INTEGRATING RECOMMENDER SYSTEMS INTO DOMAIN SPECIFIC MODELING TOOLS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Arvind Nair

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

May 2017

Purdue University

Indianapolis, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF THESIS APPROVAL

To My Parents.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my parents, my father Sudhakaran Nair, and my mother Beena Nair, for all the sacrifices they have done for me. They have always ensured that I did not face problems of any nature, especially financial ones, by putting my needs first always before theirs. Such great is parents' love! I will forever be indebted to my parents. This Thesis is dedicated to them, as without them, I would never be able to complete my Masters Thesis.

Next, I want to thank Dr. James H. Hill, for teaching me the fundamentals of Software Engineering and Design. I was fortunate enough to take all 3 graduate courses of Dr. Hill, *i.e.*, Management of Software Development Process, Object Oriented Analysis & Design and Software Quality Assurance. These courses taught me good software engineering practices, how to design software systems compliant to coding standards and test them effectively. The teaching methods of Dr. Hill encouraged me to push myself to develop high quality software. I got to learn a lot from being a grader twice and Teaching Assistant once for the courses under Dr. Hill along with research. While doing research under Dr. Hill, he used to encourage me to think deeply and apply all the concepts taught appropriately, as well as guiding me to do the work assigned in the best way possible. I have immense respect for Dr. Hill. I am proud to learn and be trained under Dr. Hill. All the concepts taught by Dr. Hill in his courses as well as his constant guidance, have been vital in the development of this Thesis, without which, it would not have been possible to do the same.

I also want to thank Dr. Xia Ning for agreeing to be a part of my Thesis development and member of my Thesis defense committee. I got to learn a lot from the Recommender Systems course of Dr. Ning. I thank her for being patient and making sure that all my questions are cleared. Dr. Ning has always been very prompt in responding and I greatly valued and implemented her inputs for the Recommender system and its evaluation in this Thesis.

I also want to thank Dr. Rajeev Raje for agreeing to be a member on my Thesis defense committee. I enjoyed Distributed Computing and Programming Languages courses taught by Dr. Raje. His courses taught me to effectively analyze any given scenario and find solutions to the same. It also gave me a good understanding of various programming languages & paradigms and their insights. I learnt a lot from being a Teaching Assistant for Distributed Computing course assigned to me subsequently.

I also want to thank Dr. John Gersting, whose Database Systems course I was fortunate enough to take, which helped me a lot to design the database for my Thesis.

A special thanks to Dennis Feiock for helping me in the initial setup of the GAME environment. Also, thanks to Dr. Manjula Peiris for answering all my questions while being the Teaching Assistant when I took courses of Dr. Hill. I also want to thank Nicole Wittlief for helping me in all administrative works and answering my questions promptly no matter how busy she was. I also thank the Computer Science Department for granting me the Dean's scholarship which included grading assignments and also two times Teaching Assistantship. These helped me gain more relevant experience, along with financially sustaining my stay in the USA. Any work assigned to me, I have ensured that it is done with utmost sincerity, devotion and hard work.

Finally, I would also like to thank my friends who have helped me in many ways, in my life, here in the USA and have also reciprocated always to my maximum capacity whenever possible.

I am very fortunate to have great professors and I have done my best to gain, as much knowledge and experience, as I can from my Masters degree study and work assignments. I would like to apply the same in real life applications and do something good and useful that benefits humanity.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

DSML	Domain Specific Modeling Language
DSMT	Domain Specific Modeling Tool
XML	eXtensible Markup Language
GME	Generic Modeling Environment
GAME	GME Automated Modeling Environment
MDE	Model Driven Engineering
PME	Proactive Modeling Engine
LMS	Library Management System
GMS	Garage Management System
OCL	Object Constraint Language
PME	Proactive Modeling Engine
UML	Unified Modeling Language
OMG	Object Management Group
MTBE	Model Transformation By Example
MTBD	Model Transformation By Demonstration
ARHR	Average Reciprocal Hit-Rank
SUS	System Usability Scale
PICML	Platform Independent Component Modeling Language
FCO	First Class Object
GUI	Graphical User Interface

GLOSSARY

user	a person who uses the software
modeler	person who performs modeling using domain-specific modeling tools
metamodeler	person who creates the metas or blueprints for modeling

ABSTRACT

Nair, Arvind. M.S., Purdue University, May 2017. Integrating Recommender Systems into Domain Specific Modeling Tools. Major Professor: James H. Hill.

This thesis investigates integrating recommender systems into model-driven engineering tools powered by domain-specific modeling languages. The objective of integrating recommender systems into such tools is overcome a shortcoming of proactive modeling where the modeler must inform the model intelligence engine how to progress when it cannot automatically determine the next modeling action to execute (*e.g.*, add, delete, or edit). To evaluate our objective, we integrated a recommender system into the Proactive Modeling Engine, which is a add-on for the Generic Modeling Environment (GME). We then conducted experiments to both subjective and objectively evaluate the enhancements to the Proactive Modeling Engine.

The results of our experiments show that integrating recommender system into the Proactive Modeling Engine results in an Average Reciprocal Hit-Rank (ARHR) of 0.871. Likewise, the integration results in System Usability Scale (SUS) rating of 77. Finally, user feedback shows that the integration of the recommender system to the Proactive Modeling Engine increases the usability and learnability of domain-specific modeling tools.

1 INTRODUCTION

Model Driven Engineering (MDE) [1] is used to address platform complexity and express domain concepts effectively using simple graphical representations. MDE mainly consists of two parts: (1) domain-specific modeling languages (DSMLs), which makes use of *metamodels* to describe relationships in a domain graphically; and (2) transformation engines and generators, which synthesize artifacts (*e.g.*, source code, XML deployment description, and configuration files) from models. Examples of MDE tools that use DSMLs include, but is not limited to: the Generic Modeling Environment (GME) [2], the Generic Eclipse Modeling System (GEMS) [3], the Eclipse Modeling Framework (EMF) [4], and Domain Specific Language (DSL) Tools [5].

Traditionally, the process of creating models using DSMLs is manual. In this manual process, the modeler must possess pre-requisite knowledge about the target domain before creating models within the domain that address their application needs. Modelers can also leverage other options to assist with creating models like constraint solvers [6–8] and model guidance [9,10]. Though, at first, constraint solvers and model guidance help the modeler, the *modeling effort* increases as the number of modeling elements and their constraints in the domain increase. Moreover, the modeler must track of the structure, functionalities, state and implementation of each of the modeling elements [11]. For example, Platform Independent Component Modeling Language (PICML) [12] is a large-scale DSML that contains approximately 930 modeling elements and 130 constraints, and has high modeling effort [11] due to the aforementioned reasons.

The limitations discussed above led to the introduction of *proactive modeling* [13]. Proactive modeling is a model intelligence technique that foresees model transformations, automatically executes them, and prompts the modeler for assistance when necessary. Proactive modeling has been shown [13] to reduce modeling effort by both automatically generating required model elements, and guiding modelers to select what actions should be executed on the model.

However, once the initial model is generated automatically from the deterministic modeling actions using proactive modeling, the *non-deterministic* modeling actions of the model must be executed manually by the modeler using model guidance of PME. Unfortunately, for large and complex DSMLs, such as PICML, the completion of non-deterministic actions on each modeling element can become overwhelming especially for novice modelers who need to use model guidance to complete the same. For example, in PICML, the `Implementation Artifact` element is part of the `Implementation Artifact Descriptor` which is generated by PICML as one of the deployment descriptors [12]. The `Implementation Artifact` element has 4 connections and 7 attributes whose values require user intervention. So, each `Implementation Artifact` added must have their associated non-deterministic modeling actions completed. Similarly, model guidance is needed for other modeling elements which contain non-deterministic modeling actions to be completed.

To address this limitation in proactive modeling, we have explored integrating recommender systems [14] with proactive modeling. The goal of the integration is to overcome the shortcoming of proactive modeling where the modeler must inform the model intelligence engine how to progress when it cannot automatically determine the next modeling action to execute (*e.g.*, add, delete, or edit). It aims to improve the overall user experience in terms of usability and learnability of domain specific modeling tools by helping all modelers, especially novice modelers, work with new DSMLs and also reducing the modeling effort further.

