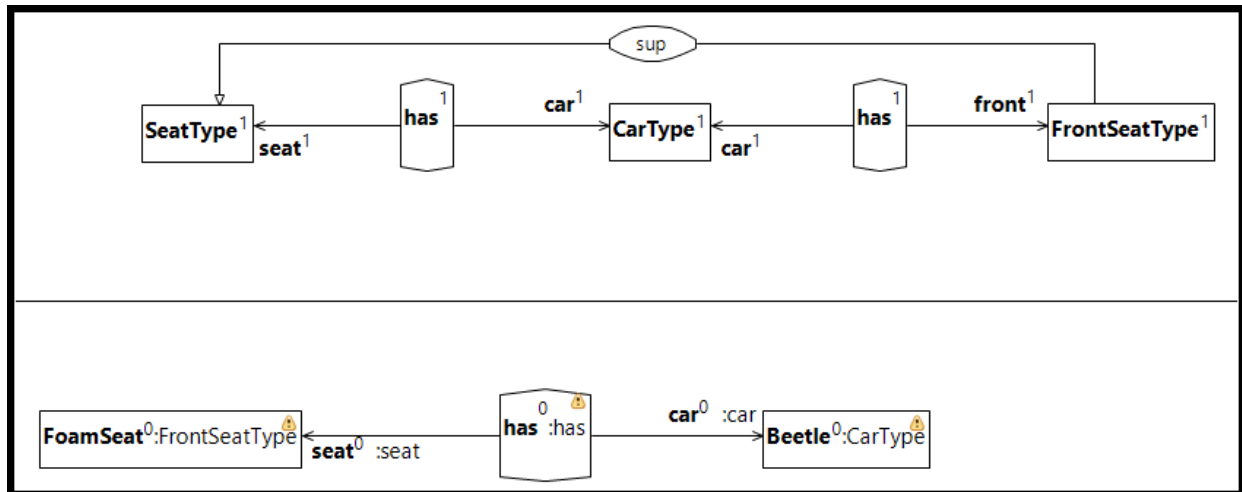


Fig. 22 Super type relationship between “FrontSeatType” and “SeatType”.

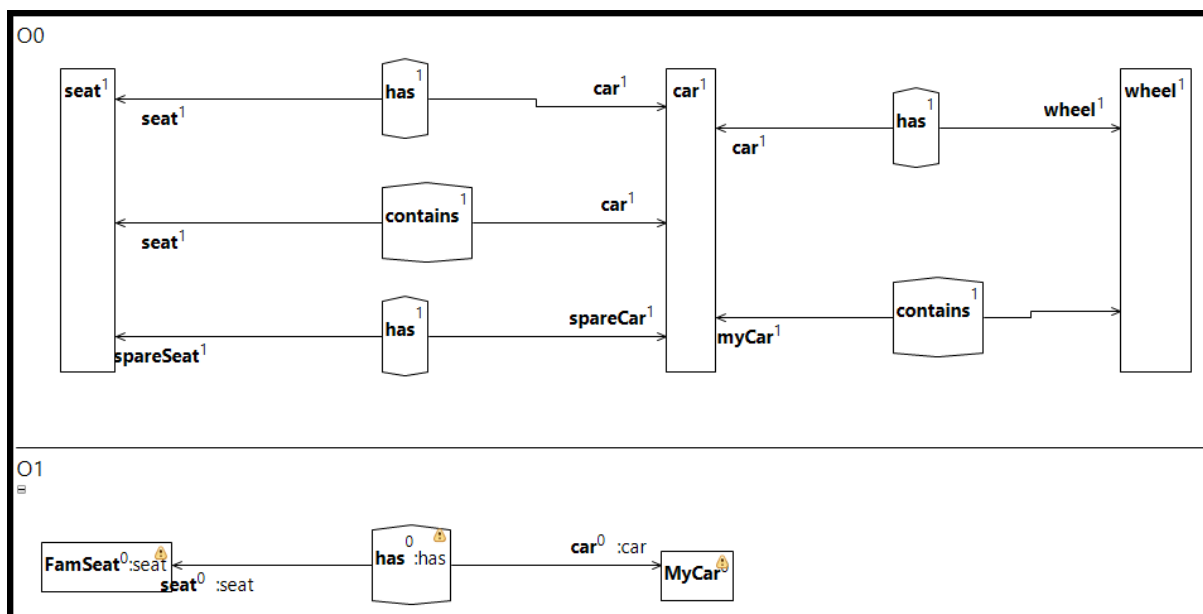


Fig. 23 A complicate scenario for examining weight design.

$$\begin{aligned}
\text{Weighted sum} &= (\text{ConnectionName match}) + (\text{BestConnectionEnds match}) \\
&= (\text{Connection Name match}) + \\
&\quad (\text{BestConnectionEnd}_1 \text{ match} + \text{BestConnectionEnd}_2 \text{ match}) + \\
&\quad (\text{BestDestination}_1 \text{ match} + \text{BestDestination}_2 \text{ match})
\end{aligned}$$

The “Best ConnectionEnds match” function is aiming to provide the best match between instance and potential type for both connection type and connection instance have two connectionends. Here I use “swapping” to find the best match, which is: compare (LL+RR) against (LR+RL) to see which set has higher weighted sum. LL means instance left connectionend compare against the type’s left connectionend, LR means instance left connectionend compare against type right connectionend and so on. Thus by doing so, the function can get the best match and will not omit possible matching situations.

The “five components” aforementioned above is very efficient as comparison variables for identifying types, but one more element needs to be taken into consideration as part of the comparison process, which is the potency of each component. In order to be the instance of a specific type, potency values of each component in the instances needs to be smaller than those of their types. Hence the instance’s component potency values should be smaller than those of the potential types’ and if the potency of a component is 0 then it should have no instance [1]. The potency uses “-1” to represent “infinite” as “\*” in LML and in PLM, and in such case the corresponding potency value of the type’s instance could be any value such as infinite, zero or any other positive numbers.

Hence, there are two attribute in each component in the that needs to match: type name and potency. If the potency fails to match or the user specified type name fails to match, the corresponding component will be considered as fail to match. In other words, the potency and user specified type name need to match at the same time, otherwise the



component will be considered fail to match, and penalty value will be added to the weighted sum.

Another part of the design is about heuristic messages: how to present information regarding modification tips to modelers and how to present the list of potential connection types so that the connections in the list can be utilized properly by modelers to solve the type definition issue. First let us discuss about the tips, or messages of the three types of results that will be presented to modelers.

The messages presented to modelers should not be as simple as “matched connection ‘XXX’ ” or “partially matched connection ‘XXX’ ”. The message should be comprehensive, correct and concise, so that modelers can see clearly which connection types are the best matches, and how to adjust connection instances to match a connection type in the model.

There are three categories of results hence there are three categories of messages to modelers. If it is a best match, a simple message will be presented like:

*Connection 'aaa' matched connection 'AAA'*

But if the current connection instance is a partial match, or what is called a possible match, then the message about this type will be more complex, for it must convey messages that related to how to change some of the type name defined in components in current connection instance to match certain connection types. This type of message will be like:

*Connection 'aaa', to match connection 'AAA', change type name to 'AAA' in connection 'aaa', change type name to 'BBB' in ...*

The length of the message is dynamic and depends how many unmatched type name or potency value can be found during matching score calculation.

The third type of result is about how to reconfigure types to match current instance, so the third kind of message will resemble the second type of message, but somewhat different. The third type of message will be like:

*To match connection instance 'aaa' for connection 'AAA', change potency to '-1' in connection 'AAA'...*

Again, the length of the message of third type of result can vary and depends how many components needs reconfiguration.

The messages presented to modelers will be reordered according to the matching scores so that best matching types will be presented on top of the list, and low possible ones will be in the rear of the list. Thus, modelers can examine the types with priorities. A complete of messages example will like:

*Connection 'aaa' matched connection 'AAA'*

*Connection 'aaa', to match connection 'AAA', change type name to 'AAA' in connection 'aaa', change type name to 'BBB' in ...*

*To match connection instance 'aaa' for connection 'AAA', change potency to '-1' in connection 'AAA'...*

The purpose of Connection Conformance is not merely aiming at tipping users of which connection hasn't been defined a type or which connection can be its type, but to provide LML and Melanee the capability of achieving "explanatory" modeling and convenience so that Melanee users can find the modeling process convenient and user-friendly. In order to achieve this, the do-fix section provided by Epsilon Validation Language is adopted here to allow users to fix the connection type definition issue easily and accurately.

Description	Resource	Path	Location	Type
Warnings (4 items)				
Resolve type for Entity 'ee1'	default.lml	/test01/src	<Domain>::test...	Melanee - Mult...
Resolve type for Entity 'ee2'	default.lml	/test01/src	<Domain>::test...	Melanee - Mult...
Resolve type for Entity 'withouttype'	default.lml	/test01/src	<Domain>::test...	Melanee - Mult...
To match connection 'cc', change userSpecifiedType	default.lml	/test01/src	<Domain>::test...	Melanee - Mult...

Fig. 24 Critique Warning for quick-fix.

Do-fix section is defined inside constraint or critique to allow users to provide solutions to these warnings or errors. There could be more than one do-fix section inside one constraint or critique and each fix section can offer one solution to the error (Fig 24 and Fig 25). As aforementioned above, there are three kinds of result that could be presented to user, and each result may contain more than one connection type. Do-fix needs fixed number of fixes and cannot handle dynamic number of situations with uncertain numbers of choices. Hence, to do this, there are three possible options to provide fixes to users: use multiple critiques for different result types, show three fixes in one critique, or call native java method in EVL script.

First the use of multiple critiques. At first glance it seems it fits our requirements: there could be three critiques and each one is for shown one category of the result. But it is clear that if there are no best matching types, the size of result array is 0, then there is no need to show such warnings and just let the critique return true. Three separate warnings, and each one is independent of others, so each critique can decide within itself if it needs to be shown or not, like shown in Fig. 26. However, multiple warnings can cause chaos if there are too many connections or entries of which their type haven't been defined yet, for there could be triple number of warnings than expected, and as mentioned above, the critiques are independent of each other, there is no way to let the three share the computing results, so in each critique, the weighted sum calculation and reordering processes need to be performed three times, which is not efficient at all. Furthermore, each do-fix section provides solution to one problem, in our case it can only provide

solution of setting to one of the types for each kind of result, dynamic number of results cannot be handled properly here.

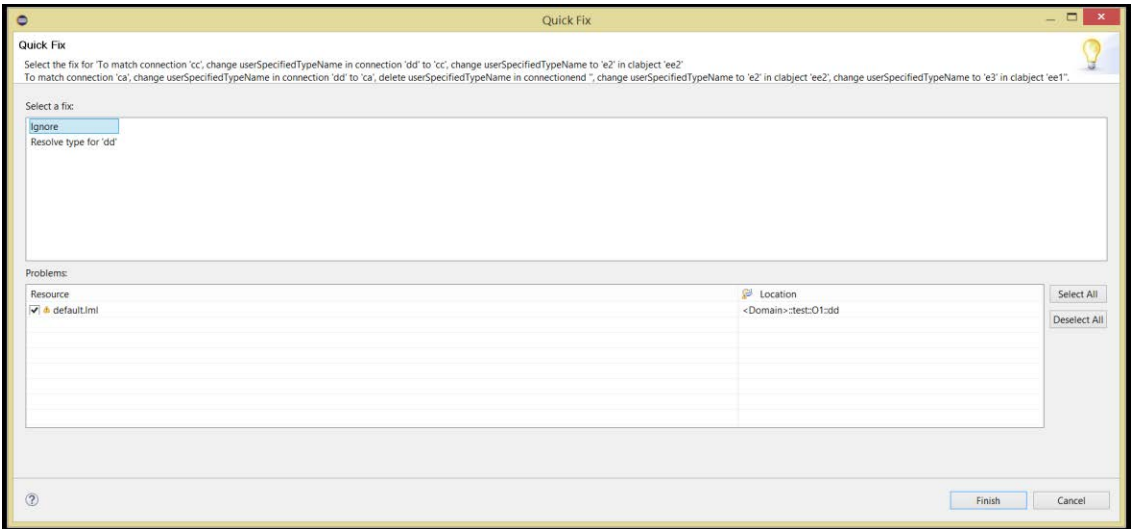


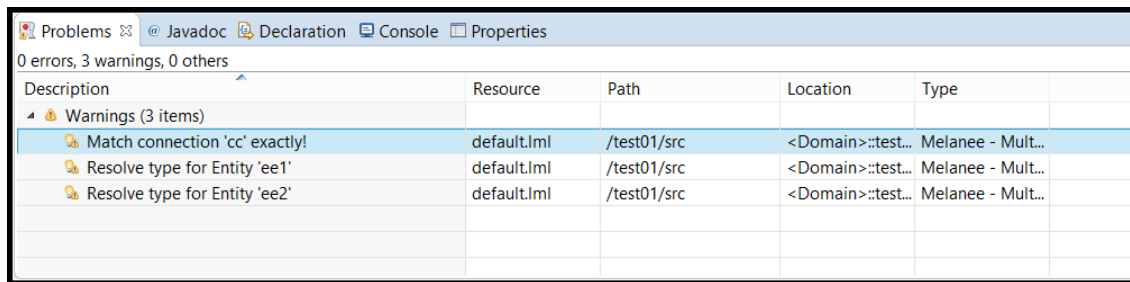
Fig. 25 Selecting one fix alternative in the quick-fix dialogue.

Problems 0 errors, 6 warnings, 0 others					
Description	Resource	Path	Location	Type	
Best matching type of 'ee1' is found	default.lml	/test01/src	<Domain>:test...	Melanee - Mult...	
Best matching type of 'ee2' is found	default.lml	/test01/src	<Domain>:test...	Melanee - Mult...	
Match connection 'cc' exactly!	default.lml	/test01/src	<Domain>:test...	Melanee - Mult...	
Select type for 'ee1'	default.lml	/test01/src	<Domain>:test...	Melanee - Mult...	
Select type for 'ee2'	default.lml	/test01/src	<Domain>:test...	Melanee - Mult...	
To match connection 'ca', change userSpecifiedType	default.lml	/test01/src	<Domain>:test...	Melanee - Mult...	

Fig. 26 Multiple critiques.

Another option is showing three fixes in one critique, this one is better compared to the first option: only one critique for each connection, hence there will be no chaos in showing warning signs if there are many connections without type and the number of warnings will be significantly reduced, like what is shown in Fig. 27; and within one critique, the calculation of weighted sum between instance and all potential types and reordering of the result will only need to be performed once. So, this option is more efficient compared to the first option. But it has its own downside: as the number of fix sections are fixed, all

the three fix in the quick-fix dialog must be shown, regardless of whether the type of connections have been found or not, which is shown in Fig. 28. Even if there are no best matches and no valid types can be reconfigured to match the instance, the two types of fix to users will still have to be shown, and if user click on it, nothing will be performed for the corresponding category of types is empty. And again, no dynamic number of result can be handled here, just like the first option.



Description	Resource	Path	Location	Type
Warnings (3 items)				
Match connection 'cc' exactly!	default.lml	/test01/src	<Domain>::test...	Melanee - Mult...
Resolve type for Entity 'ee1'	default.lml	/test01/src	<Domain>::test...	Melanee - Mult...
Resolve type for Entity 'ee2'	default.lml	/test01/src	<Domain>::test...	Melanee - Mult...

Fig. 27 Few warning signs by calling java method.

Hence I propose using the third option here: calling java method inside do-fix section to prompt a second-stage dialog. By implementing in this way only one critique is needed, so the efficiency problem is resolved compared to the first option. And in this way only one do-fix section is required even if the number of do-fix sections is fixed. And by calling java method, dynamic number of potential types can be handled in the second-stage dialog (Fig. 29) in Java code. And by developing so, few clicks are needed and few warning messages are presented to users, which will provide a clean and user-friendly modeling environment.

The quick-fix will fix the connection instance's type issue more conveniently. But it should do more than just setting the type. For just fixing the type issue there could be inconsistencies between the type and the instance, such as the potency of instance is larger than that in the type, or the moniker type name is wrongly specified. The connectionend's type will be set, and since the types of connectionend and connection

are known, the graph will present types' name after the colon to present types' names instead of using values from user specified type name. The potency will also be automatically adjusted, if the potency in connection instance component is larger than that of its type, it will be reset to be type's potency value decreased by one, users may want to specify other values such as decreased by two or three for other reasons, but that is left for modeler's intervention, the automatic fix will set the instance's potency to decreased by one if they are larger than the type's potency.

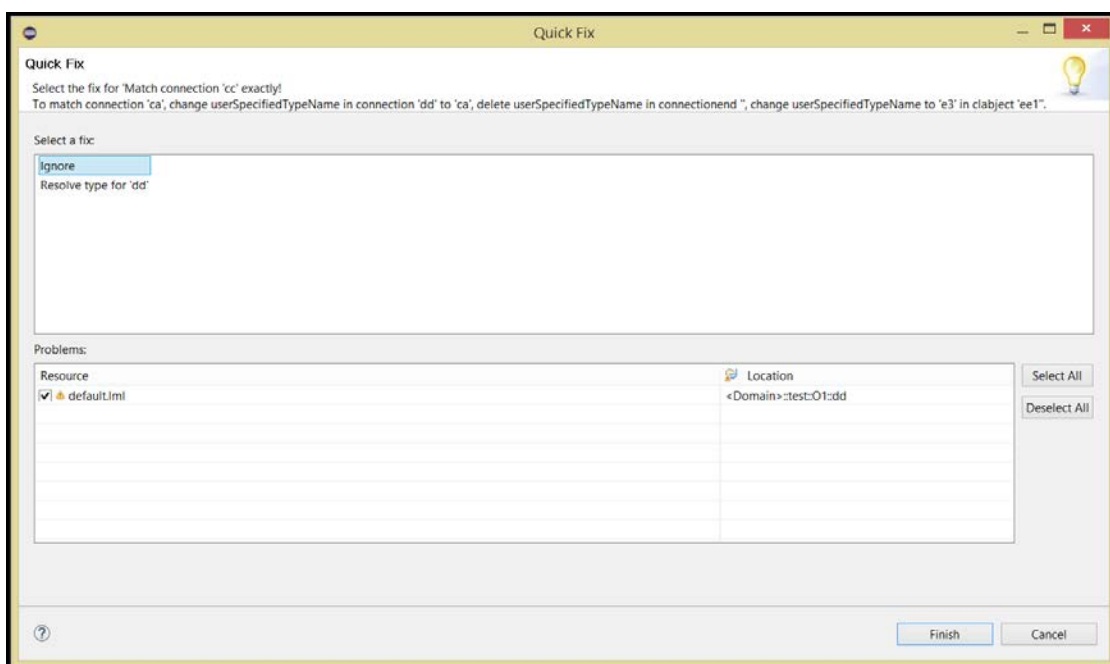


Fig. 28 Clean and clear fix section.

The destinations' types are not automatically fixed in the Connection Conformance's quick-fix, but there is a validation if the connection instance's type is set to wrong destination types. The destinations are important as part of an identifier to identify a connection as discussed, hence errors instead of warning is used here. Also, there are quick-fix for destinations' type setting to support Connection Conformance function, but the destinations' types will be set to the appropriate ones only if the destination's type hasn't been set yet. This is under the consideration that each entity could have more than

one connection, and when modelers selecting type for a connection instance a mistake could be made to accidentally choose the wrong type, if this happened all the other connections that associated with this entity will be invalid. Hence, changing entity's type is performed in another validation's fix section to avoid mistakes.

To ensure the model is correctly constructed, several validations will be added to Melanee which are:

1. If the connection's type is determined, then the connection ends' types must be set accordingly.
2. If the connection's type is determined, then the destinations' types must be set accordingly.

### 3.2.2 Entity Conformance Design

It is necessary that entities' type checking, warning and quick fixes should also be provided. As part of Entity Conformance functions, entities are also enhanced with checking for type function and type selection function to ensure the correctness and integrity of the whole model in support of "explanatory" modeling in Melanee.

The mechanism of entity type checking is similar to that of connection. Entities in all levels except the topmost one will be checked to see if the type has been defined, and if the type is not defined, then there will be a warning sign to indicate such situation (shown in Fig. 30). The reason it is a warning instead of an error is the same as that of connection conformance: it is not a critical error but a reminder to modelers. The flow to set the entity's type works almost the same as that of the connection type setting: right click on the corresponding warning sign and select resolve type in the prompted dialog, the quick-fix for entity conformance also call the Java native method to prompt a second-stage dialog to notify the user if there is best match, show the best match, otherwise just show the potential types to let users themselves to choose the correct type. There are two types

of potential result in entity type setting: best match and potential match, the reason there doesn't need a third type of result is that in entity type matching, only entity type name and its potency need to be compared, and the type name is crucial in deciding the best match, so reconfigure entity type attribute value is not an option here. All the entities in immediate level above will be acquired and compared. Fig. 30 shows the warning of entity type, and the warning is also displayed in the Problem Properties View in which modelers can choose quick-fixes to fix the problem (shown in Fig. 31). Fig. 32 (a) and Fig. 32 (b) show the dialog after right click on the warning and the second-stage dialog for entity type selection.

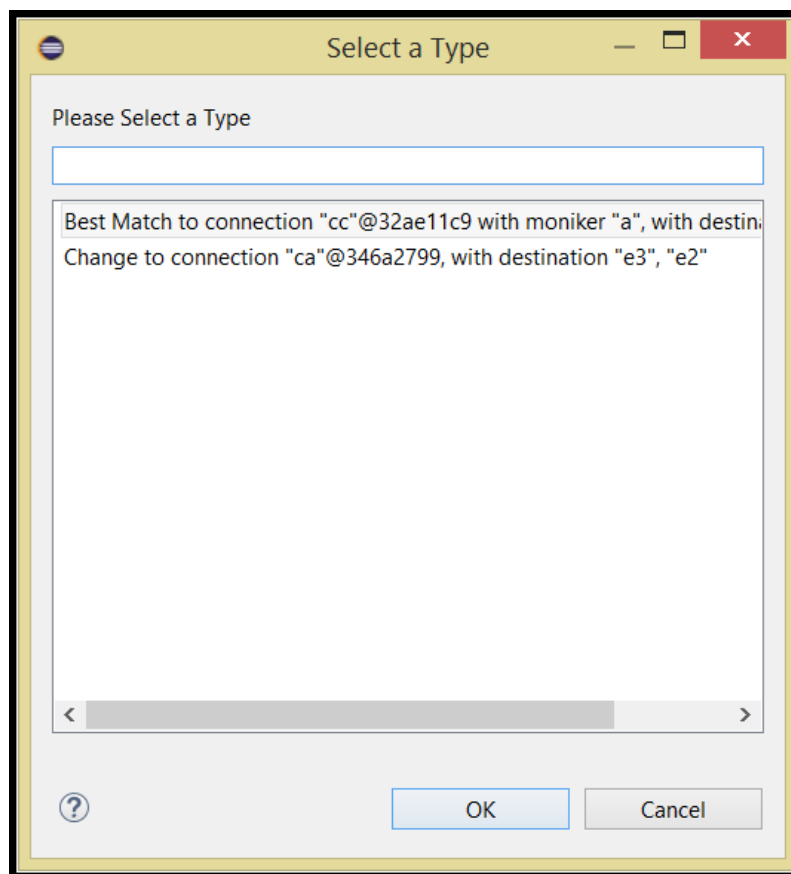


Fig. 29 Second-stage dialog for resolving type issue.