Based on this understanding, this thesis has the following contributions to MDE:

- First, it identifies *Object Constraint Language (OCL) [15] expression failures* which helps in identifying deterministic and non-deterministic modeling actions.

- Second, it introduces a new recommendation parameter and its calculation called as *action presence based recommendation*, in the area of MDE and DSMLs. This parameter is based on the number of times a particular modeling action was not presented to the modeler due to OCL constraint satisfaction.

- Third, it shows how recommendation parameters can be combined mathematically using ensemble learning to perform recommendations for modeling actions in domain specific modeling tools.

The results of our experiments show that integrating recommender system into the Proactive Modeling Engine results in an Average Reciprocal Hit-Rank (ARHR) of 0.871. Likewise, the integration results in System Usability Scale (SUS) rating of 77. Finally, user feedback shows that the integration of the recommender system to the Proactive Modeling Engine increases the usability and learnability of domain-specific modeling tools.

1.1 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 gives an overview of all the works similar to what has been done in the area of model generation and transformation using constraints or guidance systems. Chapter 3 describes two modeling scenarios of Car Rental System and Library Management System by describing their metamodels, constraints as well as a sample model. Chapter 4 describes the 6 modeler based recommendation factors considered for our recommender system and how they are combined to recommend modeling actions to the modeler. Chapter 5 describes as to how the recommender system is implemented and integrated in GME along with PME. Chapter 6 describes our experiments based on the two modeling scenarios in the examples chapter which is used to evaluate our recommender system. Lastly, Chapter 7 describes what we have achieved and what the future direction of the research should be in the area of recommender systems in domain-specific modeling tools.

2 RELATED WORKS

This chapter compares our work on integrating recommender systems into domain-specific modeling tools to other works from the areas of model generation and transformation using constraints or guidance systems.

2.1 Model Completion Recommendations

Sen et al. [8] presented an integrated software system capable of generating recommendations for model completion of partial models built in arbitrary domain-specific model editors. This automatic completion feature is powered by a Prolog engine whose input is a Constraint Logic Program (CLP) obtained by a model transformation from models in multiple languages: the meta-model (as a class diagram), constraints on it which are randomly shuffled, and a partial model [8]. The Prolog engine solves the generated logic program and the solution is returned to the model editor as a set of recommendations for properties of the partial model [8]. Our approach is different from the work of Sen et al. [8] from the fact that, we use ensemble learning to combine various recommendation parameters based on sequences, frequent item set (*i.e.*, modeling actions frequently performed by the modeler), Markoff Decision Processes (MDPs) [16], context, temporal dynamics (*i.e.*, at what point of time in the modeling process the modeling action was performed by the modeler) and constraint satisfaction to determine which modeling actions must be given priority and recommended based on the modeling activity of the modeler.

2.2 Model Guidance Techniques

Hessellund et al. [7] presented SmartEMF, which is an extension of the Eclipse Modeling Framework. SmartEMF provides support for representing, checking, and maintaining constraints in the context of multiple loosely-coupled DSLs. It also computes various model guidance actions for the user in a given context. It, however, does not recommend modeling actions. Our approach differs from the fact that it recommends actions related to the selected object dynamically to modelers based on ensemble learning integrated with proactive modeling and also provides the user guidance to create modeling actions that are of general nature as described in Section 5.1.1.

White et al [6] created a Domain-Specific Intelligence Framework (DSIF) to help a modeler solve combinatorially challenging modeling problems on large and complex models. The DSIF generates a knowledge-base and users specify constraints in a declarative format which is used to derive modeling solutions. Our approach differs in that it uses auto-completion of deterministic modeling actions and recommends the non-deterministic modeling actions based on learning from modeler activity. The modeler, in our approach, does not have to deal with specifying constraints in a declarative format. Instead, the modeler completes the model by performing modeling actions which are recommended by our system.

Janota et al. [10] have created Interactive Model Derivation where the modeler is given the list of possible operations to perform on the model. Their guidance algorithm gives *advice* in form of operations and the modeler selects from the exhaustive advice [10]. Our approach, however, gives a list of all possible non-deterministic actions and their order changes as per the modeler activity. Our work differs in that if the model that the modeler is working with, has thousands of modeling actions, then those modeling actions most likely to be selected, will be displayed first. This is done by learning from the modelers activity for the corresponding DSML.

White et al. [9] developed model intelligence which uses domain constraints in Object Constraint Language (OCL) to guide modelers. It presents valid modeling actions based on selected element relations and constraints. In our approach, PME generates a partial model by performing automatically deterministic modeling actions derived from solving OCL constraints and then recommends the non-deterministic modeling actions from the failed OCL constraint expressions, which are non-deterministic (see Section A) based on modeler activity.

2.3 Model Transformation Techniques

Varro [17] introduced *Model Transformation By Example (MTBE)* approach where models can be transformed using source target modeling pairs. Kappel [18] discussed MTBE approaches that address the issue of difficulty for modelers to develop model transformations using abstract syntax of modeling languages(metamodels and mappings using OCL) because they are familiar with the concrete syntax (graphical elements), but not with its computer internal representation (metamodels and OCL). Sun [19] introduced *Model Transformation By Demonstration (MTBD)* approach, which creates of model transformations by recording and analyzing the operational behavior exhibited by an end user. Our approach is similar to MTBE and MTBD because it learns the priority of modeling actions by using modeler activities as examples and demonstrations on the model level (concrete syntax). However, our approach is different from both MTBE and MTBD approaches, when the recommender system learns and recommends the future non-deterministic modeling actions towards model completion in cohesion with auto-completion of deterministic modeling actions using PME.

3 MOTIVATING EXAMPLES

This chapter introduces two example DSMLs used throughout this thesis to discuss our approach for integrating recommender systems into domain-specific modeling tools.

3.1 The Car Rental System

The first example is the *Car Rental System (CRS)*. The CRS allows representatives to approve persons, who are considered customers, to rent cars. A person, however, does not have to be a customer to view a car (*i.e.,* a guest).

3.1.1 Modeling Elements

There are many ways to design a metamodel for CRS. Figure 3.1 presents a simple metamodel for the CRS. As shown in this figure, the metamodel is the `Garage`, which is place where the rentals take place. It contains the following model elements:

- The `Customer` represents the person who wants to rent a car;
- The `Representative` is the person who approves the car rental;
- The `Car` represents the car being rented; and
- The `Mechanic` represents the person working in the `Garage`, who works on `Car` models.

The attributes of the modeling elements of CRS have been discussed in Appendix D.

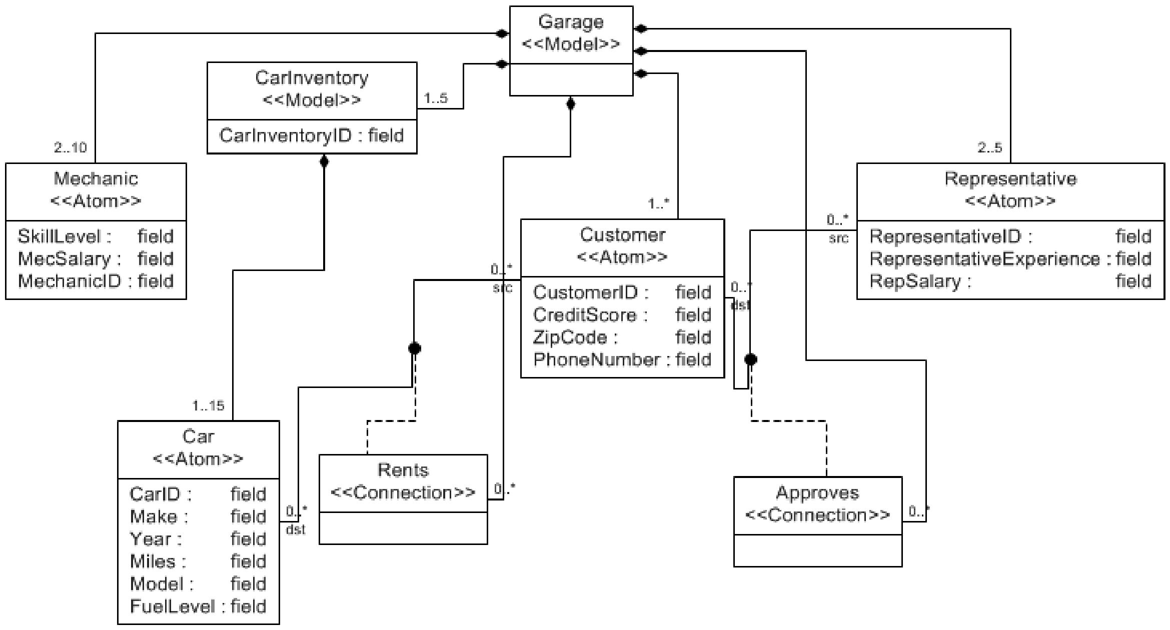


Figure 3.1. An example metamodel for the Car Rental System (CRS).