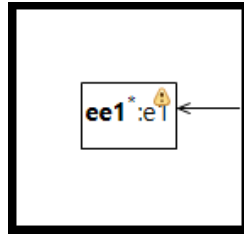
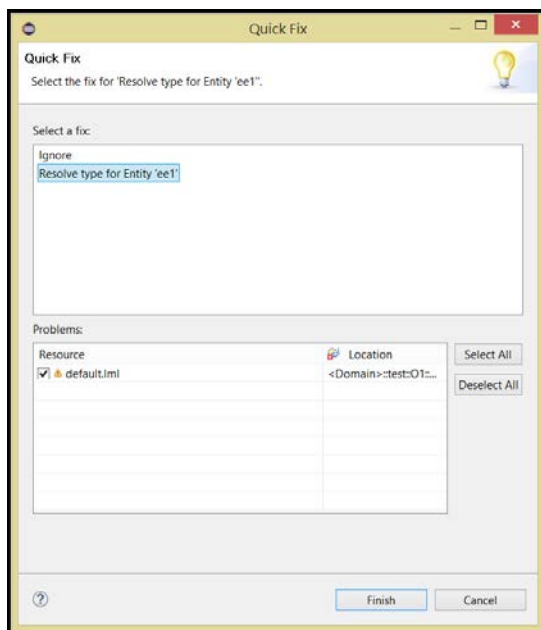


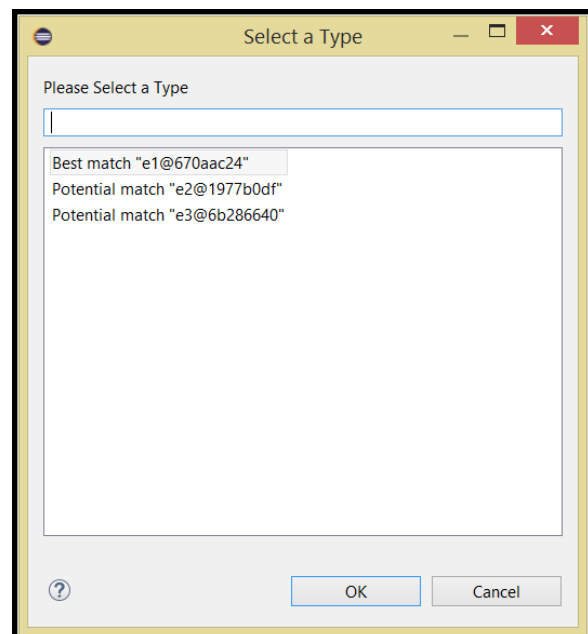
Fig. 30 Warning in entity to indicate the type is not set.

Problems 1 error, 2 warnings, 0 others					
Description	Resource	Path	Location	Type	
Errors (1 item)					
Warnings (2 items)					
Resolve type for Entity 'ee1'	default.lml	/test01/src	<Domain>::test...	Melanee - Mult...	
Resolve type for Entity 'ee2'	default.lml	/test01/src	<Domain>::test...	Melanee - Mult...	

Fig. 31 Entity quick-fix warning.



(a) Entity type resolve window.



(b) Second-stage dialog for selecting entity type.

Fig.32 Entity type issue solution dialogs.



## 4 Implementation

According to the design discussed in last chapter, there are three main features needs to be implemented: deep-connections, connection conformance and entity conformance. To understand what detailed extensions to Melanee are required, we first need to understand what Melanee already supports: connections in Melanee are view as “clabjects”, and each connection has two connectionends. Each connectionends has a moniker attribute but no potency in it. In addition to this, the multiplicity can only describe how many connections instances there should be in immediate one level below, deep multiplicity and proper validation are not in place. And so far, user specified type name in entities and connections are not supported. Hence, before implementing conformance functions, deep connection feature should be implemented in Melanee. Hence this chapter contains three parts:

1. Deep-connection feature implementation
2. Connection Conformance implementation.
3. Entity Conformance implementation

Before entering development details, it is necessary to briefly introduce Melanee structure and develop flow for better understanding of big picture of Melanee, which can help to facilitate the development process. Fig. 33 depicts the frameworks and technologies that Melanee built on. Melanee was developed based on GMF, and according to the discussion in Background Chapter, GMF is based on another two frameworks: EMF and GEF. EMF is responsible for meta-model related development and model validations, and GEF can generate model controlling code for graphical editing. The GMF framework can generate model development related code such as model code and diagram control code. Whereas Melanee supports more features than GMF can automatically generate, since Melanee was built on Eclipse platform, Eclipse Plugin development can allow Melanee to have more features such as customized properties sheet view, customized view of model or customized dialogs etc.

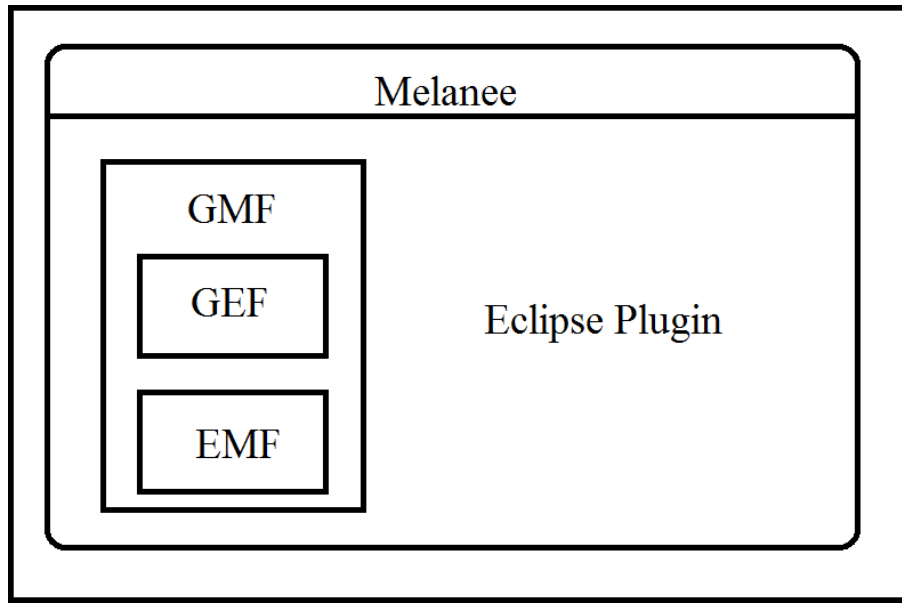


Fig. 33 Melanee structure.

EMF and GMF can generate code which simplify Melanee development process for not all the model controlling related code needs to be written by hand. The code generation of the frameworks can allow developers to customize generated code to provide customized features by writing customized code into corresponding templates. The development flow of Melanee is shown in Fig. 34. PLM.genmodel can generate model code and model control code according to the meta model defined in PLM.ecore file, and the customized code can be written into the generated code by modifying templates stored in templates folder under plugin org.melanee.core.models.plm. PLM.gmfgen generates diagram control code, and customized code can be written into templates stored under templates folder under plugin org.melanee.core.models.gmf. The generated code and customized code together allow Melanee to have more features to support constructing models in multi-level modeling environment.

In the following of this chapter some code will be shown because Melanee was built on complex plugins that uses several frameworks. Therefore, what would normally be a relatively straightforward coding task is much more involved. Showing the

actual code illuminates how the existing plugin and framework need to be used to accomplish the respective changes. Revealing code in this thesis is for the benefit of future coders.

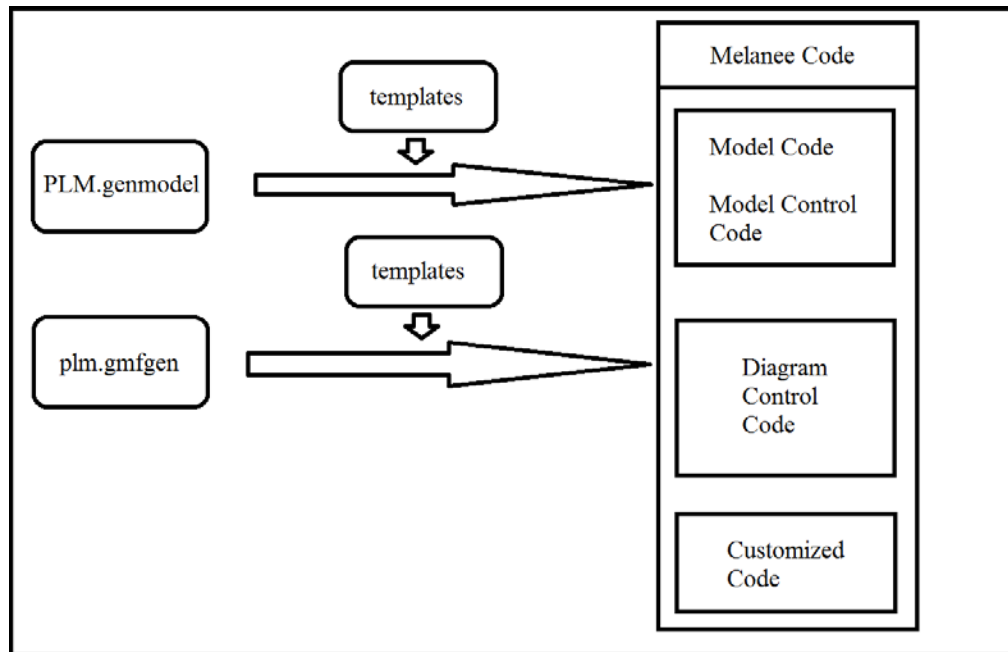


Fig. 34 Melanee development flow.

#### 4.1 Deep Connection Implementation

Adding deep-connection feature support for Melanee can be divided into three smaller tasks, which are listed as follow:

The first task of adding potency to moniker, it requires modification to the meta-model. After adding potency to connectionends in meta-model, connectionends can be constructed to hold potency value so that the potency can indicate how many levels down the moniker can describe its instances. The potency value should be decreased by one when instantiated in one level below.

The second task of adding deep multiplicities, concerns adding components in the meta-model of Melanee to add support for deep multiplicities. A deep-connection may have an arbitrary number of multiplicities with different potency values to indicate how many connection instances there should have in the lower levels. Deep-multiplicities can be derived from the corresponding connectionend instance's type, with potency of each multiplicity reduced by 1.

The third task of adding deep multiplicities concerns the extension of "wellformedness.evl" rules for deep-multiplicities. Regarding the newly added components discussed above, the following "wellformedness.evl" checks are required:

#### **Potency range checking:**

The potency value of each multiplicity should not be negative (except -1, which stands for infinite ("\*") in Melanee). Only positive numbers (including zero) have a well-defined meaning regarding the potency of features.

#### **Multiplicity potency duplication checking:**

Among all the potency values of all the multiplicities in a connectionend, there should be no duplicate values. Multiplicity declares a number range to indicate how many connections are valid at the specified level. For example, a multiplicity range "2..3" with potency 1 indicates that there should be two to three connections instances at the level below. If there are duplicated potency values, either the specified ranges are redundant, or they conflict with each other. Hence duplicated potency values should be avoided in multiplicities.

### **Multiplicity lower bound and upper bound checking:**

Lower bound should not be larger than upper bound, and lower and upper bounds should not be negative numbers (except for -1, which stands for infinite (“\*”) in Melanee).

### **Connection instances multiplicity checking:**

This is to check whether the number of instances of a given connection is within the range defined in the respective multiplicity constraint.

All the wellformedness.evl validations mentioned above can be implemented with the Eclipse validation framework using an EVL script in Melanee. The first step, however, of implementing deep connections involves extending the meta-model which is discussed in 4.1.1 as a starting point of development.

#### **4.1.1 Adding Feature to PLM.Ecore File**

In order to support deep-connection, there are two parts in meta-model that need modifications: adding potency to moniker and add multiple multiplicities to connectionend. The definition of components relating to meta-model is the responsibility of EMF, and the respective PLM.ecore file contains the meta-model definitions. Hence as the first step, the modification will be performed by editing the PLM.ecore file.

Adding potency to connectionend is simple. We simply add a potency EAttribute to ConnectionEnd. The potency in connectionend is mandatory and must be assigned a value, hence the lower bound and upper bound of the EAttribute “potency” are all 1. The EAttribute type is “EInt” (means integer value in EMF).

Adding deep multiplicities is a little more complicated: first, the original multiplicity support needs to be removed from `connectionend` for it does not support multiple level specification. Since modelers can specify as many levels as required in deep-multiplicities, the number of multiplicities in `connectionend` needs to be dynamic. Inside EClass `"ConnectionEnd"` there are only EAttributes that cannot be defined with dynamic number of multiplicities, hence multiplicity is represented as an independent EClass `"Multiplicity"`, and reference object of the EClass `"Multiplicity"` is placed inside the `connectionend`, i.e., I introduce a "one to many" relationship from `connectionend` to `Multiplicity`. Inside the `Multiplicity` EClass, there are lower bound, upper bound and a potency attribute. These EAttributes are mandatory so their upper bound and lower bounds are 1. Inside `ConnectionEnd` there is a newly added EReference referencing the EClass `"Multiplicity"`. Since the number of multiplicities could be from zero to as many as required, the lower bound of multiplicity EReference is 0 and the upper bound of multiplicity EReference is -1 (-1 means "\*"). The modified meta-model is shown in Fig. 35.

Since the legacy multiplicity is removed, some sections of code that related to legacy multiplicity needs to be removed to prevent errors in runtime calling.

After modifying the `PLM.ecore` file and template `FactoryClass.javajet`, it is necessary to generate code according to modified `PLM.ecore` so that the modifications can take effect in Melanee. Using `PLM.genmode` to reload `PLM.ecore`, and then regenerate:

1. model code under folder `src` in plugin `org.melanee.core.models.plm`;
2. editor code in plugin `org.melanee.core.models.plm.edit`;
3. editor code in plugin `org.melanee.core.models.plm.editor`.





for connectionends should be automatically set so that modelers only need to change the respective attributes in properties view if necessary. However, multiplicities, are referenced in EClass connectionend, and inside the code multiplicities are treated as a list in PLM.ecore file, so it is impossible to create a default multiplicity and store it in the list in PLM.ecore using OCL. Hence, they need to be added to a connectionend's multiplicity list using Java code, which is using customization code in the initialization.

As discussed earlier in Background Chapter, the manipulation of models in GMF is accomplished by using commands instead of directly accessing the model. The creation of connectionend is achieved through a command called “ConnectionEndCreateCommand” in plugin org.melanee.core.modeleditor. This command is generated by the GMF framework, and there are templates for the customization of the generated command code. If the connection is created without a type, we want the connectionend to have a default multiplicity with lower bound 0, upper bound -1 (“\*”) and potency 1. And if the connection is created with its type defined, then the multiplicities need to be instantiated from the connectionend type with all potency values decreased by one. These derived multiplicities do not include the one with potency 0, for potency 0 indicates that such multiplicity should not describe connections in further levels below. In addition to multiplicities, the potency values of monikers will also be automatically decreased by one when a connection instance is derived from a connection type.

The method “doExecuteWithResultImplicitMiddle” defined inside class “ConnectionEndCreateCommand”, is used for creating a connectionend for a connection in Melanee. As introduced earlier in this chapter, multiplicities are referenced in connectionend as a list to dynamically add or remove multiplicities, so adding a default multiplicity to connectionend is achieved by adding one “Multiplicity” instance to the list. The respective code is listed below:

```
protected CommandResult doExecuteWithResultImplicitMiddle(IProgressMonitor monitor, IAdaptable  
info) throws ExecutionException {
```

```

    super.doExecuteWithResult(monitor, info);

    if (!canExecute()) {
        throw new ExecutionException("Invalid arguments in create link command");
        //$NON-NLS-1$
    }

    ConnectionEnd newElement = PLMFactory.eINSTANCE.createConnectionEnd();
    newElement.setDestination((Clabject) source);
    ((Connection) middle).getConnectionEnd().add(newElement);
    newElement.setConnection((Connection) middle);

    ((CreateElementRequest) getRequest()).setNewElement(newElement);

    if (getRequest().getParameter(ImplicitConnectionGraphicalNodeEditPolicy.DSL_TYPE) != null)
        instantiateDSLValues(
            (Connection)
getRequest().getParameter(ImplicitConnectionGraphicalNodeEditPolicy.DSL_TYPE),
            (Connection) middle);
    else {
        newElement.getMultiplicity().add(getDefaultMultiplicity());
        //Call getDefaultMultiplicity() to get a new multiplicity with default
        values
    }

    return CommandResult.newOKCommandResult(newElement);
}

...

...

private Multiplicity getDefaultMultiplicity(){
    org.melanee.core.models.plm.PLM.Multiplicity multi =
    org.melanee.core.models.plm.PLM.PLMFactory.eINSTANCE.createMultiplicity();
    //Create a new multiplicity
    multi.setLower(0); //set default lower boundary to 0 for the new multiplicity
    multi.setUpper(-1); //set default upper boundary to * for the new multiplicity
    multi.setPotency(1); //set default potency to 1 for the new multiplicity

    return multi;
}

```

The if-else statement in the code will examine if the connectionend has a type, if it does, then execution needs to be aborted since another method is used for configuring a connectionend when a type is defined for it. If the connection has no type defined, then an empty “Multiplicity” instance is created. EMF uses factory pattern to create new elements, which can have better extension ability and can reduce complexity whenever

introducing new components in the meta-model. Besides, factory pattern gathers object creation code in one place, and when a change needs to be done to some of the models in the factory, the creation of these objects outside factory pattern can remain unchanged, which reduces complexity in conducting logic changes. Then the default values are set to the corresponding attributes. In the last line of the else section, the multiplicities list inside the connectionend is obtained and a default multiplicity is added to it.

The function above is only responsible for creating one of the connectionends in a connection. Connections usually consist of two connectionends, defining the other connectionend is performed in another function called “createImplicitTargetLink”. The code is similar to that in “doExecuteWithResultImplicitMiddle”, the code is:

```
protected IElementTypeAwareAdapter createImplicitTargetLink(IProgressMonitor monitor, IAdaptable
info) throws ExecutionException {

    implicitConnectionEnd = PLMFactory.eINSTANCE.createConnectionEnd();

    implicitConnectionEnd.getMultiplicity().add(getDefaultMultiplicity());
    //Call getDefaultMultiplicity() to get a new multiplicity with default values

    return new EObjectAndElementTypeAdapter(implicitConnectionEnd,
        PLMElementTypes.ConnectionEnd_4036, ConnectionEndEditPart.VISUAL_ID);
}
```

After updating the code of these two methods, the newly created connectionends can have default deep-multiplicities during initialization.

The next step in deep multiplicities development is to support the creation of a connectionend instance by deriving multiplicities from its type. This feature is also done in class “ConnectionEndCreateCommand”, in a method called “instantiateDSLValues”. The key code of “instantiateDSLValues” is:

```
for (int i = 0; i < typeConnectionEnds.size(); ++i) {
    Clabject destination = (Clabject) (i == 0 ? source : target);
    ConnectionEnd instanceConnectionEnd = getConnectionEndToDestination(instanceConnection,
                                                                    destination);
}
```

```

instanceConnectionEnd.setMoniker(typeConnectionEnd[i].getMoniker());
instanceConnectionEnd.setNavigable(typeConnectionEnd[i].isNavigable());
instanceConnectionEnd.setKind(typeConnectionEnd[i].getKind());
instanceConnectionEnd.getMultiplicity().clear();

int typePotency=typeConnectionEnd[i].getPotency();//the potency of its type

if(typePotency==-1){
    instanceConnectionEnd.setPotency(-1);
}else if(typePotency>0){

    instanceConnectionEnd.setPotency(typeConnectionEnd[i].getPotency()-1);
}

for (Multiplicity m : typeConnectionEnd[i].getMultiplicity()) {
    if (m.getPotency() > -2 && m.getPotency() != 0) {(
        //This is to consider the potency could be -1 for representing infinite
        (*) in Melanee
        Multiplicity multi = PLMFactory.eINSTANCE.createMultiplicity();
        multi.setLower(m.getLower());
        multi.setUpper(m.getUpper());

        if (m.getPotency() == -1)
            multi.setPotency(-1);
        else
            multi.setPotency(m.getPotency() - 1);

        instanceConnectionEnd.getMultiplicity().add(multi);
    }
}

instanceConnectionEnd.setType(typeConnectionEnd[i]);
}

```

In the “for” loop, every connectionend is examined. First, the destinations of connectionend are set. Then for the potency, its value is examined against -1, for if it is -1, then the potency of instance will also be -1, which means infinite, like its type. If the potency is not -1, simply set the instance’s potency according to the type’s potency. For the multiplicities, there is also a for loop to iterate through all of them. Again, there are if clauses to exam the potency so that multiplicity with potency 0 will not be considered. For the other multiplicities, if the potency is -1, then the instance will derive this multiplicity with potency -1; if the potency is larger than 0, then the instance will inherit this multiplicity with potency value decreased by one.

All these changes are written in template called “CreateLinkCommand.xpt” located under folder `templates.xpt.diagram.commands` in plugin `org.melanee.core.models.gmf` to prevent customized code overwritten when generating control code in GMF.

#### 4.1.3 Adding Multiplicity Control Tab in Properties View

Unlike potency values, multiplicity constraints can be added and removed dynamically. Modelers should also be able to select a specific multiplicity to modify it or delete it. All these requirements imply that treating multiplicities simply like other attributes is not sufficient.

Hence in Melanee a way of setting multiplicities in another tab inside a regular property sheet is proposed. When selecting a connectionend in the graph, the Multiplicity Tab inside a property sheet can show the corresponding multiplicities list, and the user may add new one or select one of the existing multiplicities for removal or editing (see Fig. 36).



Multiplicity	Multiplicity: 2..2 <sup>3</sup>	▼	Add	Remove
Name	Multiplicity: 0..-1 <sup>1</sup>			
Lower	Multiplicity: 2..2 <sup>3</sup>			
Upper	Multiplicity: 2..3 <sup>2</sup>			
Potency	2			
	3			

Fig. 36 Multiplicity tab when selecting a connectionend in the model graph.

The plugin “org.melanee.core.modeleditor” allows extension of existing property sheet sections by registering extensions in file `plugin.xml`, so that during runtime when choosing this tab, the corresponding code will be executed to show the Multiplicity Control Tab.

The class containing the multiplicity tab code is placed under folder custom-src.org.melanee.core.modeleditor.custom.propertySheet in plugin org.melanee.core.modeleditor. All the files under custom-src are customized code and will not be overwritten during code generation by GMF.

The class containing the multiplicity tab code is called “MultiplicityPropertiesSection” and it extends class “AbstractPropertySection”. AbstractPropertySection is an abstract class for defining a properties view sheet. As it is shown in Fig.36, there are several components: a list to select from, two buttons and a TableView that contains details of the selected multiplicity constraint.

According to the comments in source code of “CCombo” (which is a combo list), “CCombo” is initialized in the setInput() method. In the method, the first step is to resolve the selected structure in the graph and to see if it is a connectionend, this is because if the selected diagram is not a connectionend, then it is not necessary to prepare the multiplicities properties sheet. Then in the code, all the items in the CCombo will be removed, and then the multiplicities of the selected connectionend will be obtained and loaded into CCombo. This is to make sure that the multiplicities showed in the properties view are the most up-to-date ones. The “TableView” section will show the first multiplicity as default. Every time a modeler adds or modifies a multiplicity, the view will refresh itself to render the most up-to-date information of the model. I have added a “selectedIndex” variable to track the index of user’s last selection. Otherwise every time the view is refreshed when modifying a multiplicity, the “TableView” would show details of the first multiplicity in the list, and the modeler’s selection will be lost. The corresponding code is as follow:

```
if (multiplicitySelectionCombo.getItems().length > 0) {  
    if (selectedIndex > multiplicities.size() - 1)  
        selectedIndex = 0;  
  
    multiplicitySelectionCombo.select(selectedIndex);  
  
    List<String> multiplicitiesValues = new ArrayList<String>();
```

```

multiplicitiesValues.add("Lower=" + String.valueOf(multiplicities.get(selectedIndex).getLower()));
multiplicitiesValues.add("Upper=" + String.valueOf(multiplicities.get(selectedIndex).getUpper()));
multiplicitiesValues.add("Potency=" +
                        String.valueOf(multiplicities.get(selectedIndex).getPotency()));

viewer.setInput(multiplicitiesValues);
viewer.refresh();
}else{
    multiplicityPropertiesViewReset();
}

```

In the “TableView” class, every time the value of a multiplicity changes, the change should go into the model immediately to prevent information loss. This update to the model is done through “ReplaceCommand”. Again, when the value is changed the view needs to be refreshed to render the most up-to-date information:

```

Multiplicity newMultiplicity = PLMFactory.eINSTANCE.createMultiplicity();
newMultiplicity.setLower(multiplicity.getLower());
newMultiplicity.setUpper(multiplicity.getUpper());
newMultiplicity.setPotency(multiplicity.getPotency());

if (key.equals("Lower")) {
    newMultiplicity.setLower(Integer.valueOf(newAttrValue));
} else if (key.equals("Upper")) {
    newMultiplicity.setUpper(Integer.valueOf(newAttrValue));
} else if (key.equals("Potency")) {
    newMultiplicity.setPotency(Integer.valueOf(newAttrValue));
}

TransactionalEditingDomain domain =
    TransactionUtil.getEditingDomain(selectedConnectionEnd);

Command editMultiplicityCommand = new ReplaceCommand(domain, selectedConnectionEnd,
PLMPackage.eINSTANCE.getConnectionEnd_Multiplicity(), multiplicity, newMultiplicity);

domain.getCommandStack().execute(editMultiplicityCommand);

viewer.refresh();

```

The two buttons, “addButton” and “removeButton”, each of these two buttons are given a SelectionListener. Each time a button is pressed, the associating method will be called. In addButton listener there is extra code to ensure that when a new multiplicity is added, the index for the TableView is recorded in “selectedIndex” to show this newly added multiplicity after refresh. The creation of a new multiplicity is again achieved through using



the create command. After creating the new command, “commandStack” is called to execute the commands:

```
Command addMultiplicityCommand = AddCommand.create(domain, selectedConnectionEnd,  
PLMPackage.eINSTANCE.getConnectionEnd_Multiplicity(), newMultiplicity);  
domain.getCommandStack().execute(addMultiplicityCommand);
```

#### 4.1.4 Rendering View of Deep-Connection

Rendering view of the potency values of monikers and multiplicities proved to be more difficult than expected. There are two reasons for this: first, the multiplicities have dynamic length; second, the structure of the information needed to conveyed to modelers is more complex than what has been envisaged by the designers of GMF.