3.1.2 Constraints

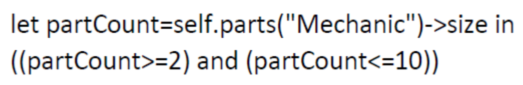
The constraints for the CRS metamodel, which are expressed in the Object Constraint Language (OCL) [15], are as follows:

- **Number of Car Inventories Required.** This constraint checks the minimum and maximum number of Car Inventories required by Garage. This constraint is generated automatically by GME from specifying the cardinality between the Garage and CarInventory containment connection. The OCL constraint is shown in Figure 3.2 where the Car Inventories contained by Garage must be in the range of 1 to 5.

```
let partCount=self.parts("CarInventory")->size in
((partCount>=1) and (partCount<=5))
```

Figure 3.2. Car Inventories Containment Constraint

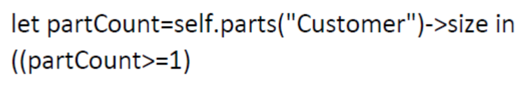
- **Number of Mechanics Required.** This constraint checks the minimum and maximum number of Mechanics required by Garage. This constraint is generated automatically by GME from specifying the cardinality between the Garage and Mechanic containment connection. The OCL constraint is shown in Figure 3.3 where the Mechanics contained by Garage must be in the range of 2 to 10.



```
let partCount=self.parts("Mechanic")->size in
((partCount>=2) and (partCount<=10))
```

Figure 3.3. Mechanic Containment Constraint

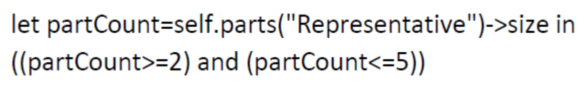
- **Number of Customers Required.** This constraint checks the minimum number of Customers required by Garage. This constraint is generated automatically by GME from specifying the cardinality between the Garage and Customer containment connection. The OCL constraint is shown in Figure 3.4 where the Customers contained by Garage must be 1 or greater.



```
let partCount=self.parts("Customer")->size in
((partCount>=1)
```

Figure 3.4. Customer Containment Constraint

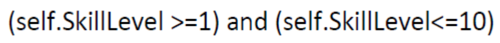
- **Number of Representatives Required.** This constraint checks the minimum and maximum number of Representatives required by Garage. This constraint is generated automatically by GME from specifying the cardinality between the Garage and Representative containment connection. The OCL constraint is shown in Figure 3.5 where the Representatives contained by Garage must be in the range of 2 to 5.



```
let partCount=self.parts("Representative")->size in
((partCount>=2) and (partCount<=5))
```

Figure 3.5. Representative Containment Constraint

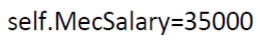
- **Skill Level Range Condition.** This constraint checks the skill level attribute value of the Mechanic. This constraint is added manually by the `metamodeler`. The OCL constraint is shown in Figure 3.6 where the skill level value must be in the range of 1 to 10.



```
(self.SkillLevel >=1) and (self.SkillLevel<=10)
```

Figure 3.6. Mechanic Skill Level Constraint

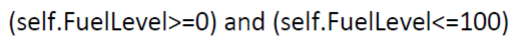
- **Mechanic Salary Required Condition.** This constraint checks if the salary attribute of the Mechanic is the assigned value. This constraint is added manually by the metamodeler. The OCL constraint is shown in Figure 3.7 where the salary value must be 35000.



```
self.MecSalary=35000
```

Figure 3.7. Mechanic Salary Constraint

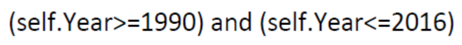
- **Car Fuel Level Range Condition.** This constraint checks if the fuel level attribute of the Car is in the specified range. This constraint is manually added by the metamodeler. The OCL constraint is shown in Figure 3.8 where the fuel level value must be in the range of 0 to 100.



```
(self.FuelLevel>=0) and (self.FuelLevel<=100)
```

Figure 3.8. Car Fuel Level Constraint

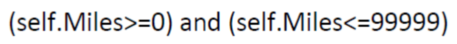
- **Car Year Range Condition.** This constraint checks if the year attribute of the Car is in the specified range. This constraint is manually added by the metamodeler. The OCL constraint is shown in Figure 3.9 where the year value must be in the range of 1990 to 2016.



```
(self.Year>=1990) and (self.Year<=2016)
```

Figure 3.9. Car Year Constraint

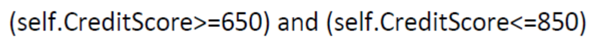
- **Car Miles Driven Range Condition.** This constraint checks if the miles attribute of the Car is in the specified range. This constraint is manually added by the metamodeler. The OCL constraint is shown in Figure 3.10 where the miles value must be in the range of 0 to 99999.



```
(self.Miles>=0) and (self.Miles<=99999)
```

Figure 3.10. Car Miles Constraint

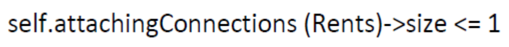
- **Customer Credit Score Range Condition.** This constraint checks if the credit score attribute of the Customer is in the specified range. This constraint is manually added by the metamodeler. The OCL constraint is shown in Figure 3.11 where the credit score value must be in the range of 650 to 850.



```
(self.CreditScore>=650) and (self.CreditScore<=850)
```

Figure 3.11. Customer Credit Score Constraint

- **Customer Car Rental Limit Condition.** This constraint checks if the Rents connection from Customer to Car is below or equal to the specified value. This constraint is manually added by the metamodeler. The OCL constraint is shown in Figure 3.12 where the number of Rents connection from Customer to Car must be lesser than or equal to 1.



```
self.attachingConnections (Rents)->size <= 1
```

Figure 3.12. Customer Car Rents Constraint

- **Representative Salary Range Condition.** This constraint checks if the salary attribute of the Representative is in the specified range. This constraint is manually added by the metamodeler. The OCL constraint is shown in Figure 3.13 where the salary value must be in the range of 40000 to 50000.

```
(self.RepSalary>=40000) and (self.RepSalary<=50000)
```

Figure 3.13. Representative Salary Constraint

- **Representative Experience Condition.** This constraint checks if the experience attribute of the Representative is more than the specified value. This constraint is manually added by the metamodeler. The OCL constraint is shown in Figure 3.14 where the representative experience value must be more than 1.

```
(self.RepresentativeExperience>1)
```