It is desirable to show moniker and multiplicity potencies using a format consistent with the style used by Melanee for the potencies of clabjects and their features.

Melanee shows potency values on the upper right corner of name label, as shown in Fig. 37. This is achieved by assigning multiple name labels separately: A name label for showing the clabject name and another potency label to show the corresponding potency value. These labels are formatted in a table layout to arrange their positions.

However, the same mechanism to show moniker potency and multiple multiplicities is not feasible. The multiple labels need to be organized inside a layout, and such a layout is contained inside a shape, for instance the connection name and potency label of connections are inside a hexagon-like shape, and the entity name and potency of clabjects are defined inside a rectangle-shape. For connectionends, however, it is impossible to put shapes inside a solid line. Furthermore, GMF does not support referencing a label's position according to the position of another label so using separate labels for moniker and potency is not feasible here.

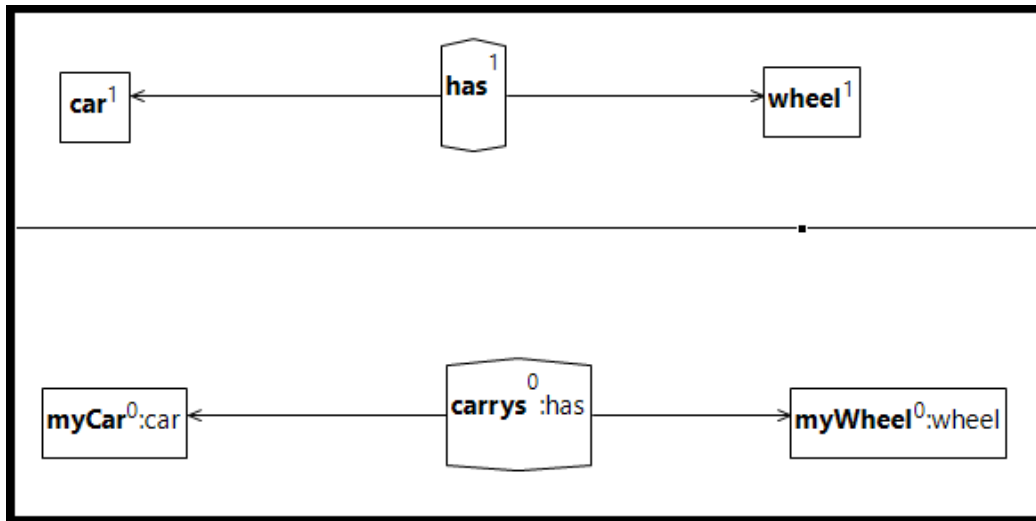


Fig. 37 Potency position in the original Melanee design.

Thus, the name and the potency for a moniker must be arranged inside one label.

The labels are defined in PLM.gmfgraph and then generated into Java code in plugin org.melanee.core.modeleditor. The file Editpart is generated under folder src.org.melanee.core.modeleditor.edit.parts in plugin org.melanee.core.modeleditor. Editpart classes are used to manage visualization of the model graph, and their corresponding code is generated by PLM.gmfgen. In order to customize the visualization code of monikers and multiplicities, we have to customize the code in the corresponding Editpart file.

Class “ConnectionEndEditPart” is responsible for the initialization and visualization of connectionend graph. The method controlling the visualization layout is called “updateLMLRenderingView”. In this method, the layout configuration and value setting of WrappingLabels are performed.

A WrappingLabel holds textual information to be visualized as part of the model. It is specifically designed to hold a simple line of text information and an icon. It does not support layout information inside the label itself for showing potency properly which involves an emulation of a superscript font. However, I need to render a layout of information instead of a single line of text. After carefully studying and tried many ways and failed, finally I found a way of showing layout information inside the WrappingLabel by exploiting the inheritance hierarchy of class “WrappingLabel”. Each WrappingLabel can also be viewed as a RectangleFigure for it extends RectangleFigure. Each WrappingLabel has a border, one can use setOutline(false) to set the border to be invisible. First the WrappingLabel is cast into a RectangleFigure, and then changed the contents of RectangleFigure to use a layout for rendering the information.

To show the name and potency in the way as expected, I decided to use a GridLayout for this. The layout is like what is shown in Fig. 38. I divided the layout into 2 columns and 2 lines: the moniker name occupies the first column and both lines; the potency value occupies the second column of the first line. The second column of the second line is left empty.

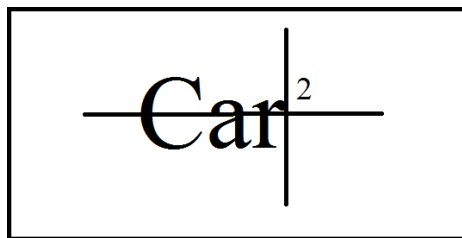


Fig. 38 GridLayout for showing moniker and potency.

The code of defining the above layout is long, hence only key aspects of the code are listed here:

```

int multiSpaceCount = 0;
int monikerSpaceCount = 0;
...

...

WrappingLabel connectionendNameLabel = ((WrappingLabel) getFigure().getChildren().get(0));

sb = new StringBuilder();

RectangleFigure innerRectangle1 = new RectangleFigure();

if (connectionendNameLabel.getChildren().size() > 1) {
    innerRectangle1 = (RectangleFigure) connectionendNameLabel.getChildren().get(1);
}

innerRectangle1.setFill(false);
innerRectangle1.setOutline(false);

GridLayout layoutInnerRectangle1 = new GridLayout();
layoutInnerRectangle1.numColumns = 2;
layoutInnerRectangle1.makeColumnsEqualWidth = false;
layoutInnerRectangle1.horizontalSpacing = 0;
layoutInnerRectangle1.verticalSpacing = 0;
layoutInnerRectangle1.marginWidth = 0;
layoutInnerRectangle1.marginHeight = 0;
innerRectangle1.setLayoutManager(layoutInnerRectangle1);

if (self.getMoniker() != null && !self.getMoniker().equals("")) {
    RectangleFigure nameRectangle2 = new RectangleFigure();

fFigureConnectionEndMonikerPotencyFigure.setText("" + (self.getPotency() == -1 ? "" : self.getPotency()));

...

if ((self.getUserSpecifiedTypeName() != null && !self.getUserSpecifiedTypeName().equals(""))
    || (self.getType() != null)) {

    ...

    ...

    monikerSpaceCount += fFigureMonikerRectangle.getText().length();
}

connectionendNameLabel.add(innerRectangle1);

```

Another feature of WrappingLabel is that it can dynamically adjust its visible width to wrap the text inside. This is a convenient feature but not for my purposes here. Since I cast it into a RectangleFigure and changed its content, there is no text defined by calling the setText() method. As consequence of this, the WrappingLabel considers itself having no text hence showing zero width of its contents, which makes its contents invisible. To resolve this, I adopted a simple way of counting the length of the moniker and potency text and setting the corresponding WrappingLabel with exact same number of spaces by calling setText(). By doing so the WrappingLabel considers itself to have text so it will adjust its visible width, thus the contents I put into the RectangleFigure will be visible to modelers.

After resolving the layout problems in WrappingLabel, the next step is to consider potency value visualization problem. Even though a potency value is non-negative by definition. In Melanee -1 is used to indicate “star potency”. Hence in the code, the potency value will be compared against -1. If it is equal to -1, “\*” will be presented to modelers instead of -1.

Visualizing multiplicities and their potencies is very similar to visualizing monikers and their potencies: I again cast the WrappingLabel to RectangleFigure, iterate through the multiplicities and set the layout accordingly; count the text length and call setText() with counted number of spaces in order to make the contents in RectangleFigure visible.

The respective code of visualizing multiplicities is a very similar to that of the moniker, but there is a major difference: for the multiplicities label, I structure it into a number of columns; the number of columns are decided according to how many multiplicities there have defined in a connectionend. For each column, another layout is kept, showing multiplicity and potency in the format as expected. The key parts of the code rendering multiplicities view lists as follow:

```
WrappingLabel multiLabel = ((WrappingLabel) getFigure().getChildren().get(1));
```

```

RectangleFigure innerRectangle0 = new RectangleFigure();

if (multiLabel.getChildren().size() > 1) {
    innerRectangle0 = (RectangleFigure) multiLabel.getChildren().get(1);
}

...

...

for (int i = 0; i < multiSize; i++) {
    Multiplicity multi = multis.get(i);

    RectangleFigure nameRectangle1 = new RectangleFigure();

    nameRectangle1.setFill(false);
    nameRectangle1.setOutline(false);

    if (i == 0) {
        fFigureConnectionNameFigureText = multi.getLower() + ".."
            + (multi.getUpper() == -1 ? "*" : multi.getUpper() + "");
    } else {
        fFigureConnectionNameFigureText = "|" + multi.getLower() + ".."
            + (multi.getUpper() == -1 ? "*" : multi.getUpper() + "");
    }

    ...

    ...

    multiLabel.add(innerRectangle0);
    ...

    ...

```

Again, all coding must be written into the template so that if there are future generations of code using PLM.gmfgen, the changes will not be overwritten. The corresponding template is called “LinkEditPart.xpt” and is located in plugin org.melanee.core.models.gmf and under folder templates.aspects.diagram.editpart.

The last component needs to be visualized for deep-connection support is the moniker type name in connectionend. The type name for moniker is derived from the

connectionend's type. This type name is not shown if the connectionend has not been specified a type yet. The moniker type name is shown in another WrappingLabel positioned inside a RectangleFigure class after moniker WrappingLabel. For every connectionend, there will be examination against the definition of its type first, if the type has been defined, then show the corresponding type name in the WrappingLabel in the connectionend instances; if the type has not been defined, then the type name in WrappingLabel for moniker type name will not be shown in the graph. The code for visualizing moniker type name lists as follow:

```
RectangleFigure monikerRectangle = new RectangleFigure();

monikerRectangle.setFill(false);
monikerRectangle.setOutline(false);

GridData constraintmonikerRectangle = new GridData();
...

...

if (self.getType() != null) {
    connectionendTypeName = (self.getType().getMoniker() == null ||
self.getType().getMoniker().equals(""))? "" : (":" + self.getType().getMoniker());
}

...

...

fFigureMonikerRectangle.setText(connectionendTypeName);

fFigureMonikerRectangle.setFont(FFIGUREINHERITEDCONNECTIONENDMONIKER_FONT);

...

...
```

After implementing the above features for deep connections, the model in Melanee looks like what is shown in Fig. 39. In the model, we can add an arbitrary number of multiplicities, derive potencies from a connectionend type, and examine potency values for all respective features in the model.

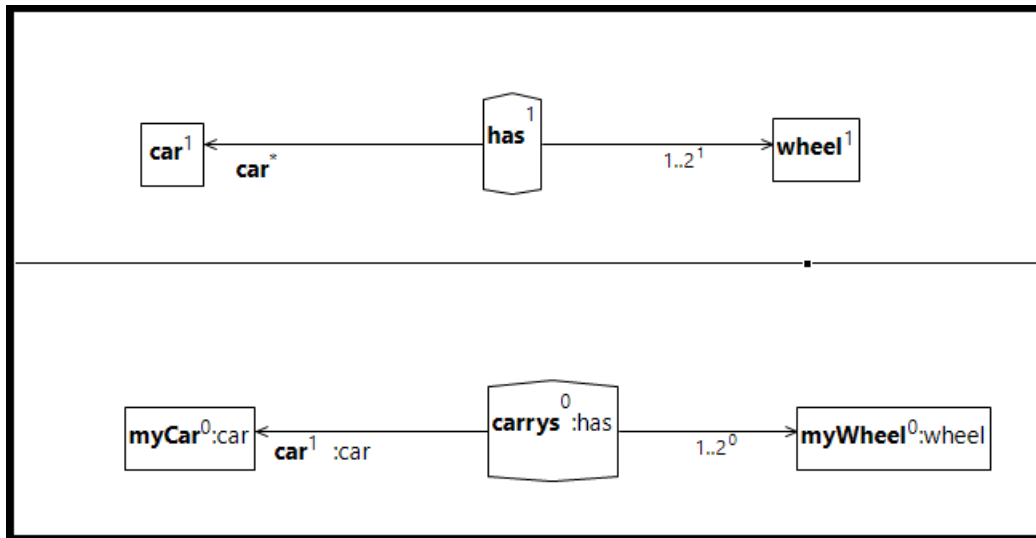


Fig. 39 Model created by Melanee after the implementation of deep-connection features.

#### 4.1.5 Deep-Connection Validations

Validations (or wellformedness validations) in GMF are performed through EVL script. The validation of deep connections is implemented in script “wellformedness.evl” in plugin `org.melanee.core.models.plm.validation`.

The EVL is very like OCL, yet in EVL it is possible to call an OCL method defined in the meta-model. Calling one of the methods is simple: each constraint is placed under a context. In the constraint, “self” is used to reference the object being evaluated. For example, in context `ConnectionEnd`, “self” references a `connectionend` that is being validated.

Most validations are simple except for validation “`connectionendMultiplicityChecking`”. This constraint is required to check if the number of connection instances complies with the ranged specified by the respective deep- multiplicity. The constraint has been formalized as follows:



```

constraint connectionendMultiplicityChecking {
guard: self.multiplicity->select(r | r.potency = 1)->size() > 0
--if there is not multiplicity with potency 1 defined, there is no need to validate

check{
    var oppositeClabjectInstances =
self.connection.connectionend->select(c|c<>self)->first().destination.getInstance();
    --Get the instances of clabject that connects on the other side of the connection

    --multiplicity lower bound with potency 1
    var multiplicityLowerValue=self.multiplicity->select(r|(r.potency = 1))->first().lower;
    --multiplicity upper bound with potency 1
    var multiplicityUpperValue=self.multiplicity->select(r|(r.potency = 1))->first().upper;
    --Get the connection which is the type
    var connectionType=self.connection;

    -- if multiplicityUpperValue is minus -1 then it means no upper bound so upper doesn't have to be
    checked.
    if (multiplicityUpperValue== -1)
        return oppositeClabjectInstances->forAll(clabjectInstance |
            (multiplicityLowerValue<=(clabjectInstance.getConnections().select(typeConnections |
                typeConnections.getTypes().includes(connectionType)).size())));

    -- validate each instance of the oppositeClabjectInstances has the correct number of connections
    between multiplicityLowerValue and multiplicityUpperValue
    else
        return oppositeClabjectInstances->forAll(clabjectInstance |
            (multiplicityLowerValue<=(clabjectInstance.getConnections().select(typeConnections |
                typeConnections.getTypes().includes(connectionType)).size()))
            and ((clabjectInstance.getConnections().select(cc |
                cc.getTypes().includes(connectionType)).size())<=multiplicityUpperValue));

    }
    message: 'Number of instances of connection \''+self.connection.name+'\' should be between
    \''+multiplicityLowerValue+'\' and \''+multiplicityUpperValue+'\'
}

```

The first line is the declaration of the constraint. In the second line, “guard” is to make sure the connectionend being validated meets the prerequisite that there are multiplicities defined and at least one of the multiplicity has potency 1. If this condition is met, then the code in check section will be executed. Here I defined the prerequisite of having a multicity specification with potency 1, because I only validate one level below against an adjacent level. Multiplicities with higher potencies will have their potencies reduced upon instantiation, and eventually their potency will reach 1 at some level. Checking multiplicity constraints only against directly one level below, avoids complicated navigation to lower levels and avoids checking multiplicity constraints multiple times for each level they reappear in.

In the “check” section the number of existing connection instances are compared against the lower and upper bounds of the relevant multiplicity constraint. The message in the constraint is designed to provide meaningful information to modelers so that a modeler can understand in which “connectionend” the multiplicity constraint is violated.

## 4.2 Connection Conformance

The main goal of developing connection conformance is to provide support for connection type identifications in the system, so that Melanee can support an explanatory way of constructing model.

According to multi-level conformance principles in [3], all connections and entities in all levels except the topmost level will be checked to examine if they have a type that they conform to or not. If a type hasn’t been set, then Melanee ought to consider all possible types in the level above the element in question (According to the design discussed in Chapter 3.2). In such cases, Melanee should then present all possible type candidates ordered from the most likely matching type to the least likely. Messages containing information regarding to which components don’t match and how to amend them to achieve a match after all will be shown to modelers through message that are present to molders in the graph and a “message” section in Problem Properties View. The Problem Properties View offers a quick-fix section that includes a fix solution. Due to limitations of the GMF framework, I had to develop a second stage dialog that allows users to choose a potential fix (repair action) to establish a proper type for the connection as discussed in the Design Chapter.

In this section, I will first introduce the functions to calculate a matching score so that I can produce an ordered list from most likely to least likely matches. Then the development of Connection Conformance functions will be introduced. Chapter 4.2.4 describes t h e

native Java classes I had to develop to implement the second-stage dialog in quick-fix selection.

#### 4.2.1 userSpecifiedTypeName

Standard Melanee does not allow users to specify a type name for clabjects. In standard Melanee a user may only connect a clabject to its type via the graphical interface, but there is no way to specify the name of an intended type that may not exist yet. Therefore, it was necessary to add a respective attribute that can hold the type name to the meta-model. I added an EAttribute called “userSpecifiedTypeName” to the Clabject Context in PLM.ecore. Since the connectionends are not treated as clabjects for it does not inherit from “Clabject”, I had to add a “userSpecifiedTypeName” attribute to EClass ConnectionEnd in PLM.ecore as well.

After extending the meta-model, to the visualization of the changed elements must be changed to account for the newly added “userSpecifiedTypeName”. Melanee uses a so-called “designation service” to provide names for clabjects. It calculates dynamically the name of each clabject’s type, and shows it in the clabject after the clabject’s name. The reason it is calculated dynamically is that a clabject can be defined with a type or without a type, and when the type is defined, the type’s name can be changed by modelers as well. In order to provide the most up-to-date type name in the clabject instances, the type name has to be calculated dynamically. Since an EAttribute is added to meta-model to allow modelers to specify a type name when a type has not been specified for the clabject through the graphical interface, Melanee needs to be changed to use “userSpecifiedTypeName” to show type’s name when a type hasn’t been set, and use type’s name if a type has been set for the clabject. I therefore added a method called “getClassificationDesignationString” to java class “DesignationService”. When “createClassificationDesignationString” is called to acquire the type’s name, this new method will be called instead of the original one which has the name “evaluate()”.

When the value of “userSpecifiedTypeName” changes, Melanee needs to automatically refresh the corresponding part of the graph to render the most up-to-date information. As discussed in the Design Chapter, EMF uses a notification mechanism to notify subscribed clients about changes to model using “Observer Patter”. I therefore had to add respective customized notification code to trigger a refresh of graph in class “EntityEditPart”, there is a method called “handleNotificationEvent”. The customized code needs to be added to this method because it handles notification events for the graph. To trigger a refresh, first the changed clabject must be identified and obtained. Then the changed feature needs to be resolved, which is finding what EAttribute has been changed. If the changed feature is the “userSpecifiedTypeName”, a refresh must be triggered to refresh to the corresponding part of the graph. All changes to file EntityEditPart must again be recorded in templates accordingly to prevent code lost like discussed in the Background Chapter. The key code regarding refreshing in file EntityEditPart is as follow:

```
if (changedFeature.getName().equals("userSpecifiedTypeName")) {
    PLMDiagramEditor plmEditor = (PLMDiagramEditor) ((DiagramEditDomain)
        ((IGraphicalEditPart)this).getDiagramEditDomain()).getEditorPart();

    Clabject changedEntity = (Entity) notification.getNotifier();

    List featureEditParts =
        plmEditor.getDiagramGraphicalView().findEditPartsForElement(EMFCoreUtil.getProxyID(changedEntity), NodeEditPart.class);

    if (featureEditParts.size() > 0) {
        IGraphicalEditPart featureEditPart = (IGraphicalEditPart) featureEditParts.get(0);

        if (featureEditPart instanceof org.melanee.core.modeleditor.edit.parts.EntityEditPart)
            ((org.melanee.core.modeleditor.edit.parts.EntityEditPart)
                featureEditPart).updateView(IVisualizationServiceBase.FORMAT_GRAPH);

        if (featureEditPart instanceof Entity2EditPart)
            ((Entity2EditPart)
                featureEditPart).updateView(IVisualizationServiceBase.FORMAT_GRAPH);
    }
}
```

Above code can trigger entities to refresh automatically. However, class “DesignationService” is only responsible for the calculation of entity type name. Hence, even with the above code in place, connections are still not properly automatically refreshed. The reason for this is that Melanee treats connections differently from entities.

Responding to a change of a connection type name needs to be performed in “PLM.gmfmap”, in one of the “Expression Label” under “Child Reference” of Connection Context. The calculation of attribute value in “Expression Label” is performed through OCL. And the customized part of setting connection lists as follows:

```

if (renderClassification) then
    //Using if statement to determine if a type has already been defined

    if (Classification.allInstances()->select(i | i.instance = self)->size() > 0) then
        //If a type has been defined, then the type's name will be returned as type
        name for the connection instance

        ' '.concat(Classification.allInstances()->select(
                                i | i.instance = self)->asOrderedSet()->first().type.name)
    Else
        // If a type has been defined, then value of userSpecifiedTypeName will be
        returned as type name for the connection instance.

        if (self.userSpecifiedTypeName.ocllsUndefined() or
                                self.userSpecifiedTypeName="") then
            "
        else ' '.concat(self.userSpecifiedTypeName)
        endif endif else
            "
        endif
    endif

```

The OCL code examines if the type of the connection has been defined or not, if there is a type, then use the type's name; otherwise, use the name defined in “userSpecifiedTypeName” to show the type name for the connection.

#### 4.2.2 Calculating the Matching Score (Weighted Sum)

The process of obtaining all the potential types in one level above is performed in `wellformedness.evl`. Defining all the calculation and reordering functions inside the critique is not a good idea: different logics with different purposes would be mixed together. For this reason, I used the Epsilon Object Language (EOL) to define “operations” in “`wellformedness.evl`” script. By doing so I could separate the complex matching score calculation into many small, independent operations with each sub-operation fulfilling only one specific part of the weighted sum calculation process. This design did not only ease the construction of the function but also allows parts of it to be changed independently of other parts.

The hierarchy of the connection type conformance matching score calculation operations is as follow:

$$\begin{aligned}\text{weighted sum} &= \text{matchConnectionType} + \text{matchConnectionEnds} \\ &= \text{matchConnectionType} + (\text{matchConnectionEnd}_1 + \text{matchConnectionEnd}_2 + \\ &\quad \text{matchConnectionConnectee}_1 + \text{matchConnectionConnectee}_2)\end{aligned}$$

Each of the above sub-functions target a specific component comparison in the TS [7]. In order to achieve better performance, when calculating the weighted sum, the heuristic messages regarding modifying instances’ type names to match possible connection type are constructed along with the calculation of the weighted sum. The matching score and its corresponding modification message are stored together with the potential connection type itself in one object:

`MatchingResultObject = {matching score, connection type candidate, message}.`

Now I can sort a list of such objects according to their matching score and still retain the integrity between the score, the type candidate and the corresponding messages. I used

the EOL type “Sequence” to store an ordered list of such matching result objects. The “match” function is the main function for calculating the matching score. The code of match is as follow:

```
operation Connection match(conInstance : Connection , conType : Connection):Sequence{
    var result=Sequence{};

    //Break the matching process down into matching connectionend component
    var matchofConnectionEndsResult=self.matchConnectionEnds(conInstance ,
                                                                conType);
    var matchofConnectionEnds=matchofConnectionEndsResult.at(0);
    var matchofConnectionEndsMsg=matchofConnectionEndsResult.at(1);

    //Break the matching process down into matching connection name component
    var matchofConTypeResult=self.matchConnectionType(conInstance , conType);
    var matchofConType=matchofConTypeResult.at(0);
    var matchofConTypeMsg=matchofConTypeResult.at(1);

    //Get the overall weighted sum, aka matching score of the connection type
    var NOofMatch=matchofConType + matchofConnectionEnds;
    result.add(NOofMatch);

    result.add(conType);

    result.add(matchofConTypeMsg+matchofConnectionEndsMsg);

    return result;
}
```

This “match” operation has two parameters: the connection instance which has not been assigned to a type yet, and one potential connection type for comparison. The operation will return a sequence of matching result objects that in turn contain a matching score, the connection type candidate and corresponding messages that are relevant to modification of the connection instance to match the possible connection type.

In “match”, two other operations are called: “matchConnectionType” and “matchConnectionEnds”. The “matchConnectionType” is for comparing the connection name component. According to the design the weight of a successful match of connection name is 2. The code of function “matchConnectionType” is as follow:

```

//Weight for a match
var matched=2;
//Penalty for a mismatch
var mismatched=-1;

//Obtain the connection type's name
var conInstanceTypeName=conInstance.userSpecifiedTypeName;
var conTypeName=conType.name;

//process the connection instance's name for it could be null or ''
if(conInstanceTypeName=null or conInstanceTypeName='')
    conInstanceTypeName=null;

//process the connection type's name for it could be null or ''
if(conTypeName=null or conTypeName='')
    conTypeName=null;

//If connection type name not specified
if(conInstanceTypeName=null and conTypeName=null){
    connectionMatch=0;
}
//If connection type name is blank, then the connection name will be used to compare
against the type, penalty of -0.5 will be added to the sum
else if(conInstanceTypeName=null and conTypeName<>null){
    var connectionName=conInstance.name;

    if(connectionName=null or connectionName='')
        connectionName=null;

    if(connectionName=conTypeName)
        connectionMatch=matched;
    else
        connectionMatch=-0.5;
}

//If connection type name not specified and connection instance type name specified
else if(conInstanceTypeName<>null and conTypeName=null){
    connectionMatch=0;
}
//If connection type name matched
else if(conInstanceTypeName=conTypeName){
    connectionMatch=matched;
}
//If connection type name matched
//If connection type name not matched
else{
    msg=', change type in connection \'' + conInstance.name + '\' to \'' +
        conType.name+'\'';

    connectionMatch=mismatched;
}

//If the potency failed to be smaller than the potential type's potency, it is
considered a mismatch
if((conType.potency<>-1 and conInstance.potency=-1) or (conType.potency<>-1 and
    conInstance.potency<>-1 and conType.potency <= conInstance.potency)){

```



```

        connectionMatch=mismatched;

        msg=msg + ', ' + conInstance.name+'\'s potency should be smaller than \'' +
conType.potency + '\'';
    }

```

The first variable “conInstanceTypeName” is used to store the value of userSpecifiedTypeName, then it is compared against “” and “null”. Both comparisons are used because in GMF, if an attribute hasn’t been assigned a value, then it is “null”; but if the attribute has been assigned a value and then the value is deleted, the attribute value is “” instead of null, but they both mean “No user specified name”. The same applies to variable “conTypeName” as well. This is what GMF decided to set the attribute as null if it has never been given a value, and set to “” if the value had been given and deleted, and this is the fact that I must deal with in the development.

In the calculation process, if the connection type name is not specified, then the connection name will be used to compared against the potential connection type’s name for calculation. Weight -0.5 is for the situation when connection type has been given a name but in connection instance, the connection type name has not been defined, which is discussed in the Design Chapter. The code for this lists as follow:

```

//If connection type name is blank, then the connection name will be used to compare
against the type, if mismatched, penalty of -0.5 will be added to the sum
else if(conInstanceTypeName=null and conTypeName<>null){
    var connectionName=conInstance.name;

    if(connectionName=null or connectionName='')
        connectionName=null;

    if(connectionName=conTypeName)
        connectionMatch=matched;
    else
        connectionMatch=-0.5;
}

```

The potency in the connection name component needs also to be compared to see if the instance’s potency is smaller than that of the potential type’s, otherwise the two components will be considered a mismatch. This comparison needs to take the potency

value -1 into consideration. A value of -1 means “infinite potency” in Melanee so simple comparing potency of instance and potency of type to see if instance’s potency is smaller than that of the type is not feasible here. Potency also needs to be matched for a success component match. In case of a mismatch, the weight will be overwritten to its according penalty, and the message shall be extended with tips regarding to how to modify potency to match instance to this specific potential type.

Another operation “matchConnectionEnds” is also called in operation “match”. The responsibility of matchConnectionEnds is to compare each of the connectionends in connection instance against each of the connectionends in connection type, i.e. to perform four individual matching attempts rather than just two. The reason for this is because in Melanee it is impossible to determine a connectionend as the “left” one or the “right” one, there is no “direction” for connections in Melanee. Consider the model shown in Fig. 40. The connectionend “myCar” should be the instance of connectionend “car”, for their components matched. However, when comparing the connectionends they occur in in a random order, as opposed to a “left to right” ordering. Hence, a single comparison may end up comparing connectionend “wheel” in the connection instance with “car” in the connection type. Therefore, the accurate way to calculate a matching score for the of the two connectionends is to consider both “orientations”, i.e., compute two scores and then see which results in a higher matching score.

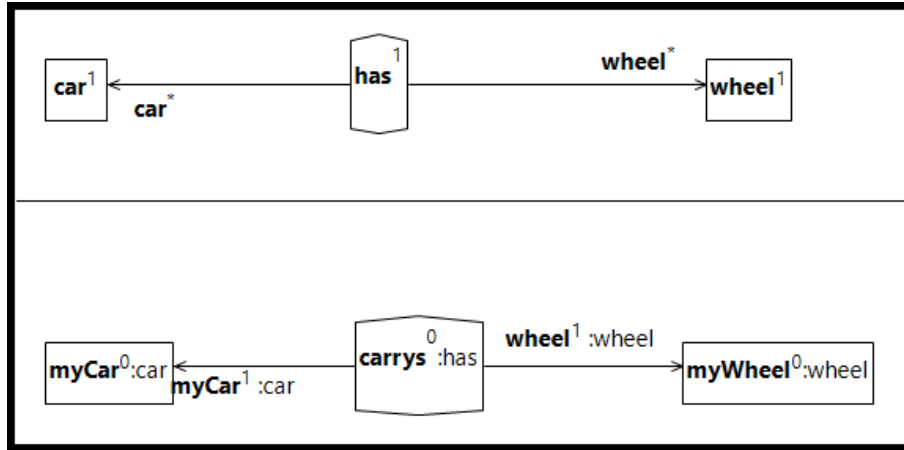


Fig. 40 Two connections with different connectionends.

The code of “matchConnectionEnds” is as follow:

```
//Obtain left connection end of connection instance
var leftConInstanceEnd=conInstance.connectionend->asSequence()->first();
//Obtain right connection end of connection instance
var rightConInstanceEnd=conInstance.connectionend->asSequence()->last();

//Obtain left connection end of connection type
var leftConTypeEnd=conType.connectionend->asSequence()->first();
//Obtain right connection end of connection type
var rightConTypeEnd=conType.connectionend->asSequence()->last();

//Conduct a LL and RR matching
var leftLeftMatchResult=self.matchConnectionEnd(leftConInstanceEnd ,
leftConTypeEnd);
var leftLeftMatch=leftLeftMatchResult.at(0);
var leftLeftMatchMsg=leftLeftMatchResult.at(1);

var rightRightMatchResult=self.matchConnectionEnd(rightConInstanceEnd ,
rightConTypeEnd);
var rightRightMatch=rightRightMatchResult.at(0);
var rightRightMatchMsg=rightRightMatchResult.at(1);

//Conduct a LR and RL matching
var leftRightMatchResult=self.matchConnectionEnd(leftConInstanceEnd ,
rightConTypeEnd);
var leftRightMatch=leftRightMatchResult.at(0);
var leftRightMatchMsg=leftRightMatchResult.at(1);

var rightLeftMatchResult=self.matchConnectionEnd(rightConInstanceEnd ,
leftConTypeEnd);
var rightLeftMatch=rightLeftMatchResult.at(0);
var rightLeftMatchMsg=rightLeftMatchResult.at(1);
```

```

//Compare the results of the swaping and get the higher one as the result
if((leftLeftMatch + rightRightMatch) > (leftRightMatch + rightLeftMatch)){
    connectionEndMsg=leftLeftMatchMsg+rightRightMatchMsg;
    connectionEndMatches= leftLeftMatch + rightRightMatch;
}
else{
    connectionEndMsg=leftRightMatchMsg+rightLeftMatchMsg;
    connectionEndMatches= leftRightMatch + rightLeftMatch;
}

result.add(connectionEndMatches);
result.add(connectionEndMsg);

```

There are two sets of possible combinations: left with left and right with right, or left with right and right with left. In the end the weight calculated of each set of combinations are compared to get the larger set, which will be considered the match between instance connectionends and the type connectionends. The returned Sequence contains the matching score and corresponding messages that relate to the match obtained in “matchConnectionEnds”.

In “matchConnectionEnds”, one operation is called which is “matchConnectionEnd”. The purpose of “matchConnectionEnd” is to compare the moniker components, and call the operation “matchConnectionConnectee”, add then two sub-results together and return them as an overall matching score for the connectionends.

The comparison of moniker components in operation “matchConnectionEnd” works in similar way to that of operation “matchConnectionType”, except for monikers the weight for a match is 1 instead of 2 for the connection name like what is discussed in the Design Chapter. The “userSpecifiedTypeName” and potential type’s name will be examined before comparison to account for the possibility of “” and null. Then the names and the potencies are compared to construct messages.

Inside operation “matchConnectionEnd”, there is one more function implemented for considering if the connectionend’s destination type name is not defined, then the moniker penalty will be increased to -3 as the destination penalty value according to the design. The code for this function lists as follow:

```
//Get the destination type name
var destSpecifiedTypeName=conInstanceEnd.destination.userSpecifiedTypeName;
//If destination type name not defined
if(conInstanceEnd.destination.getTypes().size()==0 and (destSpecifiedTypeName=null or
destSpecifiedTypeName='')){
    //Use penalty of destination mismatch
    conEndMatch=destMismatched;
}
//If destination type name defined
else{
    conEndMatch=mismatched;
}
```

Operation “matchConnectionConnectee” is similar to that of operation “matchConnectionType” except in this operation, if a mismatch is found, a score of -3 is returned, in accordance with the “penalties” in Design Chapter. In destinations calculation, sub/super type relationships are considered and if the type name defined in destination can form a sub/super type relationship with the potential connection type’s destination, a score of 0.5 is given to consider this relationship, like what has been discussed in the Design Chapter. Operation “containsInSubTypes” and “containsInSuperTypes” are responsible for detecting sub/super type relation between two destinations. The code of examining sub/super type relationship is listed as follow:

```
//If the type name fialed to match the connection type's destintion, then
subtype and supertype relation is examined
if(destinationMatch=mismatched){
    //If it is sub type
    if(self.containsInSubTypes(instanceDefinedDestType ,
                                conTypeEnd.destination)){
        msg='';
        destinationMatch=0.5;
    }
    //If it is super type
    }else if(self.containsInSuperTypes(instanceDefinedDestType ,
                                        conTypeEnd.destination)){
        msg='';
        destinationMatch=0.5;
    }
}
```

For the purposes of creating warning and error indicators, instead of calling operation “match” and reorder the list, another operation called “filterConCandidatesforMatchingConTypes” and it will be the one to manage and to sort the list and return the sorted Sequence for use. The code of “filterConCandidatesforMatchingConTypes” is listed as follow:

```

operation Connection filterConCandidatesforMatchingConTypes(conInstance : Connection ,
                                                             connectionCandidates :
                                                             Sequence):Sequence{

    var result=Sequence();

    for (c : Connection in connectionCandidates) {
        result.add(self.match(conInstance , c));
    }

    return result.sortBy(s | s->first()).invert();
}

```

The weight calculation functions of instance matching type and type matching instance are performed in separate operations. The reason for this is that the two kinds of matching operations are different, hence mixing the code together is not a good idea; furthermore, the design of type matching instance are different. The hierarchy of type matching instance weight calculation is:

Type weighted sum= typeMatch

= typeMatchConnectionType+ typeMatchConnectionEnds

= typeMatchConnectionType+ (two typeMatchConnectionEnd+ two  
typeMatchConnectionConnectee)

#### 4.2.3 Connection Conformance Critique Development

The design of connection conformance is to take all connections in one level up into account. Hence in the critique of connection conformance, the first step is to get all the connections in the direct one level above of the current level.

Before entering validation body, it is needed to check that the current level is not the topmost level and, the connection being validated does not have been assigned a type already. This part of prerequisite checking is done in “guard” section inside a critique, the code is simple:

```
critique connectionConformance {  
  guard: self.levelindex-1>-1 and self.getTypes().size()==0
```

The meta-model of Melanee reveals how Melanee supports multi-level modeling and how each clabject is organized in each level. From PLM.ecore, it can be seen that each connection or entity is stored inside a level context, and the levels are stored inside a deep model context. Hence in order to get all the connections in one level above from the current level, it is necessary to navigate back the whole model and select only the level of the current level above. Inside the level all the connections in that level will be acquired and these connections are the potential connection types needs to be calculated and reordered. The code of getting potential connections listed as follow:

```
var currentLevelIndex=self.levelindex;  
var upperLevel=self.getLevel().getDeepModel().getLevelAtIndex(currentLevelIndex);  
var connectionCandidates=upperLevel.getConnections().select(c | c.potency > -2 and  
c.potency<>0)->asSequence();  
  
var potentialTypes:Sequence=self.filterConCandidatesforMatchingConTypes(self ,  
connectionCandidates);
```

First the level index of current level needs to be acquired, and the level index is stored in each clabjects in Melanee. EVL resembles OCL language to navigate back to the deep model and get the target level, which will contain all the connections as expected. Connections with potency 0 is not acquired for they are considered as instances and cannot be type for any connection instances in lower levels.

The next step is to calculate matching score and sort the list according to the matching scores. This is done by calling operation “filterConCandidatesforMatchingConTypes” like explained earlier. The “filterConCandidatesforMatchingConTypes” will return a list which

contains data structure of a sequence of {weighted sum, potential connection type, messages}. According to the design, the results need to be processed into three categories: best matches, possible matches and type matching instances. The results returned by “filterConCandidatesforMatchingConTypes” contains the first two categories, and the results returned by “filterConTypeCandidatesforMatchingConTypes” contains the third category.

Operation “getConnectionConformancePerfectMatchingMessage” takes the Sequence returned by “filterConCandidatesforMatchingConTypes” as input. It is responsible for extracting the best matching types and their messages. Regarding to how to identify best match, simply by examining the value of weighted sum is not accurate. The reason is that according to the design, to uniquely identifying a connection, it is not necessary to use all the five components. If a connection with weighted sum of 5 out of 6 (matching score of 6 means all component type names matched perfectly), it could still be considered the best match. Hence, to identify more accurately, contents of the message is examined instead in operation “getConnectionConformancePerfectMatchingMessage”. If a message is “”, which is blank, this means there will needs no modification for the connection instance to match the connection type. This kind of connection are considered best matching even if the corresponding weight is not 6.

The key code of operation “getConnectionConformancePerfectMatchingMessage” is listed as follow:

```
for(s : Sequence in msg){
    if(s.at(2)!=""){
        if(i==0){
            result='Connection \''+self.name+'\, matched connection \''+ s.at(1).name +'\';
        }else{
            result=result + '\nConnection \''+self.name+'\, matched connection \''+
            s.at(1).name +'\';
        }
        arrays.add(s);
        i=i+1;
    }
}
```



Similarly, to extract possible matches, in “getConnectionConformanceMessage” the process is to examine the contents of messages to see if it is not “. If there are contents which means that the instance needs modification to match such connection type. Hence such connection type will not be best matches to this instance. The key code is like:

```
for(s : Sequence in msg){
    if(s.at(2)<>""){
        if(i=0){
            result='Connection \''+self.name+'\', to match connection \'' + s.at(1).name + '\'' +
            s.at(2);
        }else{
            result=result + "\nConnection \''+self.name+'\', to match connection \'' +
            s.at(1).name + '\'' + s.at(2);
        }
        arrays.add(s);
        i=i+1;
    }
}
```

In operation “connectionConformance” the messages are linked together and the best matching, possible matching and type matching connection type lists will be reordered and the list will be presented to modelers to choose from in a Second-Stage Dialog.

#### 4.2.4 Second-Stage Dialog

In EVL it is possible to call Java native methods, the code to call Java methods in EVL is:

```
var typeSelectionDialog = new
    Native("org.melanee.core.models.plm.validation.tools.ConnectionConfor
    manceTypeSelectorCaller");

typeSelectionDialog.setResultsForFiltering(bestMatchTypesForSelection ,
    secondBestMatchTypesForSelection ,
    typeBestMatchTypesForSelection , self);
```

In order to let the Java class to be able to be called in EVL, the methods need to be registered as an extension in plugin.xml.

According to the design, the best matching types will be listed in front the of list, and all possible matching are listed from highest possible match to the lowest and be placed after best matches in the list. The third category of results will be listed in the rear section of the list. After user specified which connection to be the type, a type-instance relation will be created and the attributes of instance or the type will be reconfigured accordingly.

The Java methods are placed in newly created folder  
 org.melanee.core.models.plm.validation.tools in plugin  
 org.melanee.core.models.plm.validation. There are three separate classes:  
 ConnectionConformanceTypeSelectorCaller,  
 ConnectionConformanceTypeSelectorDialog and TypeSettingTool.

Class “ConnectionConformanceTypeSelectorCaller” is what is called inside quick-fix section. It is the method for interacting with EVL script and managing the dialog. It is responsible for accepting the three categories of results, call a dialog and allow modelers selected an entry, then set the type accordingly by calling class “TypeSettingTool”. The code to open a dialog and get its result is:

```
ConnectionConformanceTypeSelectorDialog selectionTool = new
    ConnectionConformanceTypeSelectorDialog(
        activeShell, bestMatchCandidates,
        secondBestMatchCandidates,
        typeConfigureMatchCandidates, theInstance);

int openResult = selectionTool.openDialog();

if (openResult == Window.OK) {
    result = selectionTool.getResult();
} else if (openResult == Window.CANCEL) {
    result = "";
} else {
    throw new SelectionDialogOpenException(ExceptionMsg);
}
```

Since the Sequences are passed into class “ConnectionConformanceTypeSelectorCaller” from EVL script, they are unprocessed and contain modification messages that is

irreverent to the dialog. Hence the three kinds of sequence need to be processed before passing them as parameters into the dialog.

After modelers selected a type in the dialog, the dialog will return a string, which is the name of the connection, according to the prefix of the string it is possible to determine which category and which connection modelers selected. Class “TypeSettingTool” will be called to create type-relation relationship and reconfigure the connection instance or the type accordingly. The code for setting type after dialog is closed is:

```
//Examine if it is a best match that modeler chooses
if (result.startsWith("Best")) {
    theType = getIndex(bestMatchCandidates,
        result.substring(result.indexOf("@"),result.indexOf("@")+9));

    typeSettingTool = new TypeSettingTool(theType, theInstance);
    typeSettingTool.setConnectionType();
    //Examine if it is a possible match that modeler chooses
} else if (result.startsWith("Change")) {
    theType = getIndex(secondBestMatchCandidates,
        result.substring(result.indexOf("@"),result.indexOf("@")+9));
    typeSettingTool = new TypeSettingTool(theType, theInstance);
    typeSettingTool.setConnectionType();
    //Examine if it is the third category match that modeler chooses
} else if (result.startsWith("Reconfigure")) {
    theType = getIndex(typeConfigureMatchCandidates,
        result.substring(result.indexOf("@"),result.indexOf("@")+9));
    typeSettingTool = new TypeSettingTool();
    typeSettingTool.reconfigureConnectionType(theType, theInstance);
}
```

Class “ConnectionConformanceTypeSelectorDialog” is the class for displaying a dialog with a list of results to choose from. Class “ConnectionConformanceTypeSelectorDialog” encapsulated a class called “ElementListSelectionDialog”, which is a dialog provided by Eclipse plugin framework. The dialog can show a list and buttons for customization. In our case the possible connection types as a list in the dialog and two buttons for confirm and cancel are presented. Class “ElementListSelectionDialog” accept an array of String as input, hence the three categories of results need to be converted to String[] so that the dialog can display them.

The method “getDisplayOptions” in class “ConnectionConformanceTypeSelectorDialog” will process the results. For presenting the list in the dialog clearly, at the beginning of each option there will be prefixes to indicate the category name, like “Best match”, “Change to” (this is the category that holds possible matches) and “Reconfigure” (this is the category that holds type matching instance category).

The dialog cannot be dismissed unless user press “Cancel” or “OK”, this can prevent unexpected exception. After the dialog is dismissed by modelers clicking “OK”, a result will be returned to class “ConnectionConformanceTypeSelectorCaller” to call class “TypeSettingTool” to establish a type-instance relationship.

Class “TypeSettingTool” is a utility class for setting types for connections and entities. The type-instance relationship is maintained inside the “eContainer” of the level that stores the type. The creation of a Classification in Java code is like:

```
//Obtain the eContainer that stores classification
Level container = (Level) type.eContainer();
Classification i = PLMFactoryImpl.eINSTANCE.createClassification();
//Set the type and instance inside a classification
i.setType(type);
i.setInstance(instance);
container.getContent().add(i);

//Set the potency of the instance accordingly
setConnectionEnds(type, instance);
if ((instance.getPotency() == -1 && type.getPotency() != -1) || (instance.getPotency() != -1
    && type.getPotency() != -1 && instance.getPotency() >=
    type.getPotency()))
    instance.setPotency(type.getPotency() - 1);
instance.setUserSpecifiedTypeName(type.getName());
```

However, defining connection type involves more than just setting type for connections, connectionend type should also be set for connectionends cannot be created and removed independently without connection.

During the setting of connectionends, it is unable to determine the “direction” of each connectionend as explained in the earlier chapter. Destinations are useful but if the types of destinations are not determined or the userspecifiedTypeName is not defined, destinations alone cannot be used as the unique type identifier for connectionend. Hence here I adopt the same mechanism as it is in the operation “matchConnectionEnds”. But there is one more possibility that needs to be considered, which is if the weight of the two sets of combinations are identical, there is a chance of setting the wrong type to the connectionends. To resolve this matter, another examination added to the if-else section, which is if the weighted sum of two sets of combination are identical, then use destination matching to better set the right type for the connectionends. The key code of setting connectionend is listed as:

```
//Swap and compare the weighted sum
if ((LL + RR) > (LR + RL)) {
    setConEndType(typeEndLeft, instanceEndLeft);
    setConEndType(typeEndRight, instanceEndRight);

} else if ((LL + RR) < (LR + RL)) {
    setConEndType(typeEndRight, instanceEndLeft);
    setConEndType(typeEndLeft, instanceEndRight);

} else {

    ...

//if the weighted sums of swapping are identical, then use destination to determine
connectionend type more precisely
if (desLeftName.equals(desTypeLeftName) || desRightName.equals(desTypeRightName)) {
    setConEndType(typeEndLeft, instanceEndLeft);
    setConEndType(typeEndRight, instanceEndRight);

} else {
    setConEndType(typeEndRight, instanceEndLeft);
    setConEndType(typeEndLeft, instanceEndRight);
}

}
```

When defining type in class “TypeSettingTool”, the potency value of the instance will be set if necessary. This means if the instance’s potency is larger than that of its type, then the instance’s potency will be reconfigured according to the type; if the instance’s potency is smaller than the type, then the potency will not be reconfigured. The reason for this is that the potency may be decreased by two or even more according to the needs of

modelers, so if the instance's potency complies with the rule of smaller than its type, then it is preserved, but modelers can always change the value of the potency in the properties sheet section.