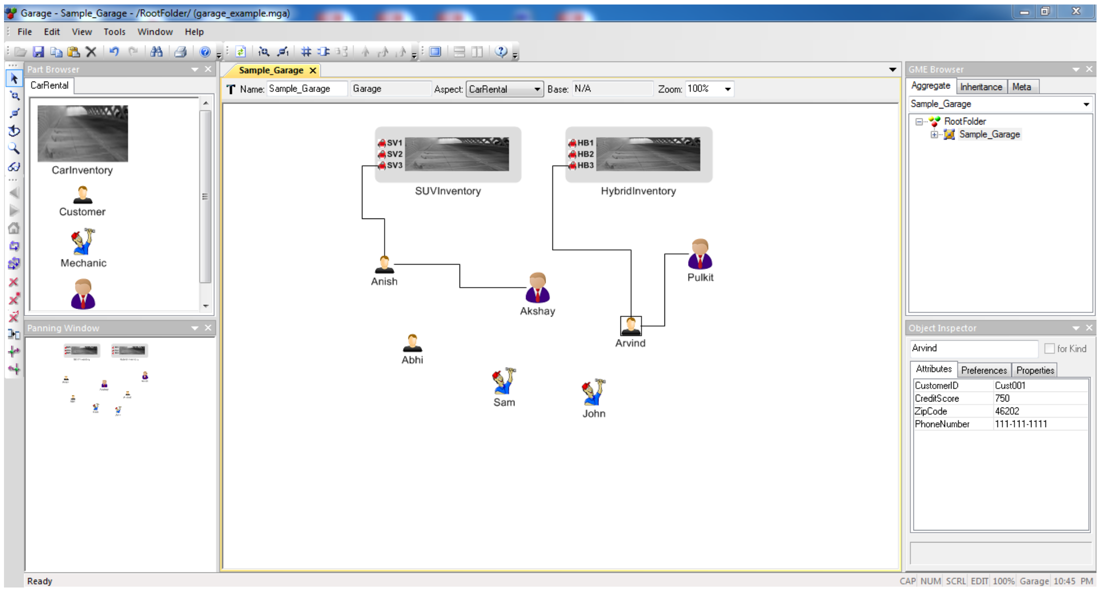
Figure 3.14. Representative Experience Constraint

3.1.3 An Example Model using the CRS Metamodel

Figure 3.15 illustrates an example model created from the CRS metamodel. As shown in this figure, the modeler has selected the *Arvind* Customer model element. In the `Object Inspector Window`, shows the attributes for the selected Customer object. In this case, the Object Inspector Window shows the customer id, credit score, zip code and phone number. The Representative *Pulkit* approves Customer *Arvind*, and Customer *Arvind* rents Car *HB3* from CarInventory *HybridInventory*.

3.2 The Library Management System

The second example is the *Library Management System (LMS)* [14] which is used to assist librarians to monitor the book inventory and monitor the book borrowing process by patrons.



Arvind

CustomerID	Cust001
CreditScore	750
ZipCode	46202
PhoneNumber	111-111-1111

Figure 3.15. A example model created from the CRS metamodel.

3.2.1 Modeling Elements

There are many ways to design a metamodel for LMS. Figure 3.16 represents a simple metamodel for LMS. As shown in this figure, the metamodel is the Library, which is the place where the book borrowing activity takes place. It contains the following modeling elements:

- The `Book` represents the book being borrowed;
- The `Patron` is the person who borrows the book;
- The `Librarian` represents the person who works for the `Library`, monitoring book borrowing activity; and
- The `PatronRef` represents a person who is a Patron from other libraries.

The attributes of the modeling elements of LMS have been discussed in Appendix D.

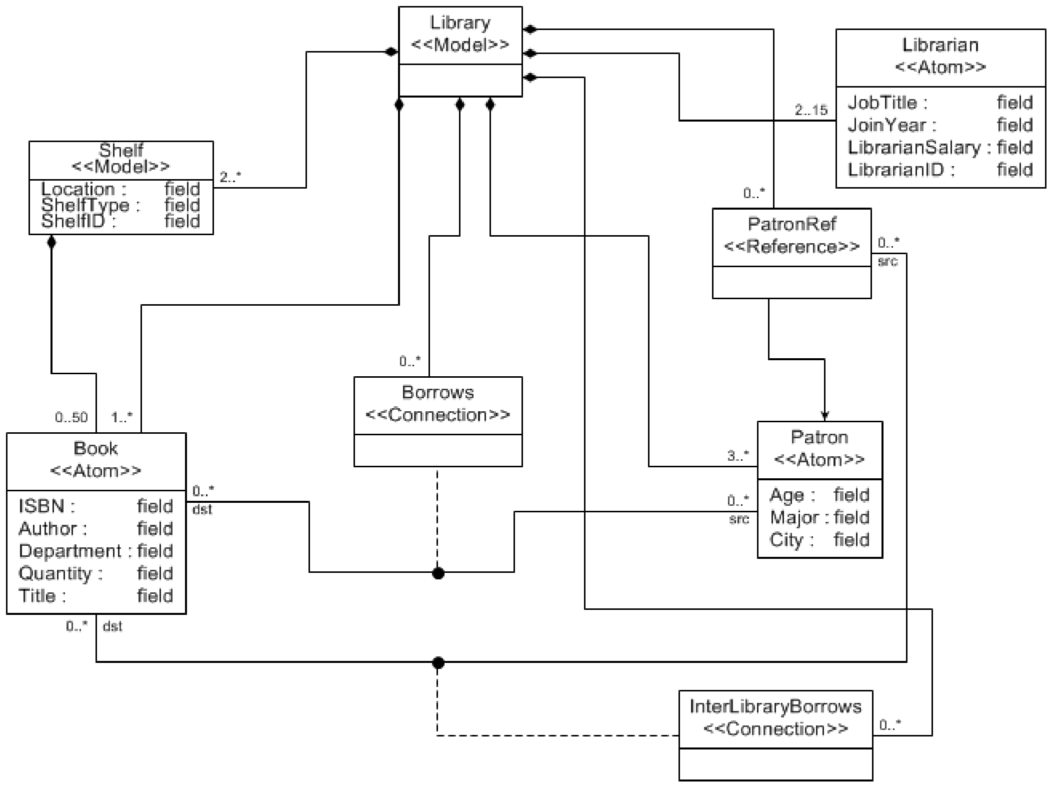


Figure 3.16. A Sample Metamodel for LMS.

3.2.2 Constraints

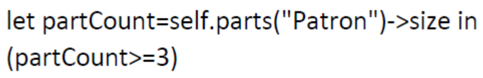
In the LMS, we have the following constraints expressed in OCL [15]:

- **Number of Librarians Required.** This constraint checks the minimum and maximum number of Librarians required by Library. This constraint is generated automatically by GME from specifying the cardinality between the Library and Librarian containment connection. The OCL constraint is shown in Figure 3.17 where the Librarians contained by Library must be in the range of 2 to 15.

```
let partCount=self.parts("Librarian")->size in
((partCount>=2) and (partCount<=15))
```

Figure 3.17. Librarian Containment Constraint

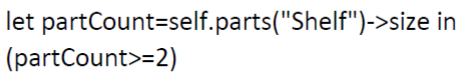
- **Minimum Number of Patrons Required.** This constraint checks the minimum number of Patrons required by Library. This constraint is generated automatically by GME from specifying the cardinality between the Library and Patron containment connection. The OCL constraint is shown in Figure 3.18 where the Patrons contained by Library must be 3 or more.



```
let partCount=self.parts("Patron")->size in
(partCount>=3)
```

Figure 3.18. Patron Containment Constraint

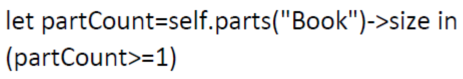
- **Minimum Number of Shelves Required.** This constraint checks the minimum number of Shelves required by Library. This constraint is generated automatically by GME from specifying the cardinality between the Library and Shelf containment connection. The OCL constraint is shown in Figure 3.19 where the Patrons contained by Library must be 2 or more.



```
let partCount=self.parts("Shelf")->size in
(partCount>=2)
```

Figure 3.19. Shelf Containment Constraint

- **Number of Books Required.** This constraint checks the minimum number of Books required by Library. This constraint is generated automatically by GME from specifying the cardinality between the Library and Book containment connection. The OCL constraint is shown in Figure 3.20 where the Books contained by Library must be 1 or more.



```
let partCount=self.parts("Book")->size in
(partCount>=1)
```