#### 4.2.5 Other Validations

To prevent further errors after resolving types for connection instances, two more validations are implemented. The first one is critique "connectionEndTypeChecking". This is to prevent the possibility of connectionend's type is not set when the connection's type has been defined, which may cause inconsistency in the model. The second one is critique "connectionDestinationConsistencyChecking". This validation is checking destinations' types after the connection type has been set. In the "guard" section, the level of the connection must not be the topmost level, and the connection must also have a type defined. The process of the check is to first get the destinations and their respective types, and check against the destinations from the connection's type. If the destinations' types are not matched with the connection type's, then the validation will return false and an error message will be presented to modelers, together with a quick-fix to solve this error. The key of this validation is that when resolving type for destinations, only the ones which hasn't been set the type will be fixed, if there is already a type for the destination, it will not be processed and set a type. This is because change an entity's type should not be implemented in connection related context, it is further development in entity section.

#### 4.3 Entity Conformance

Connections are not the only clabjects that could allow user specify type name in Melanee, after adding "userSpecifiedTypeName" in clabject in meta-model, entities are also entitles the ability to let user specify type name and check its potential type if a type has not been defined. If an entity which the type of it hasn't been set, critique "entityTypeBestMatch" will validate the entity, check against all the potential types in one level up, and provide a

quick-fix in which a second-stage dialog can be prompted to allow modelers to choose and define type. The mechanism is similar to that of connection conformance.

Inside the critique, first in the “guard” section, check the current level and examine if the corresponding entity has a type or not. If the entity has a type or it is in the topmost level, then the critique will not be performed. The next step in the “check” section, get all the entities in one level up, the way to get all the entities is similar to getting all the connections: navigate to the deep model, get the upper one level and extract all the entities of those whose potency is not 0:

```
var currentLevelIndex=self.levelindex;  
var upperLevel=self.getLevel().getDeepModel().getLevelAtIndex(currentLevelIndex);  
var entityCandidates=upperLevel.getEntities().select(e | e.potency > -2 and  
e.potency<>0)->asSequence();
```

The type name modelers specified are used to identify the potential type for the entity. Hence if there is a match between userSpecifiedTypeName and entity’s name, the entity will be considered as the best matching type for the entity instance. However, there could be no match or wrong match for the value defined in userSpecifiedTypeName may be wrong. According to the design, the best matches will be placed in the front of the list if applicable, the possible matches, which is the other entities in the upper level, will be placed after the best matches to allow modelers to choose from. Hence all the potential entities must be divided into two categories: best match and potential match. Operation “getEntityMatch” and operation “getEntityTypeCandidates” are defined to categorize the entities. Operation “getEntityMatch” uses the type name to check against entities’ names, if a match is found, the entity will be placed into a sequence:

```
for(s : Entity in potentialTypes){  
    if(s.name=self.userSpecifiedTypeName){  
        result.add(s);  
    }  
}
```

“getEntityTypeCandidates” will find the entities that their names are different from userSpecifiedTypeName, and place them in a sequence for returning:

```
for(s : Entity in potentialTypes){  
    if(s.name<>self.userSpecifiedTypeName){  
        result.add(s);  
    }  
}
```

The second stage dialog is constructed in Java class as well, and in critique Java native method is called by registering the Java class as an extension in plugin.xml. There are two more classes added: class “EntityTypeSelectorCaller” and class “EntityTypeSelectorDialog”. Class “EntityTypeSelectorCaller” receive the two categories of entities and process them and pass them to class “EntityTypeSelectorDialog”. After “EntityTypeSelectorDialog” is dismissed, according to modeler’s selection, class “TypeSettingTool” is called to set the type accordingly. Class “EntityTypeSelectorDialog” is a class that encapsulated class “ElementListSelectionDialog”. It accepts String[] as input.

In class “TypeSettingTool”, a new method is added called “setEntityType”, the type setting of entities is similar to that of connections, hence I will not elaborate the details in the thesis: create a classification, store it in the corresponding type level’s eContainer and set the potency value if necessary.



## 5 Evaluation

All the features developed and discussed in Implementation Chapter need proper testing and evaluation before it is ready for academic and practical use. And the features implemented by extending Melanee need proper evaluations to examine the correctness of the implementation and detect possible deficiencies in the design. In this chapter, all the functions implemented in Melanee will be tested by adopting models from existing papers and by conducting some serials of experiments to analyze the result. Every part of the implementation will be examined through model to test if the feature works the way as expected, and to see if the design of each function meet the requirements or not.

There are three categories of implementation in this thesis: deep-connection implementation, connection conformance implementation and entity conformance.

### 5.1 Deep-Connection Implementation Evaluation

Originally Melanee supports deep connections to certain degree, which means there could be type-instance relationship between connection type and connection instance, and each connection there is potency value for deep instantiation implementation. However, according to what was proposed in [7], the following feature of deep-connection needs to be supported by Melanee:

1. Each connectionend associated with connection must have a moniker with potency, and the instance should inherit features from the connectionend type and display type 's name after the moniker;
2. Besides the first one, deep multiplicity should be implemented and should support arbitrary number of multiplicity according. And new connections should have multiplicity with boundary from 0 to \* with potency 1.

### 3. Validations related to multiplicity and connection to level conformance.

To test if the above requirements are met, a model from [7] is implemented in Melanee. This model has deep-connections and number of deep-connection related features which is the best for testing the implemented deep-connection related features. The model set is shown in Fig. 41.

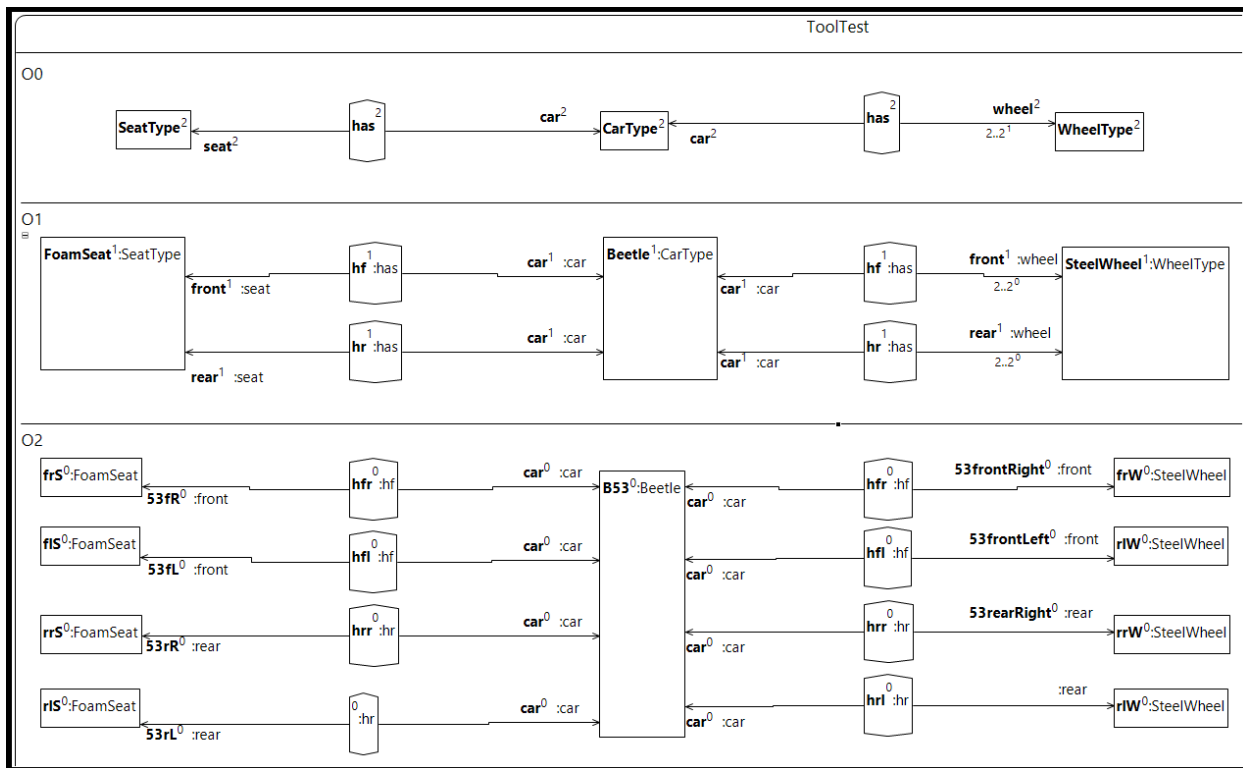


Fig. 41 Model for testing deep-connection taken from [7].

The model set depicted in Fig. 41 has 3 levels. In the topmost level, there are three entities: SeatType, CarType and WheelType. Each entity has potency value two. The two connections have the same name of “has”, but they are connected to different destinations. The connection “has” connected to SeatType and CaType has monikers “seat” and “car”, and the potency of the whole connection is 2. The other connection “has”

that connects to CarType and WheelType has the monikers of “car” and “wheel” with potency 2. There is a multiplicity with lower 2, upper 2 and potency 1 defined in this connection. The connections and entities in “O1” level are the instances of the according entities or connections in “O0” level, and clabjects in “O2” are instances of the clabjects in “O1”. In this model, clabjects in “O1” level have the feature of being instances and types, and they are the types for clabjects in “O2”. The type of each clabject is shown in the instance with the colon in front of the type name. For instance, entity “FoamSeat :SeatType” in “O1” indicates entity “SeatType” in “O0” is its type; connection “hf :has” means connection “has” in “O0” is its type. One type could have many instances.

From the model depicted in Fig.41, it can be clearly seen that the model constructed comply with the rules of deep-model for Melanee has no complaints about this model. Each moniker can show the type name and a potency value on the shoulder of each moniker. The multiplicity is shown in the corresponding connectionend, and is inherited by the instance of the connectionend with potency decreased by 1.

### 5.1.1 Moniker and Potency Rendering

When creating connection instance of connection “has” which connects to “SeatType” and “CarType”, the potency values of both the monikers are inherited from connectionend, which means the potency of monikers are decreased by 1 according to those of the type. In our example, potency 1 is the result of calculation of potency 2 in “seat”. And, the type name is defined according to the type’s name of the connectionend instance. And if a type’s name is changed, the type name after the colon of the corresponding instances will be refreshed as well. Up-to-date information of the connectionends will be shown each time a value changed in all connectionends. Moniker and potency are rendered in the format as expected in Design Chapter.

### 5.1.2 Deep Multiplicity Support

The multiplicities in connectionend support dynamic number of multiplicity definition. In our case, there is one multiplicity, but if adding more multiplicities, each multiplicity will be rendered separately with “|”, which is like what is shown in Fig. 42. The potency is shown in the right shoulder position and both potency and the bounds are auto refreshable if any value in multipliers is changed.

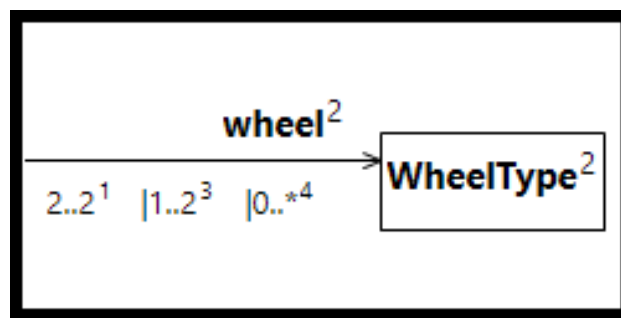


Fig.42 Dynamic number of multiplicities support.

### 5.1.3 Validations Related to Deep-Connection

There are several validations developed under the connectionend context to check inconsistencies or errors. The first one is potency value of each moniker should not be defined as negative value (except -1). However, in Melanee -1 is considered as representing infinite number. Hence in the validation the potency value should not be smaller than -2, otherwise it is an error. For instance, by changing potency of connectionend “seat” to -3, then an error sign will be shown in the correspond connectionend and error message will be shown if hovering the mouse over the error sign, which contains message regarding what is wrong and how to rectify it, like it is shown in Fig. 43.

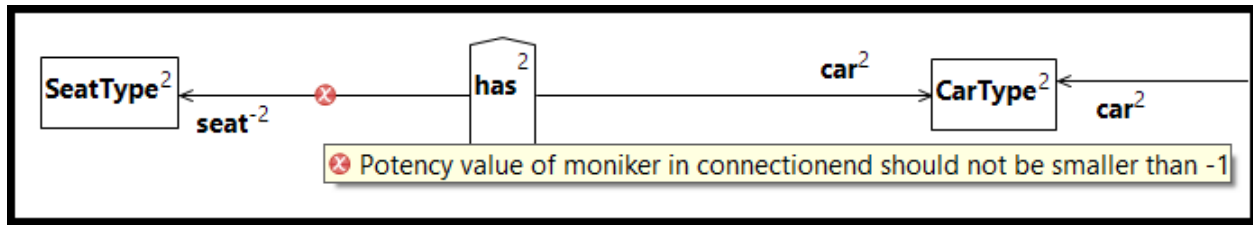


Fig. 43 Error sign and error message for defining potency as negative value.

The potency of each multiplicity is validated as well. If any of the multiplicities has a potency of smaller than -2, then an error sign and an error message will be placed in the corresponding connectionend graph.

Also, the lower and upper bound values of each multiplicity need to be validated. Not only should they be larger than -2, but also they need to follow the rule of lower value should be smaller or equal to that of upper. In the following tests an error sing will be placed in the corresponding connectionend with an error message, the tests are:

1. Change lower value to be smaller than -1;
2. Change upper value to be smaller than -1;
3. Change lower value to be larger than that of upper;
4. Change two multiplicities' potency values to the same.

The last test is to make sure that the deep multiplicities have no contradictions. If the multiplicities have the same potency value but different boundary limits, then it is impossible for the tool to know which multiplicity is the correct one to follow.

Deep multiplicities are not implemented to shown numbers in the model only. Deep multiplicities are defined to validate against the number of instances there are between the upper and lower boundaries to many levels down. The design of deep multiplicities is

to inherited the multiplicities and validate against only one level below for convenience in implementation and achieving high efficiency.

Each multiplicity in the connectionend is inherited by its instances. For example, in Fig.41, connection “hf” and connection “hr” that connect to “Beetle” and “SteelWheel” inherited the multiplicity of 2...2 with potency 1. Melanee does not complain for the multiplicity validation for the number of connection instances are 2, which corresponds with the boundary defined in the multiplicity.

To test the multiplicity validation, it is necessary to modify boundary numbers and check if the validation can still pass after boundary values are changed. The first is to change the lower boundary to a lower value to 1 in Fig. 41. By doing so the boundary becomes [1,2] instead of [2,2], and the number of instances are 2, which does not violate the multiplicity defined.

The next step is to adjust the upper bound. Like in our case increase the upper bound to 5 is just increase the boundary of [2,2] to [2.5], and the number of instances is 2, which does not violate the multiplicity defined.

Many other tests are conducted to let the number of instances fall between the boundaries defined in multiplicity. And if the number of instances is between the boundary, then no error sign or error message will be reported by Melanee.

Then I change the value of both lower and upper bound to another value so that the number of instances will not match the boundary. For instance, the boundary of multiplicity in Fig.41 are changed to between 3 and 5, and there are only two connection

instances in one level down. Then an error sign and an error messages will be placed in the corresponding connectionend, like what is shown in Fig. 44.

The model has no complaints after construction and the features of each components of deep-connection are tested and passed. However, there are two problems that needs to be resolved: the first one is that the potency of each moniker is derived from its type by decreasing it by 1, but the potency is configurable after it is inherited. This leaves a problem which is that the potency of connectionend instances could be modified to a value larger than that of its type. This will create an inconsistency and no error will be reported for now; the second problem is that the multiplicities are inherited, and only the one with potency 1 is checked against the number of instances. This also creates a problem: even though all the multiplicities are inherited and could be validated against the number of instances eventually, but the multiplicity validation may be incorrect by inheritance: the number of instances of the connection may fail to match the deep-multiplicities defined for deriving multiplicity from its type in connectionend can cause problem. For example, there are one another multiplicity defined in the same connectionend which is [4,4] with potency 2 in in Fig 41. This multiplicity is directly inherited by its instances in one level down, like it is shown in Fig. 45. Connection “hr” and connection “hf” that connect to “Beetle” and “SteelWheel” are the instances of connection “has” that has multiplicities. The two connection instances inherited the multiplicity and as the potency becomes 1, they will be validated to check if the number of their instances match this criterion. To be valid, each connection instance needs to have 4 instances of their own, which is 8 instances in total. But according to the definition, in “O0” level, there should be 4 instances in “O2” level.

The first problem can be solved easily be adding another validation regarding to the potency checking against the connectioned’s type under connectionend context, the validation code lists as follow:

```
constraint potencyViolationChecking {  
guard: self.type<>null
```

```

check {
    var typePotency= self.type.potency;

    if(self.potency=-1 and typePotency>0){
        return false;
    }else if(self.potency>0 and typePotency>0 and self.potency>=typePotency){
        return false;
    }
    return true;
}

message: 'potency of \''+self.moniker+'\' should be smaller than \''+self.type.potency+'\' according
to its type'
}

```

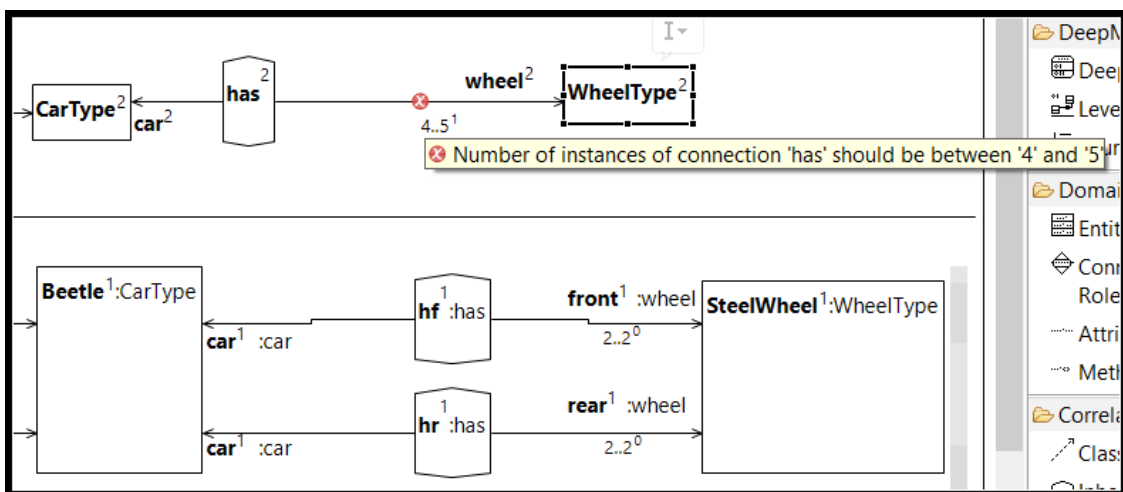


Fig. 44 Multiplicity violation error sign and message.

From the example presented above it is clear that by multiplicities inheritance there could be a problem in validation.

Considering the possibility of potency being -1 to represent “\*”, in “potencyViolationChecking”, only the mismatch circumstances are checked in if-else statement. If a mismatch between type and instance happened, an error message and an error will be shown in the connectionend in the model, like it is shown in Fig. 46.



The other problem regarding to multiplicity checking, considering multiplicity inheritance and validation of multiplicity when the potency reached 1 can cause problem, there needs to be a new design in multiplicity validation to match what deep-multiplicities represents.

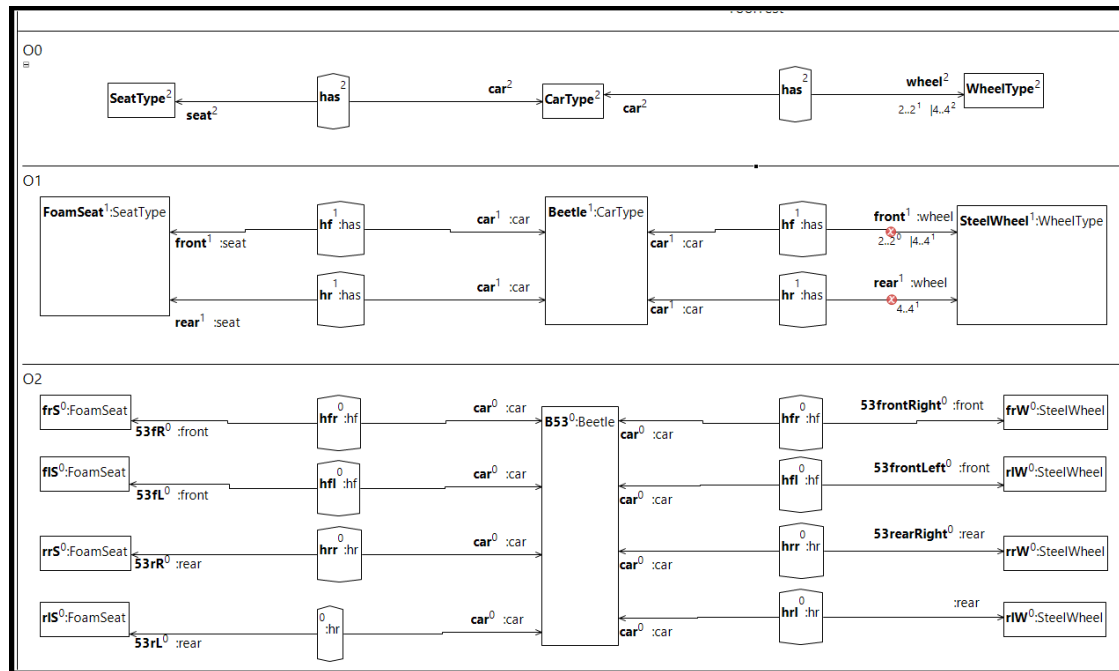


Fig. 45 Multiplicity mismatch in multiple level definition.

The “connectionendMultiplicityChecking” in connectionend context can only validate number of connection instances in one level down, but in order to validate correctly, it is necessary to validate all the multiplicities in the level as the potency value defined in deep-multiplicities, so that it has to go through many levels down to validate instead of only validate against immediate level below.

The first step is to delete all the code related to multiplicity inheritance in `ConnectionEndCreateCommand` and in related templates.

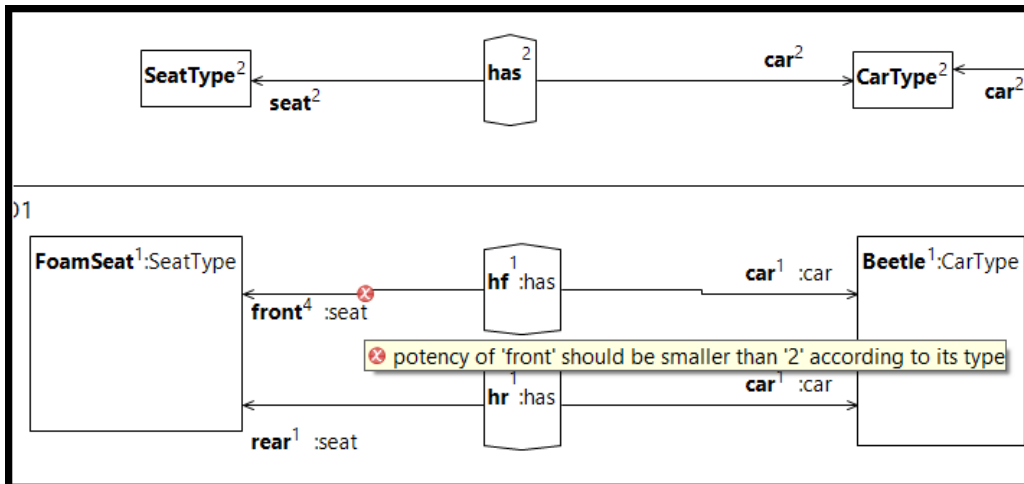


Fig. 46 Potency violation error sign and message in connectionend.