Figure 3.20. Book Containment Constraint

- **Book ISBN Condition.** This constraint checks if the ISBN Attribute of the Book is the specified value. It should not equal the specified value. This constraint is added manually by the metamodeler. The OCL constraint is shown in Figure 3.21 where the ISBN value must not be empty.

```
self.ISBN <> ""
```

Figure 3.21. Book ISBN Constraint

- **Patron Minimum Age Requirement.** This constraint checks if the age attribute of the Patron is more than or equal to the specified value. This constraint is manually added by the metamodeler. The OCL constraint is shown in Figure 3.22 where the age of the Patron must be 18 or greater.

```
self.Age >= 18
```

Figure 3.22. Patron Age Constraint

- **Patron Book Borrowing Limit.** This constraint checks if the Borrows connection from Patron to Book is below or equal to the specified value. This constraint is manually added by the metamodeler. The OCL constraint is shown in Figure 3.23 where the number of Borrows connection from Patron to Book must be lesser than or equal to 2.

```
self.attachingConnections (Borrows)->size <= 2
```

Figure 3.23. Patron Book Borrow Constraint

- **Librarian Salary Range Condition.** This constraint checks if the salary attribute of the Librarian is the assigned value as specified by the OCL constraint. This constraint is added manually by the metamodeler. The OCL constraint is shown in Figure 3.24 where the salary value must be in the range of 20000 to 30000.

```
(self.LibrarianSalary>=20000) and (self.LibrarianSalary<=30000)
```

Figure 3.24. Librarian Salary Constraint

- **Librarian Identification Condition.** This constraint checks if the librarian id attribute of the Librarian is the specified value. It should not equal the specified value. This constraint is added manually by the metamodeler. The OCL constraint is shown in Figure 3.25 where the librarian id value must not be empty.

```
self.LibrarianID <> ""
```

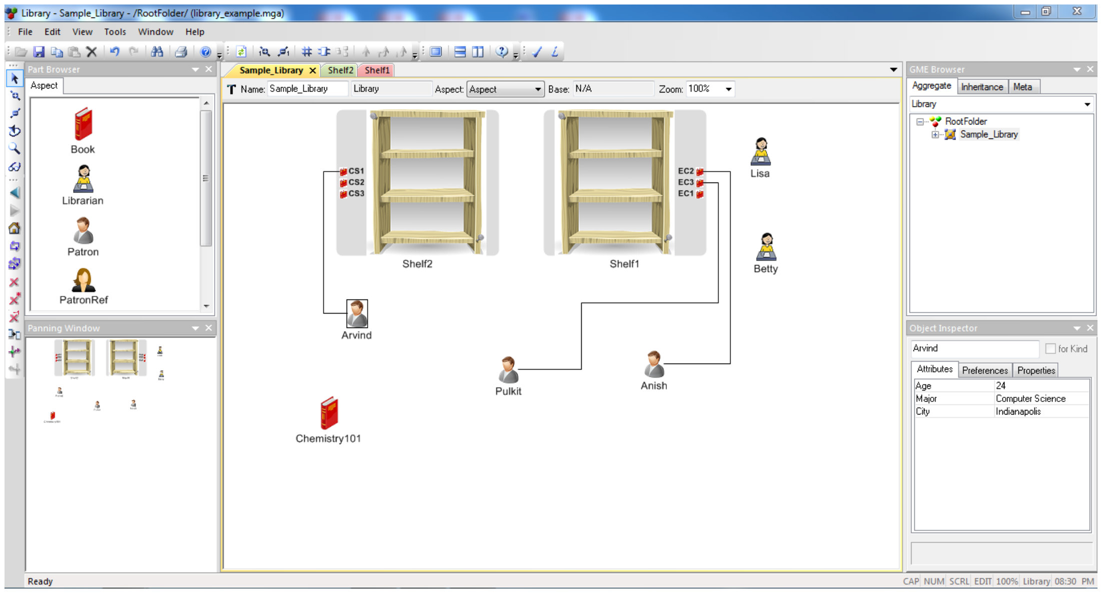
Figure 3.25. Librarian Identification Constraint

3.2.3 An Example Model using the LMS Metamodel

Figure 3.26 illustrates an example model created from the LMS metamodel. As shown in this figure, the modeler has selected the *Arvind* Patron model element. In the `Object Inspector Window`, shows the attributes for the selected Customer object. In this case, the Object Inspector Window shows the age, major and city. The Patron *Arvind* borrows Book *CS1* from Shelf *Shelf2*.

3.3 Current Limitations of Proactive Modeling

Suppose, we create a sample Garage model from the CRS metamodel. Proactive modeling without recommender systems will automatically perform the deterministic modeling actions, such as Customer containment constraint as shown in Figure 3.4, by adding one Customer element when the Garage model in created. The non-



Arvind

Age	24
Major	Computer Science
City	Indianapolis

Figure 3.26. A Sample Library Example

deterministic modeling actions such as, adding more Customers to complete the partial Garage model are performed using model guidance and model guidance handler does not save these actions [14]. Unfortunately, this increases the complexity of using proactive modeling. Ideally, proactive modeling should remember these modeling actions and recommend the actions within the correct context to alleviate this usage complexity.

From this simple example, we can infer that while using only proactive modeling, the modeler must inform the model intelligence engine how to progress when it cannot automatically determine the next modeling action to execute (*e.g.*, add, delete, or edit). Although proactive modeling is designed to act in this manner, it is also a shortcoming of proactive modeling. The remainder of this thesis will therefore discuss how we use recommender systems to address this inherent complexity in proactive modeling.

4 APPROACH

This chapter discusses our approach to integrating recommender systems into DSMLs. We first give an overview of what we mean by recommender systems. We then discuss our conceptual approach for integrating recommender systems into DSMLs. Lastly, we discuss, in detail, the theoretical aspect of our approach, and how it is realized in the Proactive Modeling Engine.

4.1 Concept Integration of Recommendation Systems with Proactive Modeling

Recommender systems are software tools and techniques providing *suggestions* for *items* to be of use to a user [20–22]. In the above definition of recommender systems, the suggestions relate to various decision-making processes and items denote what the system recommends to users [23]. Recommender systems are primarily directed towards individuals who lack sufficient personal experience or competence to evaluate the potentially overwhelming number of alternative items [23]. Also, recommender systems can also be used in various scenarios, such as web search [24] and e-commerce [25] to name a few.

By considering these applications of recommender systems, an effort has been made, in this thesis, to apply a similar concept in domain specific modeling tools and DSMLs in the context of proactive modeling. In traditional proactive modeling, once the partial model is created by completing the deterministic modeling actions, the modeler must use model guidance to complete the non-deterministic modeling actions. Model guidance does not have an intelligent presentation of the modeling actions based on modeler activity and also does not remember the modeling actions.

When the recommender system is integrated in proactive modeling, the non-deterministic modeling actions are recommended to the modeler in an intelligent and unified format (see Section 5.1.1) based on the modeler activity. The recommender system helps overcome the shortcoming of proactive modeling where the modeler must inform the model intelligence engine how to progress when it cannot automatically determine the next modeling action to execute (*e.g.*, add, delete, or edit). The recommender system considers the frequency, changing modeler preferences, contextual information of modeling actions, modeler sequences and MDPs associated with modeling actions as well as OCL constraint satisfaction as part of the modeler activity to recommend the modeling actions. It also saves and recommends the general modeling actions created by the modeler (see Section 5.1.1).

An illustration is shown in Figure 4.1 as to how the recommender system works when integrated in proactive modeling. When the modeler starts the modeling activity, proactive modeling generates a partial model by completion of all the deterministic modeling actions. The non-deterministic modeling actions are then recommended to the modeler. If the performance of recommended modeling actions result in deterministic modeling actions, they are automatically completed by proactive modeling. If the performance of recommended modeling actions results in non-deterministic modeling actions, they are completed using the recommender system by the modeler. This creates a cycle between proactive modeling and the recommender system until the model is completed as per the requirements of the modeler.

4.2 Recommender Systems in Domain Specific Modeling Tools

We need to develop a dynamic model based on different factors which impact how a modeler constructs a model in a DSML. This dynamic model is used to develop a recommender system that recommends the non-deterministic modeling actions. Based on this need, we have identified six modeler based recommendation factors that can contribute to this model.

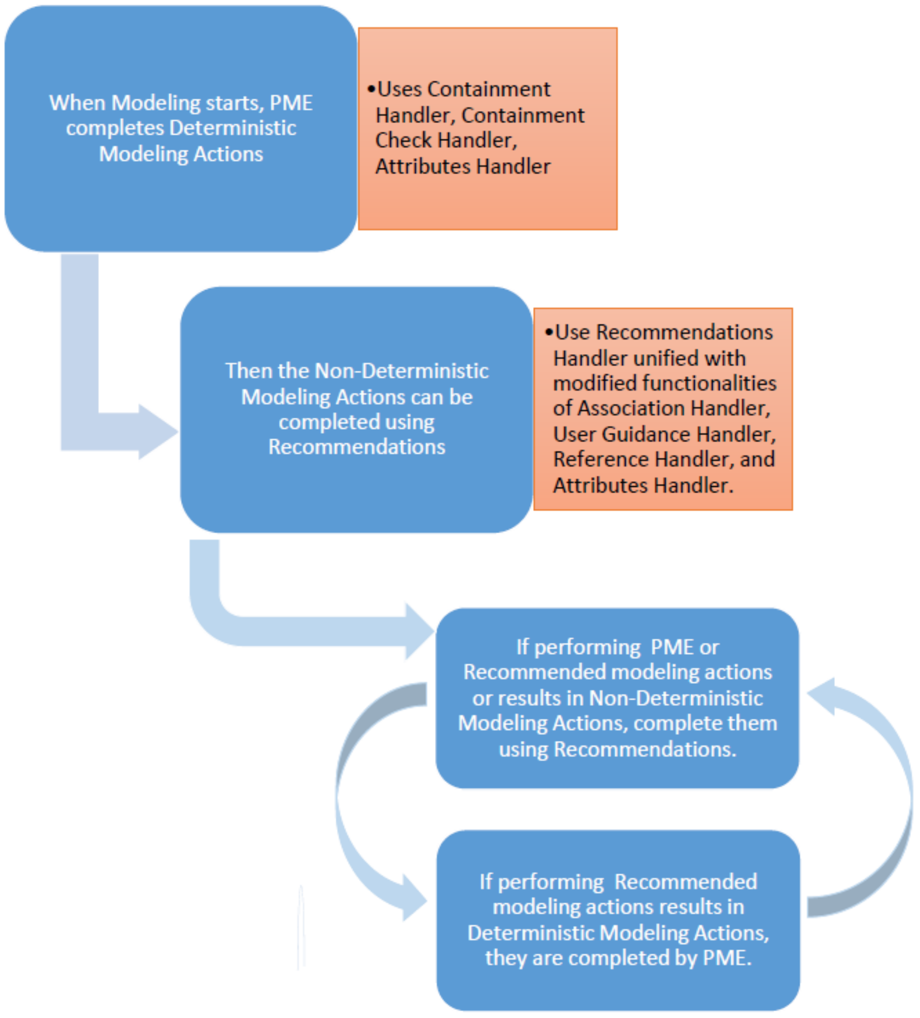

When Modeling starts, PME completes Deterministic Modeling Actions
•Uses Containment Handler, Containment Check Handler, Attributes Handler
Then the Non-Deterministic Modeling Actions can be completed using Recommendations
•Use Recommendations Handler unified with modified functionalities of Association Handler, User Guidance Handler, Reference Handler, and Attributes Handler.
If performing PME or Recommended modeling actions or results in Non-Deterministic Modeling Actions, complete them using Recommendations.
If performing Recommended modeling actions results in Deterministic Modeling Actions, they are completed by PME.

Figure 4.1. Recommender System integrated in Proactive Modeling Working

4.2.1 Recommendation Factors

1. **Sequence Based Recommendation**: This recommendation factor is based on the sequence in which the modeler performs the modeling actions associated with the meta of the modeling elements. If the set of modeling actions are performed in the same order for two modeling elements of the same meta, then for the other modeling elements of the same meta, the modeling actions are recommended based on the sequence order. This continues till the modeler does not break the sequence order of performance of modeling actions. Clegg et al. [26]

suggested that subjects can learn *sequences* based on different information in a hierarchical representation, including either sequences of stimuli or sequences of responses which can occur both with and without explicit awareness of the sequence. In our domain, it can be interpreted that, while completing various modeling actions, modelers tend to follow a sequence of actions repeatedly for specific objects. This is important as it identifies the modeler sequences of modeling actions.

2. **Count Based Recommendation**: This recommendation factor is based on the frequency of performance of the modeling actions with the associated meta. This is based on the notion of *frequent itemset* [27, 28] recommendation and results of items which can be of special interest to the user [29]. In our domain, a modeler is more likely to choose a modeling action associated with a meta that has been frequently selected for the modeling elements of that meta. This parameter is important because the non-deterministic modeling actions that are not completed for the remaining modeling elements of a specific meta will be repeated.

3. **History Based Recommendation**: This recommendation factor is based on *temporal dynamics* that may affect the selection of modeling actions by the modeler. In our domain, the point of time during the modeling process when the modeling action is selected is considered. Koren [30] states that user preferences can be expected to change over time and these must be captured by a temporal model that assumes drifting nature of the customer. Capturing time drifting patterns in user behavior is essential to improving accuracy of recommender systems, which therefore must be a predominant factor in building recommender systems [30]. We therefore consider a modeling action that occurs more recently is more likely to be chosen by the modeler, as compared to a different modeling action that was selected long ago. This parameter is therefore important because

it can be used to detect the change in modeler preferences during the modeling process.

4. **Action Context Based Recommendation**: Action context based recommendation is the factor based on *Markov Decision Processes(MDP)* [16]—a well known stochastic model of sequential decisions. In our domain, we find the probability of performing the next modeling action—after a particular modeling action is performed—independent of the previous modeling actions. Shani et al. [31] argue that it is more appropriate to view the problem of generating recommendations as a sequential optimization problem and, consequently, that MDPs provide a more appropriate model for recommender systems. This parameter is therefore important because it is used to identify what modeling action is most likely to be performed by the modeler after a particular modeling action is performed.

5. **Context-based Recommendation**: Context-based recommendation is another factor based on *context aware recommendation* [32] where the contextual information is taken into consideration for the recommendation process. Examples of contextual recommendations include music recommendations where the mood of the user is the contextual information, and travel guides recommendations where location and weather serve as the contextual information for recommending restaurants and travel locations [32]. In our domain, the context of the modeling action is taken into consideration. We consider that all of the non-deterministic modeling actions fall into any of the three contexts: attribute, connection, and general actions. If any modeling action is performed, then all the modeling actions of the same context as the the previous modeling action will be recommended to the modeler. This parameter is therefore important because it is helps to identify and recommend the modeling actions of a particular context while the modeler is working within that context.

6. **Action Presence Based Recommendation**: This recommendation parameter is based on OCL constraint satisfaction. We introduce this as a recommendation parameter that is specific to the modeling domain. In DSMLs, the modeling actions are recommended to the modeler when the particular underlying non-deterministic constraint of the modeling action must be satisfied. Once this modeling constraint is satisfied, then, it is no longer part of the recommended modeling actions displayed for the particular modeling element selected. If the modeler moves to work with another object of the same meta, then the action that was satisfied first should be given more priority. We therefore take a count of how many times the modeling action was not presented when other modeling actions are performed due to satisfied constraints. This ensures that the order the modeling actions are performed is considered because modeling actions are not pesented once satisfied.

4.2.2 Combining All Recommendation Parameters

Ensemble learning is a predictive model built by integrating multiple models based on classifiers or learning algorithms [33]. It is considered as similar to consulting several experts before making a final decision [34]. Opitz et al. [35] have shown that ensemble learning can be useful to improve the predictive performance. We apply ensemble learning in DSMLs by combining the above recommendation parameters to produce a normalized scalar score between 0 and 1. Lastly, weights are assigned to each recommendation parameters to emphasize the their priority. For example, a recommendation parameter with a higher weight is considered more important than a recommendation parameter with a lower weight.