The second step is to modify “connectionendMultiplicityChecking” developed in earlier chapter to allow it to be able to validate against instances in many levels down. To achieve this goal, two functions, or called two operations in EOL, are required. The first operation is to get an entity’s instances in a specific level below. This is because the mechanism of multiplicity validation is to get the opposite destination’s instances in a specific level below and then get their connections for processing.

Operation “getInstancesAccordingtoPotency” is developed to achieve this goal, it requires two parameters: the opposite destination and the potency of the multiplicity. It needs the potency to control how many levels below to explore, and after gathering all the entity instances, it will return a sequence containing all the required instances of the destinations. The code lists as follow:

```
operation ConnectionEnd getInstancesAccordingtoPotency(des: Entity, potency:Integer):Sequence{
    var res:Sequence;
    res.add(des);

    while(potency>0){
        var temp:Sequence;
        for(e : Entity in res){
            for(en : Entity in e.getInstances()){
                temp.add(en);
            }
        }
    }
}
```

```

        }
        res.clear();
        res=temp;
        potency=potency-1;
    }
    return res;
}

```

In the while loop the potency will control in how many levels below the entities need to be obtained. Two for loops are developed, with one nested inside another. The outer for loop is for iterating all the entities of a level. Then in the inner for loop, the instances of each entity that are stored in “res” will be extracted and stored in a temporary sequence. Then all the entities will be stored inside “res”, and potency will decrease itself by one till it becomes 0. The reason “res” is cleared is that it only needs to store instances of a certain level, not all the instances in all levels below. The whole operation works in similar way to that of BFS Algorithm (Breadth First Search Algorithm).

Another operation is to get the connection’s type in certain levels up, this is because that each of the connection instances needs to be examined against the type so that only the connections that are the instances of the specific connection type will be acquired. Operation “getTypeAccordingtoPotency” accepts two inputs, which are potency of the multiplicity and the connection that needs to explore further up to get its type. The code of the operation is as follow:

```

operation ConnectionEnd getTypeAccordingtoPotency(con: Connection, potency:Integer):Connection{
    while(potency>0){
        con=con.getTypes().get(0);
        potency=potency-1;
    }
    return con;
}

```

The while loop controls how many levels up will be explored. Each time inside the loop the direct type of the connection will be navigated to and be stored so that after the loop is completed, the correct type in certain levels up can be returned.

In the constraint “connectionendMultiplicityChecking”, each of the defined multiplicities will be examined against the number of instances in certain levels down. Each of the multiplicities is iterated by a for loop. Inside the loop, the instances of the opposite destinations in the specific levels below are obtained, and the connections of all the instances will be checked and counted to see how many of them are the instances of the connection type, which is just the connection that the connectionend with the multiplicity connects to. The counted number will be examined to see if it is within the ranged defined in the multiplicity. The code lists as follow:

```
for(m : Multiplicity in self.multiplicity){
    if(m.potency<>0){
        var multiplicityLoIrValue=m.lower;
        var multiplicityUpperValue=m.upper;
        var oppositeClabjectInstances=self.getInstanceAccordingtoPotency(oppositeDes,m.potency);
        //Examine if it is infinite
        if (multiplicityUpperValue==1){
            if((oppositeClabjectInstances->forall(clabjectInstance |
            (multiplicityLoIrValue<=(clabjectInstance.getConnections().select(typeConnections |
            self.getTypeAccordingtoPotency(typeConnections,
            m.potency)=connectionType).size()))))=false){
                isValid=false;
                if(flag=0){
                    messages=messages+'Number of instances of connection
                    \' +self.connection.name+\' should be between \' +multiplicityLoIrValue+\'
                    and \' +m.potency+\' level down';
                    //Reset the flag to indicate it is not the first line
                    flag=1;
                }else
                    messages=messages+\'\\nNumber of instances of connection
                    \' +self.connection.name+\' should be between \' +multiplicityLoIrValue+\'
                    and \' +m.potency+\' level down';
            }
        }else{
            //Compare against lower and upper boundary
            if((oppositeClabjectInstances->forall(clabjectInstance |
            (multiplicityLoIrValue<=(clabjectInstance.getConnections().select(typeCo
            nnections | self.getTypeAccordingtoPotency(typeConnections,
            m.potency)=connectionType).size()))
            and
            ((clabjectInstance.getConnections().select(typeConnections |
            self.getTypeAccordingtoPotency(typeConnections,
            m.potency)=connectionType).size())<=multiplicityUpperValue)))=false){
                isValid=false;
                //Construct messages to modelers
                if(flag=0){
                    messages=messages+'Number of instances of
                    connection \' +self.connection.name+\' should be
                    between \' +multiplicityLoIrValue+\' and
                    \' +multiplicityUpperValue+\' in \' +m.potency+\' level
                    down';
                }
            }
        }
    }
}
```

```

        flag=1;
    }else
        messages=messages+"\nNumber of instances of
        connection '"+self.connection.name+"' should be
        between '"+multiplicityLoIrValue+"' and
        '"+multiplicityUpperValue+"' in '"+m.potency+"' level
        down';
    }
}
}
}

```

The message that will be presented to modelers are constructed by the help of a variable called flag, which is an integer number. It is for determine if the message that will be stored inside “messages” is the first line of message, if it is not, then “\n” will be added in front of the new message to make it in a human readable format.

#### 5.1.4 Evaluation of Moniker Potency Validation and New Multiplicity Validation

The evaluation of potency value in moniker have several possible scenarios:

1. Potency of type is -1, and instance potency is any number, which passed the tests;
2. Potency of type is larger than 0, and the potency of instance is -1, which creates an error and error message;
3. Potency of type is larger than 0, and the potency of instance is larger than that of the type, which creates an error and error message;

As a conclusion, the moniker potency validation fits the requirements of deep-connection features.

The multiplicity validation, which requires a model of more than 2 levels. Here I adapt the model depicted in Fig. 45 but remove the inherited multiplicities. The two multiplicities indicate there should be 2 instances in 1 level below and 4 instances in 2 level below, which fits what it depicts in Fig. 45, and no complain from Melanee received.

Then the multiplicity with potency is adjusted to let the number of instances in 2 levels below smaller than lower boundary of the multiplicity. Then an error sign showed and an error message occurred, like shown in Fig.47.

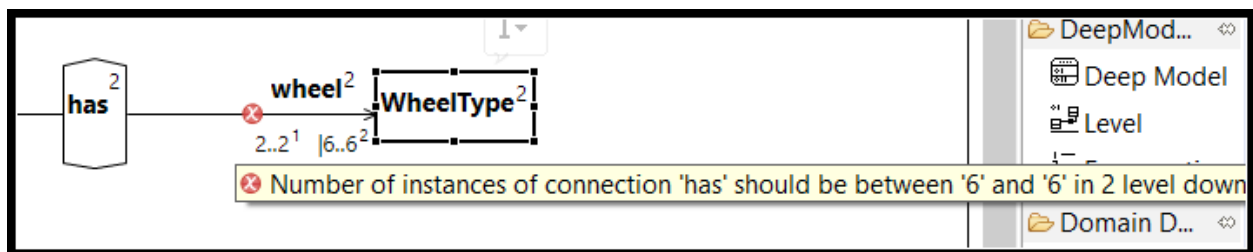


Fig. 47 Instances number smaller than lower bound.

After this the lower and upper bound are adjusted so that the instance number is larger than the upper boundary. Again, an error sing and error message will be shown in the corresponding connectionend, which means the model failed to pass the multiplicities validation, but the validation passed the test here.

The next test is to adjust both the multiplicities so that in 1 or 2 levels below, the instance number will fail to match, then two violation messages will be prompted at the error sign, indicating that the instances in the next 2 levels below all failed to comply with the multiplicities defined in “O0” level. Other tests have been conducted, and the newly developed multiplicity validation passed these tests.

In the evaluation of deep-connection implementation, there are some problems detected but resolved at last. And the final implementation passed all the test after the two problems are corrected.

## 5.2 Connection Conformance Evaluation

Connection conformance aims to provide functionalities that can guarantee the conformance between connection instances and their matching types to allow Melanee support “explanatory” modeling. According to the design there are a few requirements that needs to be met in the implementation, which are:

1. Each of the connections in all levels except the ones in the topmost level are validated to see if a type has been defined for them or not;
2. If the type has been set, then abort the validation functions and no warning sign will be presented; if the type has not been defined, then all the connections in one level above will be obtained for calculation, and user specified type name will be shown in the connection instance;
3. Using the equation defined in Design Chapter to calculate each possible matching types' matching score, and get three categories of results. The results will be reordered according to the matching score, with highest possible matches in the front and lower possible ones in the rear;
4. Present the list to modelers, and with each matching type there are suggestions about how to change the connection instance to match a connection type. The suggestion will show which component does not match and how to modify it;
5. In the quick-fix section, a second-stage dialog with the list calculated and reordered will be shown so that modelers can select a type in the list to set the type. By

setting the type, the components except destinations that are not matched will be set according to the selected type.

To evaluate the extended Melanee of connection conformance features, two sets of models are adopted from papers for testing. One model is the “Online Store” model from the [1] papers and the other one is the “Car parts” model from [7].

### 5.2.1 userSpecifiedTypeName

Tests are conducted to examine if a clabject will show type name according to the value defined in “userSpecifiedTypeName”. When the type is not defined in a clabject, it shows type name according to the value of userSpecifiedTypeName; if the type has been defined for a clabject, then it shows type name according to the type’s name instead of according to userSpecifiedTypeName. When values in userSpecifiedTypeName is changed, the type name in the graph will refresh accordingly to render the most up-to-date information.

### 5.2.2 Weight Calculation Functions Evaluation

The first model set that is used to evaluate the connection conformance is the “Online Store” model. The details of the model are shown in Fig. 48. As depicted in the figure, there are three levels and each level contains several number of entities and connections. All the entities and connections haven’t been assigned a type yet.

As it is shown in the figure, each connection and entity have a small warning sign on the upper right corner of each shape except the ones in the topmost level. This complies with the design requirements of not validating against the clabjects in the topmost level, which only have type facet.



In the topmost level, there are three entities: “ProductType”, “ComputerModel” and “MonitorModel”. “MonitorModel” and “ComputerModel” are subtypes of “ProductType”. There is only one connection “has” connects the two entities “MonitorModel” and “ComputerModel”. Hence in one level below this level, all the connections only have one possible candidate which is connection “has”.

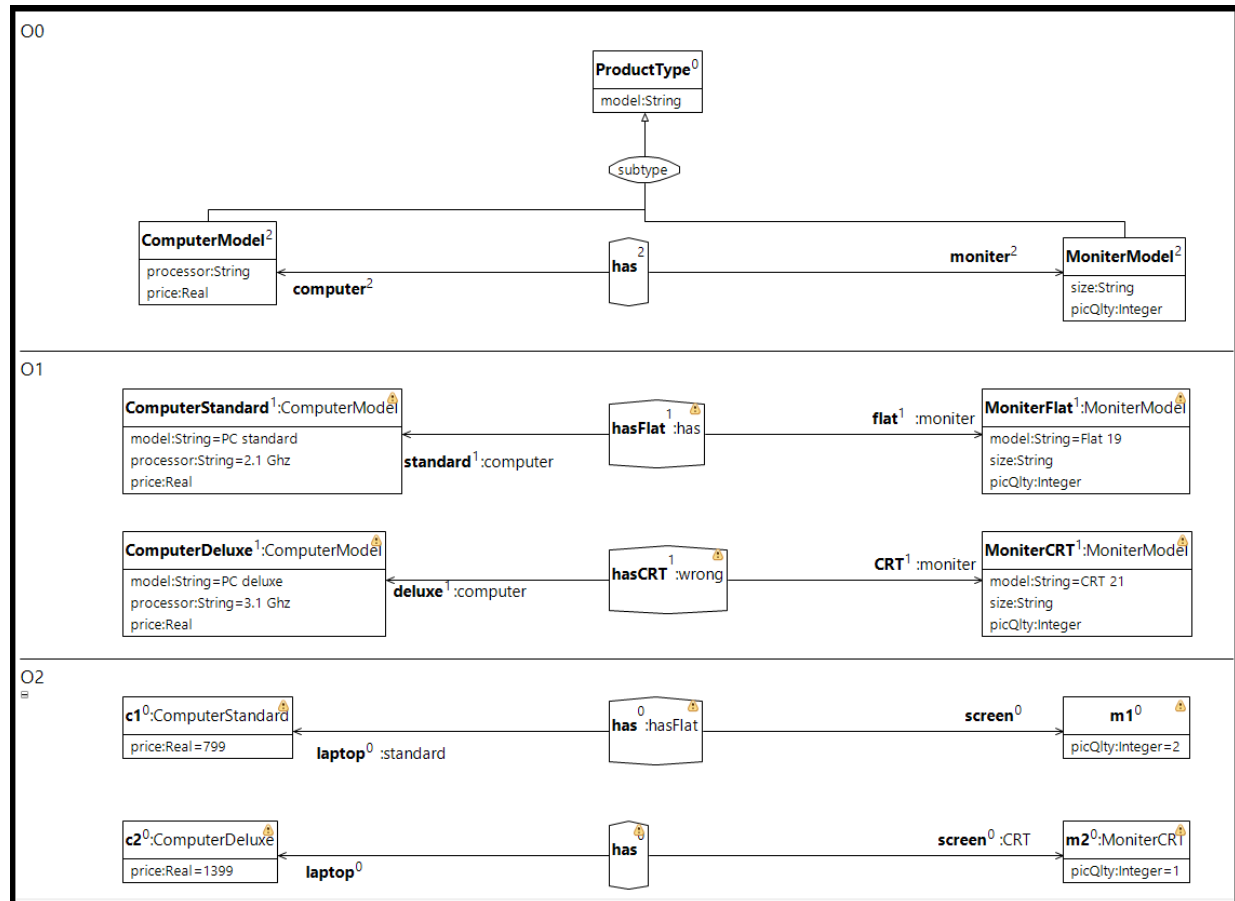


Fig. 48 Online Store model taken from [1].

In the next level below, the first connection “hasFlat” with moniker “standard” and “flat” matched connection “has” perfectly. The two destinations of “hasFlat” declared “MonitorModel” and “ComputerModel” as their types respectively, the type name of the two moniker matched with moniker names of those of connection “has” in Level “O0”, and the connection’s type name has been specified as “has”. Judging by all the given

components type names, connection “has” is the perfect matching type for connection instance “hasFlat”. In the list, only one connection “has” is listed, and it is a best matching type. The information of the list is shown in Fig. 49, which is “Connection ‘hasFlat’, matched connection ‘has’ ”.

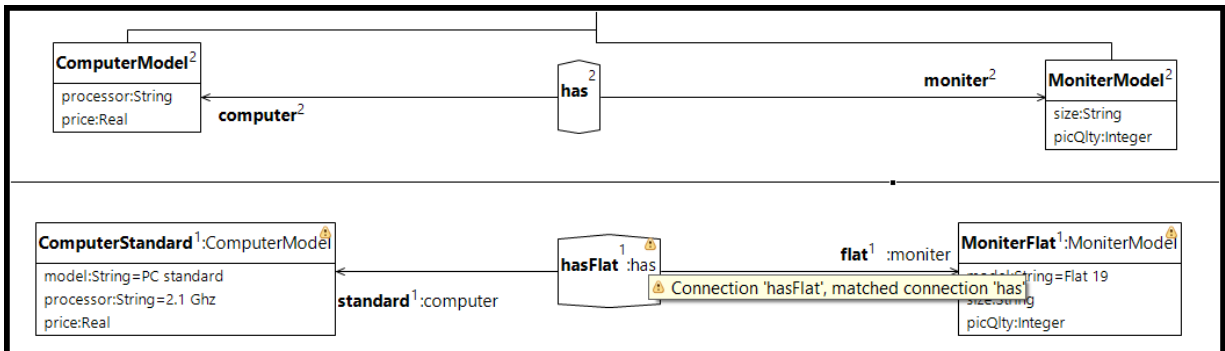


Fig. 49 Best connection matching list and message.

For connection “hasCRT”, the destinations are defined with proper type names and they matched the destinations connected to connection “has” in Level “O0”. But the connection type name is defined as “wrong”, which failed to match connection “has” in Level “O0”. Hence in the list even though there is only one possible matching type, it is a possible match for one of the component’s type name is wrongly defined. Hence the message should be “to match connection ‘has’, change type name in connection ‘hasCRT’ to ‘has’ ”, which is exactly what is shown in Fig. 50. And considering one component type name mismatched, the third category of result which is “type matching instance” us found and shown in Melanee, like depicted in Fig. 50.

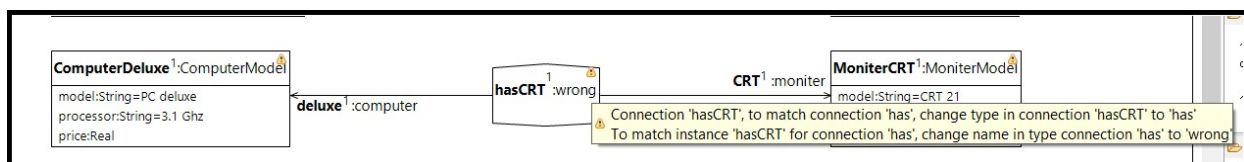


Fig. 50 Recommendation message is shown when partial match found.

For the two connection instances in the lowest level, both named as “has”, for them there are two connection types in one level above, but only one of them can be considered as the best matching type for each of the two connection instances. For connection “has” that connections to “c1” and “m1”, four components match with connection “hasFlat”, the components are “c1”, “standard”, “laptop” and “has”. Again, not all components need to be provided to identify a connection, and what is provide in the connection instance “has” is enough to identify “hasFlat” as its type. However, the other connection “has” with moniker “CRT” and “deluxe” is not considered as the type. This is because the connection type names failed to matched, and with two destinations failed to match, the matching score of connection “hasCRT” as the type for connection “has” will be smaller than 0. The overall weight is smaller than 0 which means that this connection will be considered as a incompatible match for connection instance “has”. The scenario is shown in Fig 51.

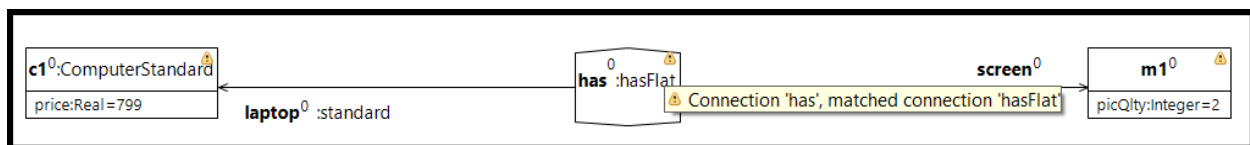


Fig. 51. Only one connection is considered as type for connection instance “has”.

The other connection in the lowest level is also named as “has”, which connects to “c2” and “m2”, and for this connection instance, even though the connection type name and a moniker type name have not been specified, the type names in other components are sufficient to indicate that the connection type should be the one connecting to “ComputerDeluxe” and “MoniterCRT”. In the example, only one connection matched this definition which is connection “hasCRT”. Connection “hasFlat” is not considered as a matching type for connection “has”, again because two destinations failed to match.

Connection conformance functions passed all the tests conducted using “Online Store” model. But for this model, the number of entities and connections are limited, and the extended functions should meet the requirements of filtering potential connection types,

reorder them and provide messages which related to modification in a more complicated scenario. In order to further test the functionalities and examine the correctness of connection conformance design, another model was adopted, which is “Car Parts” model from [7], and to better test the functions in the following tests, all the entities and connections are constructed without defining types in the whole mode. The model is shown in Fig. 52.

There are three levels, and in the topmost level there are two connections, both named “has” and they share one same destination which is “carType”. In the next level below, entities “FoamSeat”, “Beetle” and “SteelWheel” are the instances of entities “seatType”, “carType” and “wheelType” respectively.

The connection “hf” that connects to “FoamSeat” and “Beetle”, type name in all the five components are defined, and all the userSpecifiedTypeName matched with connection “has” that links to “carType” and “seatType” in Level “O0”. Thus, connection the “has” is the perfect matching type for connection “hf”. According to the design, the best match types will be listed in the very front of the list. For the other connection type “has” that connects to “wheelType” instead of “seatType”, the moniker type name “:car” matched one of the moniker in “has”, and the corresponding connectionend connects to “Beetle”, which is the instance of “carType”. However, the other moniker type name “:seat” failed to match the moniker “wheel”, and the destination type name in “FoamSeat” failed to match with “wheelType”. One destination failed to match, but according to the design principle, one destination match could also be considered as a possible match. Hence the overall matching score calculation is: one destination, one moniker and connection name matched will add 4 to over all weighted sum, but one moniker and one destination type name failed to match, penalties of -4 will be added to overall weighted sum. As the result, the matching score of the other connection “has” is 0. Hence connection “has” that connects to “wheelType” will be considered as a lowest possible match, but it is listed after the best matching one in the list with messages regarding to how to modify the type names to match connection “has” that connects to “wheelType”.

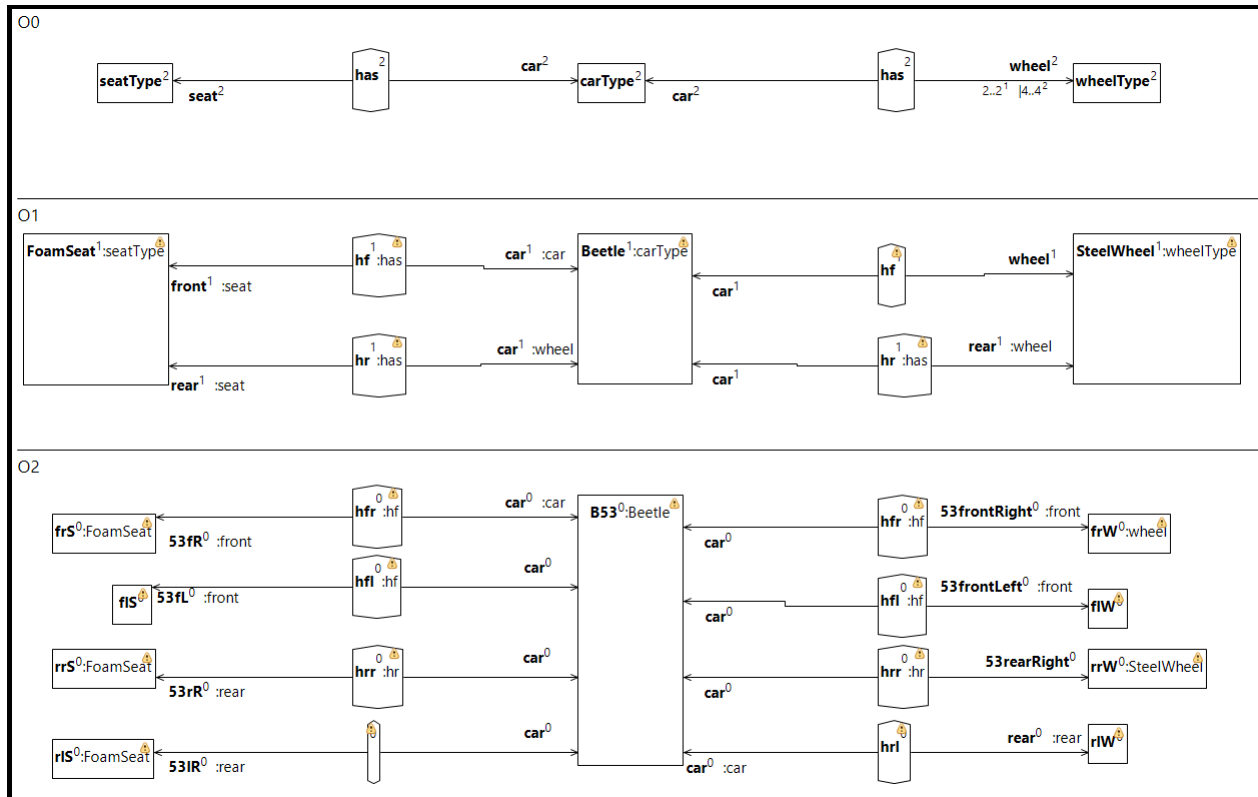


Fig. 52 Car parts model set taken from [7] with types for clabjects undefined.