The equation for calculating the score is shown in Figure 4.2. We assign the weights for various parameters, as shown in Table 4.1, in decreasing order of their priority. We will see the reasoning behind the weight assignment in Section 4.2.3. We do not consider sequence based recommendation in the initial score calculation.

This is because if a sequence is detected then, we would recommend the sequential modeling action first and then, the remaining recommended modeling actions are presented as per their calculated scores.

Table 4.1.
Recommendation Parameters with their associated Weights

#	**Parameter Name**	**Weight**
1	Action Context Based	0.30
2	History Based	0.25
3	Action Presence Based	0.20
4	Context Based	0.15
5	Count Based	0.10

Final Recommendation Score =
(0.1*CountCS))+(0.25*HistoryCS)+(0.3*ActionContextCS)+(0.2*ActionPresenceCS)+(0.15*ContextCS);

Where CS = Calculated Score.

Figure 4.2. Recommendation Score Equation

4.2.3 Recommendation Factors Priority Assignment

Generally, in recommender systems, the weights for such parameters are assigned using some learning technique from available datasets. The weights are learned by using additional machine learning techniques, on some specific dataset and then, updating them as the modeler proceeds. In DSMLs, however, there are no datasets available. Also, each DSML is different. The data needed to learn the recommendation parameters must be large. This is because larger datasets can help in improving performance of recommendations [36].

Also, larger datasets can be relatively comprehensive and help find various relatively rare phenomena, or patterns, that may not be discernible in small datasets. [37]. An example can be given of AdaBoost algorithm developed by Freund and Schapire [38–41], which manipulates training examples to generate multiple hypotheses. The learning algorithm is invoked to minimize the weighted error on the training set, which returns a hypothesis whose weighted error is computed and applied to update the weights on the training examples [42]. Another example can be the *bigchaos* solution to the Netflix grand prize [43], which makes use of trainable weights in ensemble learning where the weights are updated for each of the individual recommendation parameters.

It is essential to assign the weights for each of the individual recommendation parameters considered for our recommender system. We have assigned weights using our intuition as of now. When training datasets are available, the weights can be updated accordingly for each domain after learning from the datasets. In the next chapter, we will see the implementation of the recommender system.

5 IMPLEMENTATION

This chapter discusses the implementation of the recommender system in domain specific modeling tools. We have implemented the recommender system for GME by integrating with the Proactive Modeling Engine (PME) in the GAME project, which is described in detail in Appendix B.

5.1 Features of the Recommender System

The recommender system is triggered upon various object select events. For example, when the modeler selects an object in the GME workspace, the *Recommendations Dialog Box* is shown to the modeler. The various functionalities of the recommender system are explained in detail. Also, refer Appendix C for more details.

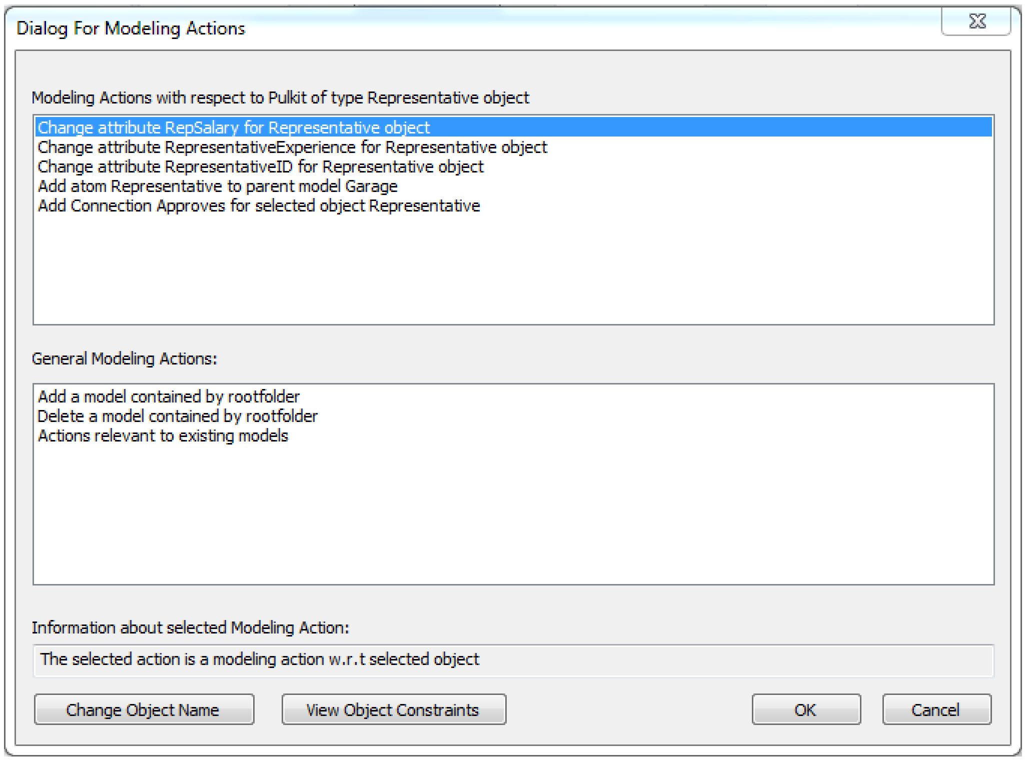
5.1.1 Recommendations Dialog Box

This is the dialog box that is displayed when an object is selected by the modeler. An object can be an Atom, Reference or Model. We use the CRS garage example from Figure 3.15 to select the Representative *Pulkit* and it displays the Recommendations Dialog Box as shown in Figure 5.1.

The Dialog Box has the following parts:

1. **Actions associated with selected Object**: These are the actions which are directly associated with the object selected. They are shown in the first selection box. These actions are divided into the following 3 types:

   (a) **Attribute Actions**: These are the modeling actions pertaining to the attributes of the selected modeling element. The attributes of Representative


Dialog For Modeling Actions

Modeling Actions with respect to Pulkit of type Representative object

Change attribute RepSalary for Representative object
Change attribute RepresentativeExperience for Representative object
Change attribute RepresentativeID for Representative object
Add atom Representative to parent model Garage
Add Connection Approves for selected object Representative

General Modeling Actions:

Add a model contained by rootfolder
Delete a model contained by rootfolder
Actions relevant to existing models

Information about selected Modeling Action:

The selected action is a modeling action w.r.t selected object

Figure 5.1. Recommendation Dialog Box to display Modeling Actions.

object like representative salary, representative experience and representative id are shown as modeling actions to be changed, *i.e.*, completed.

(b) **Association Actions**: These are the modeling actions pertaining to the connection where the selected element acts as the *source* of the connection. Here, the selected object is a Representative, as shown in in Figure 3.1, that is the source of the connection Approves. The modeling action for adding a connection Approves for selected Representative is therefore presented to the modeler.

(c) **General Actions**: These are the modeling actions that have been performed from the general actions section of the `Recommendation Dialog Box` for the meta of the element. This section of the `Recommendation Dialog Box` acts as model guidance for the model. The performed modeling actions are saved and recommended to the modeler for future selections of the object of the particular meta. For example, we select Representative *Akshay* and perform the addition of another Representative object to the sample Garage model through the general actions. Then, upon selection of Representative *Pulkit*, we would see "Add Atom Representative to parent Garage" modeling action along with the other modeling actions in the first selection box as shown in Figure 5.1.