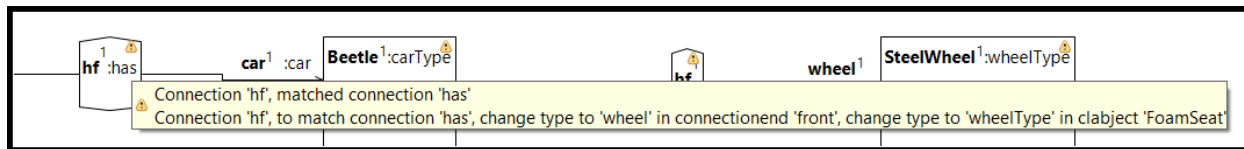


Fig. 53 One best match and one possible match found for connection “hf”.

The third category of matching result, which is let the type matching instance, and for connection “hf” there is no third kind of matching result. The reason is that connection “has” that connections to “seatType” matched connection “hf” perfectly, there is no need to reconfigure type “has” to match instance “hf”, and for the other connection “has”, one destination and one moniker type name failed to match, which violates the rules discussed in design, only small “mistakes” will be considered in the third type of result, which means at most two monikers can be wrong, or one connection name or one destination

mismatch is acceptable to produce a third category of result. In this case, one destination mismatch and one moniker mismatch, hence there is no third type of result.

Melanee showed exactly as expected in the warning sign of connection “hf”, for there are two mismatching components, the messages shown for connection “hf” contains messages about how to modify these two component type names to match the connection type. The message is shown in Fig. 53.

For connection “hr” that connects to “FoamSeat” in “O1” level, again all the type names are defined in all the components, but one of the moniker type name is wrongly specified as “:wheel”, hence the connection type “has” that connects to “seatType” can no longer be considered as a 100% matching type, but it is still considered as a possible matching type for all the other components matched connection “has” perfectly. The matching between “hr” and the other connection “has” that connects to “wheelType” is similar to the matching between connection instance “hf” and type “has”, one destination and type name matched, two monikers and one destination failed to match, so according to weight calculation, the weight of the other connection “has” is -1, which means it is considered an incompatible match, which will not be shown in the list.

There is one possible matching type for connection “hr”, but there is no perfection matching types, which means the possible matching type will be listed in the very front of the list. There is also one third category matching type for connection “hr”, which is connection “has” that connects to “FoamSeat” can be reconfigured to match connection instance “hr”, for only one moniker type name mismatched, which fits the requirements of being a third category of result. Hence in the overall result, there will be one possible matching type in front of the list, and one type matching instance result in the rear of the list. Melanee show the result just as expected, which is shown in Fig. 54.

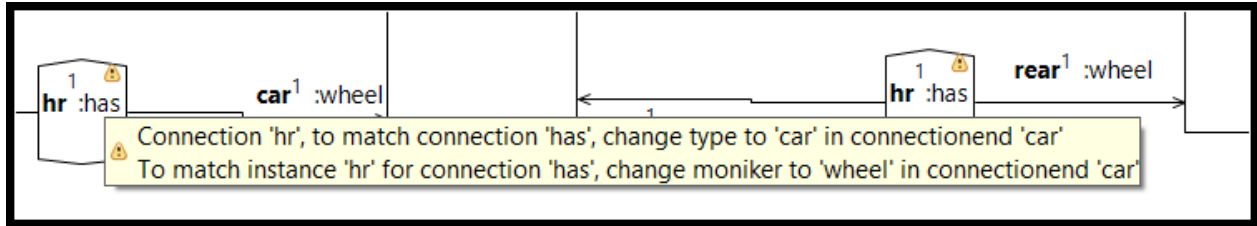


Fig. 54 Potential types listed for connection “hr”.

It is possible that sometimes the type names for connection and monikers are not specified, in such case destinations' type names, as well as connection name and moniker names, will be utilized to identify a connection. For example, the connection “hf” that connects to “SteelWheel” instead of “FoamSeat” in Fig. 52. The type names for all the components except the destinations' type names are unspecified, hence penalty of missing type name in connection instance will be added. In this case, connection “has” that connects to “wheelType” will be a perfect matching type for connection “hf”, for the type names in destinations match the destinations' names in “has”, and monikers are matched with connection “has” that connects to “WheelType”, but the connection name failed to match, hence the overall matching score is 3.5 after calculation considering a penalty of -0.5 is given for connection name missing, monikers are not given this penalties for according to the design, moniker names in the connection instance matched with monikers in the connection “has”. But the other connection “has” that connects to “seatType” is not considered as a possible match at all, for one destination match and one destination mismatch, two moniker type names missing, the weighted sum will be -2 according to the design. Melanee works the same way as expected and showed the list for connection “hr”, with connection “has” that connects to “wheelType” as a perfect matching and the other connection “has” as an incompatible match which is not even presented to modelers.

In “O2” level, there are more entities and connections than there are in “O1” level, and in “O1” there are 4 connections that connects to one same destination “Beetle” that could

be considered as potential connection types, which is more complicated. Some of the type name information are left blank intentionally to test all possible scenarios.

The connection “hfr” that connects to “frS”, it has type names defined in all the five components. All the type names are defined to match connection “hf” that connects to “FoamSeat”. Hence “hf” will be considered as a perfect matching type for “hfr”. For connection “hr” between “FoamSeat” and “Beetle”, the destinations’ type names and a moniker type name defined in connection instance “hfr” matched with those of connection “hr”. However, there are two components failed to match: connection type name and one moniker type name. The overall calculated weighted sum will be 1 for penalties of moniker mismatch and connection mismatch are added, which means connection “hr” will be considered as a possible match for connection instance “hfr”.

For the other two connection types between “Beetle” and “SteelWheel”, even though one connection destination failed to match the type name defined in “hfr”, the other one destination “Beetle” matched, so they can still be considered as possible match. The type name “:Beetle”, “:car” and “:hf” matched with connection “hf”, but destination type name “:FoamSeat” and moniker type name “:front” failed to match, which means -4 will be added to the weighted sum, hence the overall weight is 0. The connection “hr” that connects to “SteelWheel” is considered as an incompatible match, this is because the overall matching score is -3 after calculation, for one destination, one moniker and one connection type name mismatched, hence it will not be listed in the list.

According to the discussion, the list for connection will be: besting matching type “hf” that links to “FoamSeat”, possible matching type “hr” that links to “FoamSeat”, and lower possible matching type “hf” that links to “SteelWheel”, which is exactly what Melanee presented after calculation.



Another scenario is possible which is that modelers prefer to use monikers instead of using connection name to identify connections. For instance, the connection in bottom left corner of Fig. 52. It has no connection name nor connection type name, and one moniker type name is not specified for moniker “car”. However, destination type names defined, and, even the moniker “car” has not been specified a type, the moniker itself will be used to compare against the type’s moniker. In our case here, moniker type name “car” matched with moniker “car” of connection “hr”. Hence, it can identify connection “hr” between “FoamSeat” and “Beetle”, for all the 4 components matched perfectly. Connection “hf” between “FoamSeat” and “Beetle” is considered as a possible match, for one moniker type name “:rear” failed to match with “front”. Hence, even though some component’s type names are missing, it is still possible to identify the best matching connection type with monikers, like what is shown in Fig. 55.

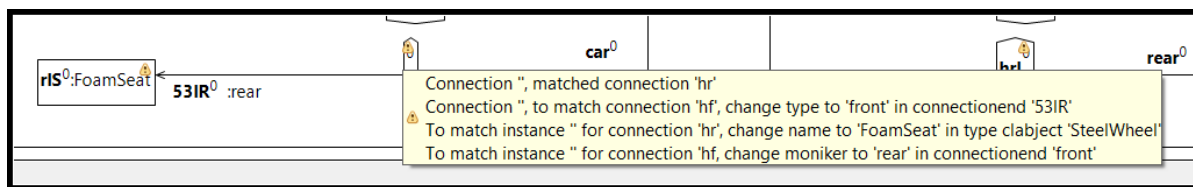


Fig. 55 Matching connection type list for connection “hfr”.

The other two connections that connect to “SteelWheel” are considered incompatible matches. Connection “hf” and connection “hr” are not considered as possible match for the overall weighted sum are -2.5 and -0.5 respectively, which are smaller than 0. Besides these two categories of result, there are two third category of matching results for the connection instance between “rIS” and “B53”, the type matching instance. There are four matches will be listed in total, divided into three categories, perfect match, possible matching type and type matching instance. For this connection instance, there are many matching types listed for many of the component type names are missing, which creates the possibility of matching many types.

Melanee showed what was expected, and the detail of the messages is shown in Fig. 55.

If not enough components' type names are defined inside a connection, there is a possibility of finding more than one best matching type for it. This is because certain components' type names are missing, only a slight penalty of -0.5 will be added to the sum, the matching score will not be greatly reduced for the missing type names in the connection instances. For instance, connection instance "hrl" that connects to "rlW" can find two best matches which are the two connection types "hr" in Level "O1".

The connection "hfl" between "B53" and "flW" in "O2" level, the type name for destination "flW" is undefined, which can result in the possibility of being the instance of entity "FoamSeat" or "SteelWheel". But according to the design, with the absence of destination type name, moniker type name will be used with increased penalty to rule out incompatible matches. Moniker type name ":front" matched with the moniker of connection "hf" that connects to "FoamSeat" in "O1", and the other connection "hf" is considered as a possible match for the moniker type name ":front" failed to match. The two connections "hr" are all considered a mismatch here, for ":front" failed to match with their monikers, and the overall matching scores are all -2.5 for moniker penalty is increased for penalty for moniker "53frontLeft:front" is increased, which means they are incompatible matches.

Melanee showed as expected, the information is like what is shown in Fig. 56 (a).

For connection "hfl", if moniker penalty is not increased when destination type name is missing, all the four connections will be listed in the list, like shown in Fig. 56 (b). From 56 (a) and 56 (b), it can be seen that increasing penalty can successfully rule out incompatible matches to present a more concise and correct matching list.

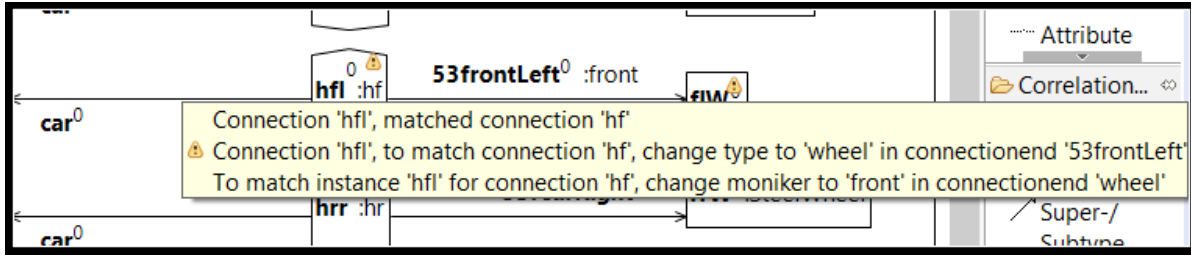


Fig. 56 (a) Four matches found for connection “hfl”.

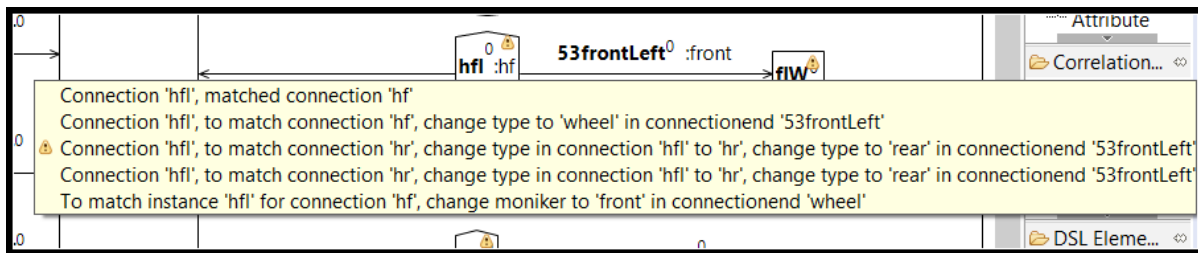


Fig. 56 (b) Three more matches found for connection “hfl” if destination penalty is not adopted by moniker.

Other connections in both model sets are tested by modifying type names to create different scenarios, and all worked as expected as well, no errors or problems detected. Considering in the Design Chapter, sub/super type relation of destinations are considered, to evaluate this function, another two scenarios are created to test against this function.

One scenario, which is depicted in Fig. 57, there are two possible connection types both named “has”, and destination “RearSeatType” is the sub type of destination “SeatType”. If there is not consideration of sub types, given the type names defined in connection instance “has”, both the connection types will be viewed as possible matches and connection “has” that connects to “SeatType” will be listed in front of the other connection “has”. But with “RearSeatType” being the sub type of “SeatType”, connection “has” that connects to “RearSeatType” will be listed in front of connection “has” that connects to “SeatType” for the matching score is 5.5, which is higher than 4.

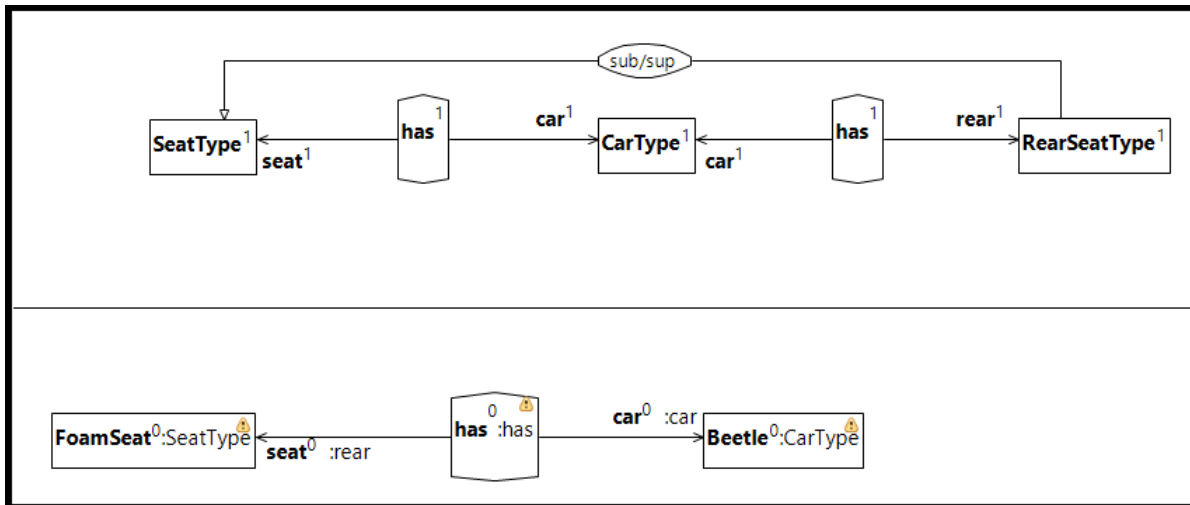


Fig. 57 Entity “RearSeatType” is the subtype of Entity “SeatType”.

Another scenario is what is depicted in Fig. 58. “SeatType” is the super type of “FrontSeatType”, and the type names defined in connection instance matched with connection “has” that connects to “SeatType” except one destination type defined to match with “FoamSeatType”. With sub/super type relationship being considered in Melanee, it shows connection “has” that connects to “SeatType” as more possible matching type and listed in front of the connection “has” that connects to “FoamSeatType”.

To test the designed functions thoroughly, and test against all possible scenarios to validate the weight calculation design, more comprehensive experiments were conducted to fully test the functions, which are explained in the rest of 5.2.2.

As `userSpecifiedTypeName` in each component can have three possible states: right name that can match, wrong value or missing value (molders choose not to specify type name for a component in the connection instance or the component name in the type). The combination of these possible states of each component in TS needs to be tested to further examine the design. Considering in the design only one destination mismatch is allowed and two destinations failed to match will be considered as incompatible matches,

and Melaniee mainly supports binary connections, three components on the same side are considered using combinations of the three states in the first round of tests. A total of 27 different scenarios (combination of one moniker, one destination and connection name) are created to examine if the connection instance can find matching type under different circumstances by change type names in the connection instance. The using of combinations of three components is because that they are typical considering a binary connection is symmetric. In this thesis, a table of results is presented and some interesting scenarios are explained and discussed.

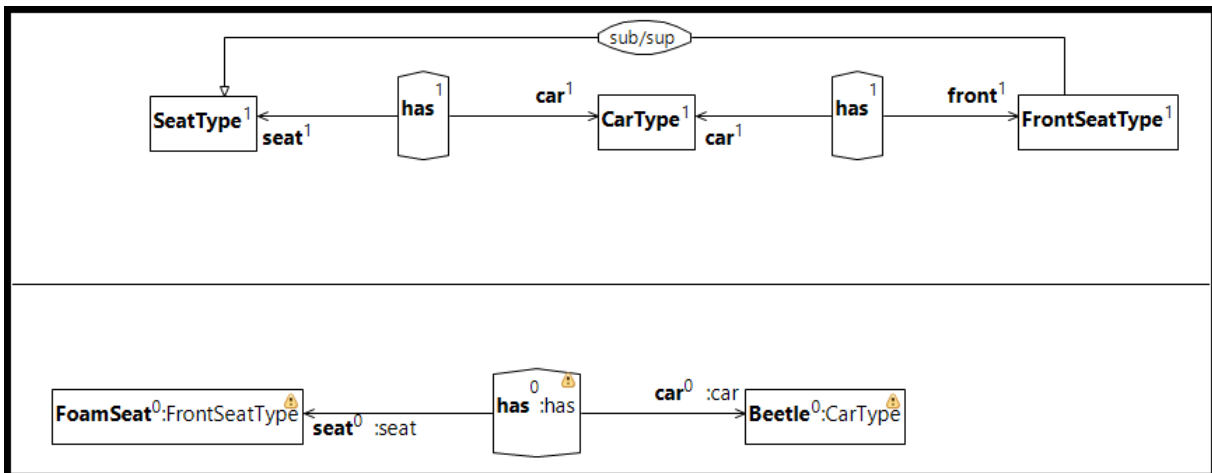


Fig. 58 Entity “FrontSeatType” is the subtype of Entity “SeatType”.

The matching scores of all 27 cases are listed in Table 2, compatible matching type score range is from 0 (0 is included) to 100, and in compatible matching type score range is from 0 (0 is not included) to -100. In the tests, only one destination, one moniker on the same side and the connection name is considered, the rest of the components are intentionally ignored.

Number	Destination	Moniker	Connection Name	Matching Score %
01	Right	Right	Right	100
02	Right	Right	Missing	37.5
03	Right	Right	Wrong	25
04	Right	Missing	Right	62.5
05	Right	Missing	Missing	0
06	Right	Missing	Wrong	-10
07	Right	Wrong	Right	50
08	Right	Wrong	Missing	-10
09	Right	Wrong	Wrong	-20
10	Missing	Right	Right	62.5
11	Missing	Right	Missing	0
12	Missing	Right	Wrong	-10
13	Missing	Missing	Right	25
14	Missing	Missing	Missing	-30
15	Missing	Missing	Wrong	-40
16	Missing	Wrong	Right	-30
17	Missing	Wrong	Missing	-80
18	Missing	Wrong	Wrong	-90
19	Wrong	Right	Right	0
20	Wrong	Right	Missing	-50
21	Wrong	Right	Wrong	-60
22	Wrong	Missing	Right	-30
23	Wrong	Missing	Missing	-80
24	Wrong	Missing	Wrong	-90
25	Wrong	Wrong	Right	-40
26	Wrong	Wrong	Missing	-90
27	Wrong	Wrong	Wrong	-100

Table 2 Matching scores for 27 different scenarios.

From the Table 2 it can be seen that when destination type name is right, it has the highest average matching score to find the type. This reflects the importance of destinations in finding matching types. Row 09 showed a mismatching score of -20 for destination type name right and both moniker and connection type names wrong. This is acceptable because even if destination matched, connection name and moniker name are all mismatched will result in the type as an incompatible one. If both moniker and connection name are missing, like what is shown in Row 05, the matching score is 0, which means the connection will be viewed as the lowest possible matching type.

One scenario in this category is shown in Fig. 59 (component type names in moniker “front” and destination “SteelWheel” are not considered in this test). The connection instance “doesHave” is defined with one moniker and connection type name missing, but the one destination type name is correct. In this scenario, the two missing component type names will add -1 to the overall weighted sum, so according to the design, connection “has” can be selected as a possible matching type for connection instance “doesHave”, and its overall weighted sum is 0 after calculation. The other connection “has” that connects to “TruckType” is an incompatible match for both the destinations failed to match.

When destination type name is missing, the average matching score is lower than that of when destination type name is right. Row 16, 17 and 18 showed the situation when moniker is used to rule out incompatible matches when destination type names are not given, hence the matching score of moniker wrong and destination name missing are very low. Another incompatible matching type is shown in Row 15, which is a situation that is both destination and moniker are missing, and connection name wrong.

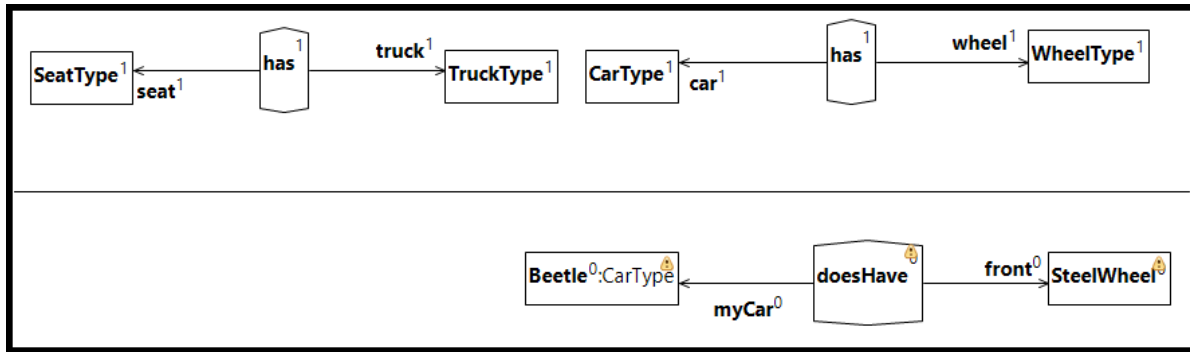


Fig. 59 Connection name and a moniker type names missing.

The scenario of without defining destination type name, but using moniker type name instead to find the most suitable matching type. This circumstance is shown in Fig. 60 (component type names in moniker “front” and destination “SteelWheel” are not considered in this test). The destination type name in “Beetle” is not defined, but the type name “:car” is specified in moniker “myCar”. The missing component will add penalty -0.5 to the overall weighted sum. In this case, the moniker type name is right so it is possible to determine that connection “has” that connects to “CarType” can be its type with a weighted sum of 2.5. Even if the destination type name and connection type name are missing, like what is shown in Fig. 61, it is still enough to identify “has” as its connection, but the overall weighted sum will be 0, which indicates that it is the lowest possible matching type.

When destination type name is wrong, the overall matching score is greatly reduced. When the three component type names are all wrong, it is considered a 100% incompatible match. The other scenarios have lower incompatible matching score except for Row 19, with one destination mismatch but the other two components matched, so it is still considered as a possible match with matching score of 0, which is considered the lowest possible match.



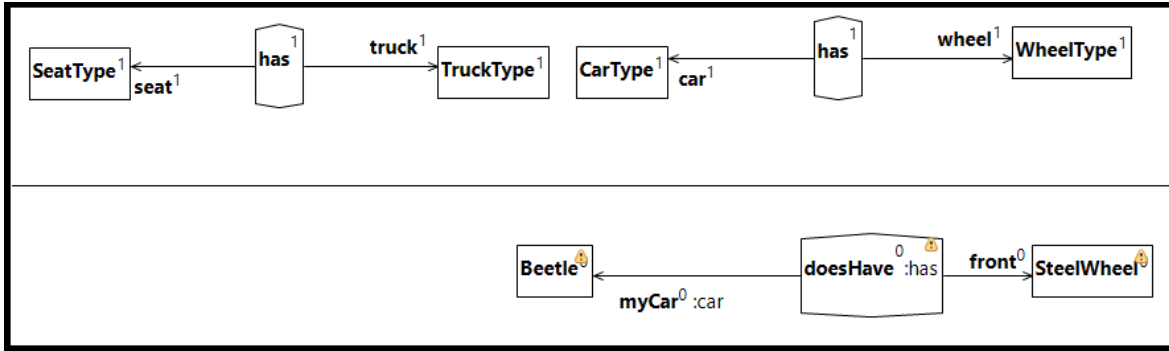


Fig. 60 Moniker is used to identify a connection with the destination type name missing.

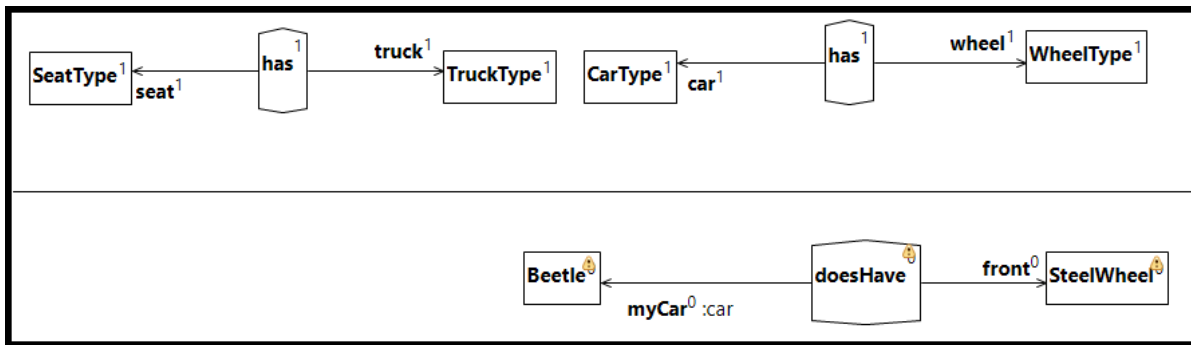


Fig. 61 Moniker, destination and connection type names missing.

Table 3 shows the reordering of all the combinations according to the matching score. The highest matching score is when all components are correct, is 100.

Number	Destination	Moniker	Connection Name	Matching Score %
01	Right	Right	Right	100
02	Right	Missing	Right	62.5
03	Missing	Right	Right	62.5
04	Right	Wrong	Right	50
05	Right	Right	Missing	37.5

06	Right	Right	Wrong	25
07	Missing	Missing	Right	25
08	Right	Missing	Missing	0
09	Missing	Right	Missing	0
10	Wrong	Right	Right	0
11	Right	Missing	Wrong	-10
12	Right	Wrong	Missing	-10
13	Missing	Right	Wrong	-10
14	Right	Wrong	Wrong	-20
15	Missing	Missing	Missing	-30
16	Missing	Wrong	Right	-30
17	Wrong	Missing	Right	-30
18	Missing	Missing	Wrong	-40
19	Wrong	Wrong	Right	-40
20	Wrong	Right	Missing	-50
21	Wrong	Right	Wrong	-60
22	Missing	Wrong	Missing	-80
23	Wrong	Missing	Missing	-80
24	Missing	Wrong	Wrong	-90
25	Wrong	Missing	Wrong	-90
26	Wrong	Wrong	Missing	-90
27	Wrong	Wrong	Wrong	-100

Table 3 Reordering of matching scores for 27 different scenarios.

Row 01 of matching score 100 indicates that it is a 100% best match. Then the score range from 62.5 to 37.5 (37.5 is not included), no component type name is wrong except Row 04, which is acceptable for monikers are given lower penalties when destination is correct. The matching scores below 37.5 till 0 (0 is included), Row 06 has lower matching score compared with Row 05 for missing component should have better score than wrong component. And even if destination and monikers are correct, wrong connection name

still will be considered a possible match, but a lower matching score than when connection name is missing will be produced. About the incompatible matches, which are the ones with negative matching scores, are mostly having wrong destination name or more than one component type names being wrong or missing. Row 16, 22 and 24 are situations of increasing moniker penalty when destination type name is missing. Row 14 has right destination type name but wrong moniker name and wrong connection name, which cannot be considered as a possible match.

Evaluation also been conducted considering two monikers and two destinations as well. In this evaluation, the monikers and the destinations on each side are considered to be right, missing or wrong at the same time. The matching scores are shown in Table 4. In Table 4, Row 19, with two destinations wrong and other components matched is considered as an incompatible match with matching score of -22.2, and when using one destination and one moniker only in Table 1, wrong, right, right of destination, moniker and connection get a matching score of 0, which is the matching score for lowest possible matches. This fits the design of considering one destination mismatch as a possible match but greatly reducing its matching score, but two destinations mismatch is considered an incompatible match despite the correctness of other components.

Number	Two Destination	Two Moniker	Connection Name	Matching Score %
01	Right	Right	Right	100
02	Right	Right	Missing	58.3
03	Right	Right	Wrong	50
04	Right	Missing	Right	50
05	Right	Missing	Missing	8.3
06	Right	Missing	Wrong	0
07	Right	Wrong	Right	33.3
08	Right	Wrong	Missing	-5.6

09	Right	Wrong	Wrong	-11.1
10	Missing	Right	Right	50
11	Missing	Right	Missing	8.3
12	Missing	Right	Wrong	0
13	Missing	Missing	Right	0
14	Missing	Missing	Missing	-27.8
15	Missing	Missing	Wrong	-33.3
16	Missing	Wrong	Right	-55.6
17	Missing	Wrong	Missing	-83.3
18	Missing	Wrong	Wrong	-88.9
19	Wrong	Right	Right	-22.2
20	Wrong	Right	Missing	-50
21	Wrong	Right	Wrong	-55.6
22	Wrong	Missing	Right	-55.6
23	Wrong	Missing	Missing	-83.3
24	Wrong	Missing	Wrong	-88.9
25	Wrong	Wrong	Right	-66.7
26	Wrong	Wrong	Missing	-94.4
27	Wrong	Wrong	Wrong	-100

Table 4 Matching scores for 27 different scenarios considering two monikers, two destinations to be right, wrong or missing at the same time.

Besides all these scenarios tested above, there are other possible situations such as connection type name is missing, like what is shown in Fig. 62 (component type names in moniker “front” and destination “WheelType” are not considered in this test). In this case, even though all the type names in destination, moniker and connection name are defined, the possible matching type has no name defined in these three components. According to the design, missing in the possible matching type will add no penalties to the overall weighted sum, hence in this case, the overall matching score is 0 (components type names in moniker “front” and destination “SteelWheel” are not considered in this

test), which means it will be considered as the lowest possible match for connection “has”. By comparing the results of this test and the tests conducted above, missing type name in connection instance and missing name in connection type are differentiated, like what is discussed in the Design Chapter.

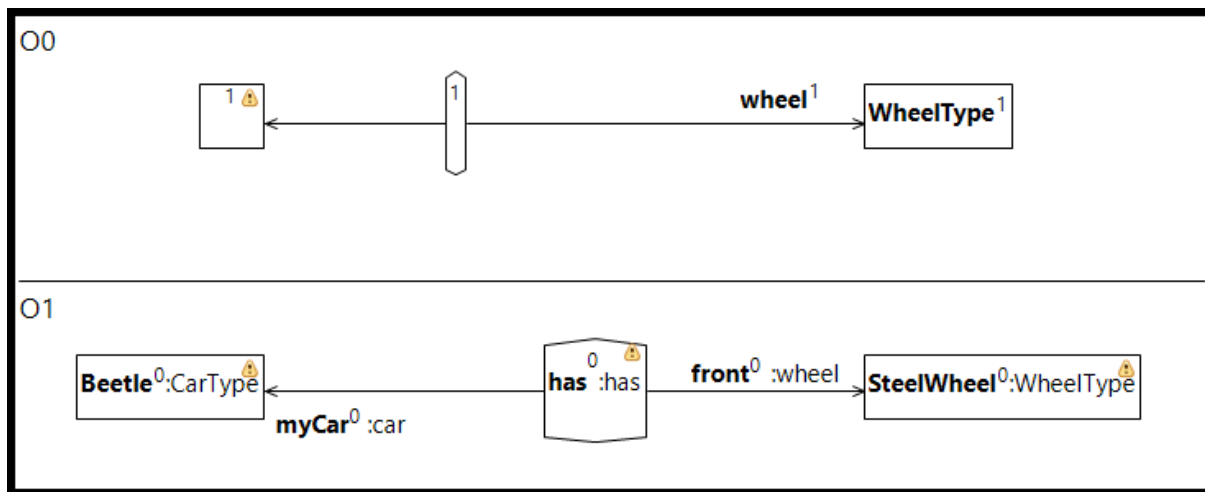


Fig. 62 Destination, moniker and connection name missing in possible matching type.

### 5.2.3 Quick-Fix for Connection Conformance

When modelers click on the quick fix, there is pre-defined a solution to solve the certain error or warning declared in advance in quick-fix section. The quick-fix for resolving connection type can prompt a second-stage dialog with all the possible types listed so that modelers can choose from it. The list in the dialog will be listed according to category, with best match in the very front, possible matches in the middle and type matching instance in the rear of the list. And the list is adjusted according to the results of calculation, there could be no best matching type category or all three categories of results are all presented, like what is shown in Fig 63 (a) and Fig 63 (b).

After deciding a type for the connection instance, the “userSpecifiedTypeName” will not be used to show type name, instead, the connection name in type connection will be rendered.

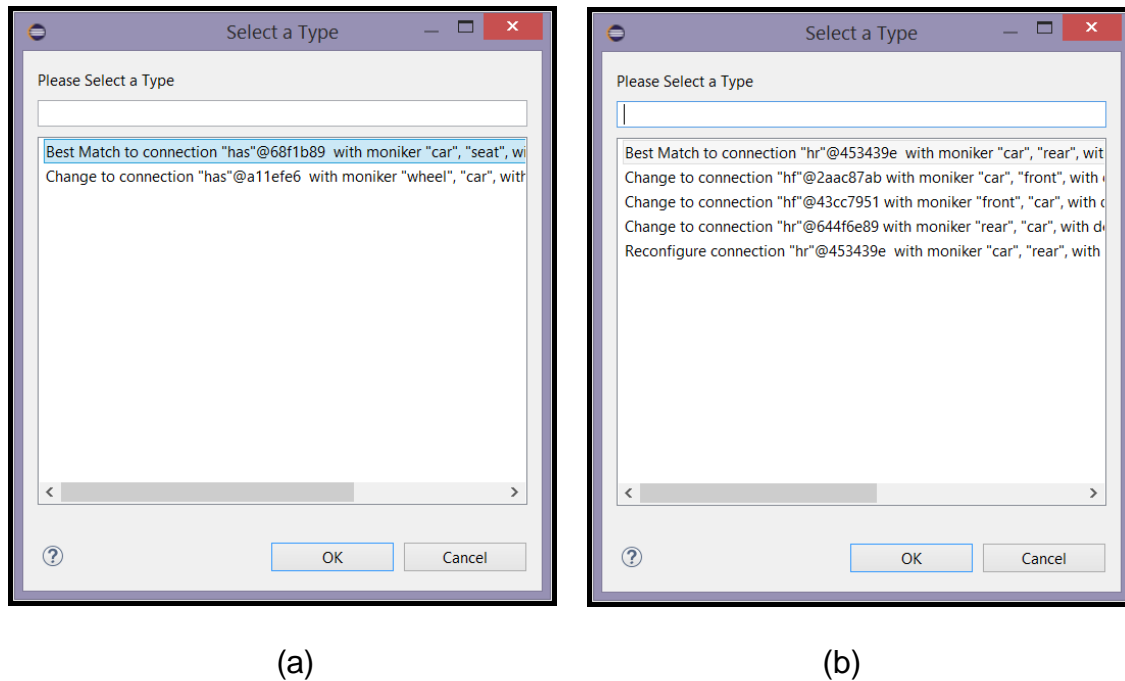


Fig. 63 Two second-stage dialogs for two different connections with different categories of results.

When modeler choose a type, the connection type and both connectionends' types will be set in Java native methods. After setting the type by creating classification relationship, the potency of each component will remain the same if the potency does not violate the rule of smaller than its type's. However, if the potency is larger than that of its type or it is equal to that of its type, then it will be reset by examining if the potency is -1 or not. If the potency values are -1, then the potency in the connection instance is retained, otherwise the potency in the instance will be reset as type's potency value decreased by 1. Melanee worked exactly as expected in the design.

#### 5.2.4 Other Validations of Connection Conformance

There are two other validations: examine the connectionend's type and destinations' type after connection type has been defined.

The first validation can examine situations of when a connection is set a type, but its connectioneds' types are still left undefined. This would create inconsistencies, and when it happened in tests, an error was shown in both connectionends to indicate the situation with messages.

The second validation is to examine destinations' types after connection type has been set. This is to ensure consistencies between connection instance and connection type. There is a quick-fix for this error as well, and the destinations during tests are corrected according to the type's destinations in the quick-fix accordingly.

#### 5.3 Entity Conformance

For each of the entities in the model, there is a validation to examine the entity's type. If the type has not been set, there will be a warning sign in upper right corner of the entity.

In testing scenarios, if the type name of the entities found matching one in the immediate level above, then in the second-stage dialog the best matches will be placed in front of the list to choose from, with all the rest of the possible matching in the rear of the list; if the best matchings cannot be found, then all the possible types will be listed in the second-stage dialog.

After choosing one as type, the entity's type name and potency will be set accordingly. The entities type issues are correctly resolved in the tests.

## 5.4 Summary

In the evaluation of connection conformance, 54 different combinations are tested to examine the correctness of the design and the implementation. The three tables in Chapter 5.2.3 are presented with test results which is satisfactory for the current design and can find the 100% matching ones if they exist, and possible matching types are listed according to their matching score from highest to lowest. When one or two components of connection type failed to match with the type name modeler specified, the third category of result will be presented to modelers which recommending how to reconfigure connection types to match the connection instance.

There are 54 combinations tested, but in total there are 5 components, and considering there are three possible states for each component, there are totally  $3 \times 3 \times 3 \times 3 \times 3$ , 243 different combinations. Due to limited time given for this thesis, there is no enough time to test all the 243 combinations. But the 54 tested in this chapter are typically scenarios and produced promising results. Other possible scenarios are created and tested in the model sets evaluations which are tested in the beginning of this chapter, by intentionally define some component type names to be missing or wrong. For example, the connection “hr” in Level “O1” in Model Set 02 with one moniker being wrong and the other moniker being right, and connection “hfl” in Level “O2”, which has one destination type name missing and the other being right, etc. The testing scenarios showed in Fig. 62 showed a situation of allowing component names in possible connection matching types to be missing and it is then calculated as a possible match for the connection instance to differentiate missing in component name in possible connection types and missing in component type name in connection instance. These tests, they all produced the correct results by listing the most matching types in front of the list, and in second stage dialog modelers can resolve the type issue conveniently and correctly.



## 6 Future Work

The main purpose of this thesis is to provide support for multi-level modeling by implementing deep-connection feature and an explanatory approaching of constructing models in Melanee. To this end, I extended the PLM and LML languages in Melanee to support explanatory modeling, and the deep-connection implemented can allow connectionends have monikers with potency value, and arbitrary number of deep-multiplicities with corresponding validations to ensure the integrity and correctness of the whole model.

Given the time limit of master study, there are few works that can be considered as future work. The future work related to deep-connection and Melanee includes the support of “n-ary” connections. Melanee mainly supports binary connections to this end, which means for each connection, two connectionends situations are considered in the code of Melanee. However, when modeling, a relationship could involve more than two destinations, which means a connection could have three or more connectionends, for example a “car” could have “wheel”, “seat” and “engine” in one relationship “has”. Most of the code involving connectionends is assuming there are two connectionends in Melanee, which should be fully extended to support n-ary connections to better support deep-connections feature.

Another feature that Melanee can support is code generation. To this end I developed basic code generation functions that can generate entity code and connection code in Java language. Entities and connections all can generate separate classes, by considering connection as a class, and in both destinations, the connection class is stored in an ArrayList. But the PLM now supports deep-connection by extended LML, and the generated code of the multi-level model can be further explored of how the connections can be better represented in the code. There are two possible ways of representing relationships in code:

The first one is to include the relationship in the generated entity code. In this approach, each relationship is represented inside the entity, hence navigating through relationship to the other destination is straightforward. But the different kind of relations: one-to-one, many-to-one and many-to many relationships must be implemented differently in the entity code, which cannot represent the idea of showing connections as clabjects in the generated code.

The other approach is to construct relationships into independent classes, which is what I have done for now. They are generated as independent classes to represent the relationships. In this approach the relationships are clearly represented, and the three kind of relation can be easily implemented inside the connection class. But it must generate more code, and navigation through relations to reach the other destinations need to be performed through independent connection classes, which is not as convenient as navigation using the first option.

These approaches needs be discussed, compared and experimented. Each approach has its pros and cons. The most fitting one can be implemented as a feature for Melanee so that it can provide better support for multi-level modeling environment.

## 7 Conclusion

In this thesis, I first introduced multilevel modeling and then discussed Melanee, PLM and LML, which is a tool that supports dynamic number of modeling levels. Then to provide better support for multi-level modeling environment and further explore relationships between entities in models. First I implemented deep-connection feature proposed in [7] in Melanee by extending PLM and LML language. Deep-connection feature allows modelers to create models with deep-connections that allow connections in Melanee not only be treated as clabject but also have been implemented with a unified semantics that increased scalability and expressiveness for connections in multi-level approaches. Monikers with potency values can describe instances within specific levels below with the help of potency. Each connectionends also been implemented with deep-multiplicities feature, and with the help of potency value for each multiplicity, deep-multiplicities can describe the boundaries of the number of connection instance in many levels down. Validations regarding deep-connections to ensure validity of the whole model have been implemented as well.

Then to add support to Melanee so that models can be constructed not only in “constructive approach”, but also in an “exploratory approach”, which is constructing models from instance level up to the type level is now possible. By introducing attributes into PLM to allow modelers specify type names for each entity, connectionend and connection, modelers can have options to start when constructing multi-level models in Melanee. Each entity and connection are validated so that if the type has not been defined, then validations of “Entity Conformance” and “Connection Conformance” functions can present heuristic messages regarding how to define instance component names so that “Entity Conformance” and “Connection Conformance” can be guaranteed to achieve “Level Conformance”. The processing of possible types for connections adopts the TS proposed in [7], and weighted sum calculation was designed and implemented with specific weight design for each component to calculate the possibility of each connection as the type for a connection instance. The weight for each component was discussed and

experimented and then implemented. Three categories of matching: 100% match, possible match and incompatible match were established and presented to modelers with ranking from the most possible ones to the lowest possible ones except for the incompatible match matches.

The usability of Melanee was increased by providing the exploratory approach of constructing models, and by the quick-fix sections inside each validation. Resolving types issues for entities and connections are more convenient by extending Melanee using EVL with EOL language.

Then all the implementation was evaluated with model sets and experiments to prove their validities. For connection conformance, many possible scenarios of connection type matching scenarios are tested and discussed in Evaluation Chapter.

The implemented of idea in [7] and the “explanatory modeling” in Melanee provide support for further exploration of relationships in multi-level modeling. I hope the introduction related to Melanee could facilitate the development process of future contribution work for Melanee, and the work done in this paper can provide help and support further experiments and researches to push relationship related study and multi-level modeling to the next level.

## Reference

- [1] Kuehne T, Schreiber D. Can programming be liberated from the two-level style: multi-level programming with deepjava[C], *ACM SIGPLAN Notices*. ACM, 2007, 42(10): 229-244.
- [2] Atkinson C, Kühne T. Reducing accidental complexity in domain models[J]. *Software & Systems Modeling*, 2008, 7(3): 345-359.
- [3] Atkinson C, Kühne T. Meta-level independent modelling[C], *International Workshop on Model Engineering at 14th European Conference on Object-Oriented Programming*. 2000, 12: 16.
- [4] Van Mierlo S, Barroca B, Vangheluwe H, et al. Multi-level modelling in the Modelverse[C], *MULTI@ MoDELS*. 2014: 83-92.
- [5] Kennel B. A unified framework for multi-level modeling[D]. *Universität Mannheim*, 2012.
- [6] Atkinson C, Kühne T. The essence of multilevel metamodeling[M], << *UML*>> 2001—*The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. Springer Berlin Heidelberg, 2001: 19-33.
- [7] Atkinson C, Gerbig R, Kuhne T. A unifying approach to connections for multi-level modeling[C], *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on*. IEEE, 2015: 216-225.
- [8] Kühne T. Matters of (meta-) modeling[J]. *Software & Systems Modeling*, 2006, 5(4): 369-385.
- [9] Kühne T. Contrasting classification with generalisation[C], *Proceedings of the Sixth Asia-Pacific Conference on Conceptual Modeling-Volume 96*. Australian Computer Society, Inc., 2009: 71-78.
- [10] Rayside D, Campbell G T. An Aristotelian understanding of object-oriented programming[C], *ACM SIGPLAN Notices*. ACM, 2000, 35(10): 337-353.

- [11] Atkinson C, Gerbig R, Kühne T. Comparing multi-level modeling approaches[C], *MULTI@ MoDELS*. 2014: 53-61.
- [12] Atkinson C. Meta-modelling for distributed object environments[C], *Enterprise Distributed Object Computing Workshop [1997]*. EDOC'97. Proceedings. First International. IEEE, 1997: 90-101.
- [13] Melanee, Deep-modeling Domain-specific Language Workbench, Version 2.0, <http://www.melanee.org/>.
- [14] Kolovos D, Rose L, Paige R, et al. *The epsilon book*[J]. Structure, 2010, 178: 1-10.
- [15] Atkinson C, Kennel B, Goß B. Supporting constructive and exploratory modes of modeling in multi-level ontologies[C], *Procs. 7th Int. Workshop on Semantic Web Enabled Software Engineering, Bonn (October 24, 2011)*. 2011.
- [16] Van Deursen A, Klint P, Visser J. *Domain-Specific Languages: An Annotated Bibliography*[J]. Sigplan Notices, 2000, 35(6): 26-36.
- [17] Hudak P. Domain-specific languages[J]. *Handbook of Programming Languages*, 1997, 3: 39-60.
- [18] OMG. Unified Modeling Language Superstructure Specification, Version 2.1.1, *OMG document formal/07-02-05*, February 2007.
- [19] OMG. MDA Guide Version 1.0.1, 2003. Version 1.0.1, *OMG document omg/03-06-01*. 8.
- [20] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *Software, IEEE*, vol. 20, no. 5, pp. 42–45, 2003.
- [21] Larman, C.: Applying UML and patterns: an introduction to object-oriented analysis and design and the unified process, 2<sup>nd</sup> (edn.) *Prentice-Hall*, Englewood cliffs (2002).
- [22] J. D. Lara, E. Guerra, and J. S. Cuadrado, "When and how to use multilevel modelling," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 2, pp. 12:1–12:46, Dec. 2014.
- [23] M. Gutheil, B. Kennel, and C. Atkinson, "A systematic approach to connectors in a multi-level modeling environment," in *MoDELS 2008*, 2008, pp. 843–857.

- [24] Steinberg D, Budinsky F, Merks E, et al. EMF: eclipse modeling framework[M]. *Pearson Education*, 2008.
- [25] Rubel D, Wren J, Clayberg E. The Eclipse Graphical Editing Framework (GEF)[M]. *Addison-Wesley Professional*, 2011.
- [26] Eclipse.org, Graphical Modeling Project (GMP), <http://www.eclipse.org/modeling/gmp/>
- [27] OMG, “Omg unified modeling language™, infrastructure version 2.4.1,” <http://www.omg.org/spec/UML/2.4.1>, 2011.
- [28] J. de Lara and E. Guerra, “Deep meta-modelling with metadepth,” in *Proceedings of the 48th TOOLS conference*, ser. LNCS. Springer, 2010, pp. 1–20.
- [29] Atkinson, C., Kühne, T.: Profiles in a strict metamodeling framework. *Journal of the Science of Computer Programming* 44(1), 5–22 (Jul 2002).
- [30] Atkinson, C., Kühne, T.: Rearchitecting the UML infrastructure. *ACM Transactions on Modeling and Computer Simulation* 12(4), 290–321 (Oct 2003).
- [31] Atkinson, C., Kennel, B., Go, B.: The level-agnostic modeling language. In: Malloy, B., Staab, S., Brand, M. (eds.) *Software Language Engineering, Lecture Notes in Computer Science*, vol. 6563, pp. 266–275. Springer Berlin Heidelberg (2011), [http://dx.doi.org/10.1007/978-3-642-19440-5\\_16](http://dx.doi.org/10.1007/978-3-642-19440-5_16).