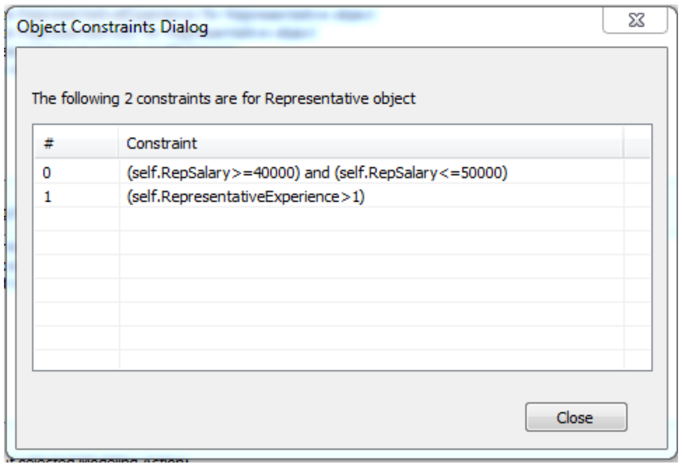
An important point to note that if a general action constraint is satisfied, it still will be recommended to the modeler. So, if the Garage model can have only from 2 to 5 Representatives then the general modeling action "Delete atom Representative from parent model Garage" will still be shown even if the Garage has exactly 2 Representative objects. This is because if the delete modeling action is performed, then the containment check handler automatically adds another Representative object. The modeler can just delete a Representative and another one is automatically added without modeler intervention as the system detects the deterministic containment constraint failure.

Similarly, if the Garage model contains 5 Representatives, then also, the general modeling action of "Add atom Representative to parent model Garage" will be shown. If the modeler performs the add modeling action, then the containment check handler automatically shows the dialog box for selecting an existing Representative for deletion. This mechanism has been retained as it would be easier for the modeler to just add another Representative rather than delete and then add in two different steps. This would reduce the modeling effort.

2. **General Actions associated with any Object**: These are the modeling actions associated with any of the elements present in the model. This is similar to the model guidance handler [14] where the modeler has to select the General actions, such as, "Add an Atom or Model", "Delete an Atom or Model", "Add

a Reference", "Delete a Reference", and "Create a Connection". As seen in previous explanation, any general action selected by the modeler will be present in the selection of the object of the same meta in the modeling actions with respect to the selected object section.

3. **View Object Constraints**: This is used to view the constraints on a particular selected object. If we click on the Representative *Pulkit* we can see the constraints on that object as shown in Figure 5.2. The modeler can view the selected object constraints. It can be used to check the constraints on a particular object, if needed by the modeler.

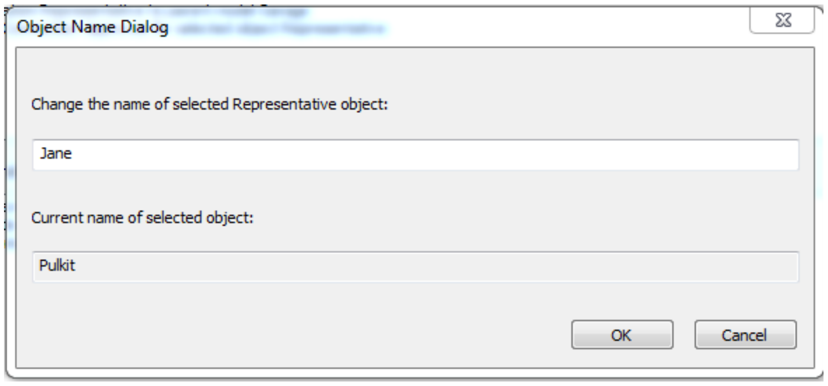


The following 2 constraints are for Representative object

#	Constraint
0	(self.RepSalary>=40000) and (self.RepSalary<=50000)
1	(self.RepresentativeExperience>1)

Figure 5.2. View Object Constraints.

4. **Change Object Name**: This is used to give an object a unique name as each of the objects have a system generated name. During the modeling process, the modeler needs to assign a unique name to every object created. PME creates a generic name for the objects which need to be changed by the modeler. This is to ensure that the recommendations are performed accurately as each object created must be unique. If we want to change the Representative name from *Pulkit* to *Jane* we click on the `Change Object Name` button and change the name as shown in Figure 5.3.


Object Name Dialog

Change the name of selected Representative object:

Jane

Current name of selected object:

Pulkit

Figure 5.3. Change Object Name.

5.2 Working of the Recommender System

In this section, we describe how recommendations work based on the modeler factors described in Section 4.2.1. An important point to note here is that, these recommendations are specific to each meta, so if we do operations with alternate metas, the recommendation information will be stored and recommended with respect to the meta of the object selected.

5.2.1 Recommendation Parameters

1. **Sequence Based Recommendation**: The sequence-based recommendation, as we have seen in Section 4.2.1 in sequence based recommendation, can be illustrated with the following example. If the modeler adds 3 Representatives in the sample Garage model, and if they complete for *Representative1* the modeling actions of setting the representative id, representative salary and then representative experience, for *Representative2* again representative id, representative salary and then representative experience, then the sequence based recommendation is triggered.

   So, when *Representative3* is selected, the *change representative id* modeling action is recommended first to the modeler. If the modeler selects and completes *change representative id* modeling action, then representative salary option is presented first which, on completion, the representative experience is recommended. So, if these 3 actions are completed in that order for *Representative3* then when we create and select *Representative4* object these modeling actions will continue to be recommended in sequence to the modeler. An illustration is shown in Figure 5.4.

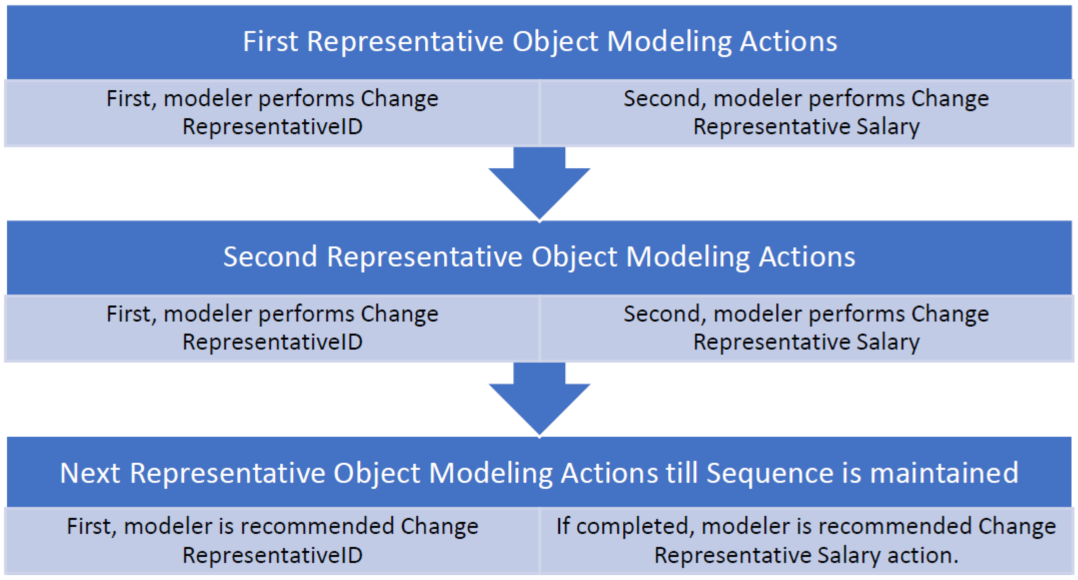

First Representative Object Modeling Actions
First, modeler performs Change RepresentativeID
Second, modeler performs Change Representative Salary
Second Representative Object Modeling Actions
First, modeler performs Change RepresentativeID
Second, modeler performs Change Representative Salary
Next Representative Object Modeling Actions till Sequence is maintained
First, modeler is recommended Change RepresentativeID
If completed, modeler is recommended Change Representative Salary action.

Figure 5.4. Sequence Recommendation Demonstration.

2. **Count Based Recommendation**: In count based recommendation, the modeling actions associated with a particular meta are recommended based on the frequency of the performance of the modeling actions. For example, if for 5 Representatives the *change representative id* modeling action was performed, and for 3 Representatives the *change representative salary* modeling action was performed, and for 1 Representative the *change representative experience* modeling action was performed. Then, when we select another Representative object, first the *change representative id*, then *change representative salary*, and finally, the *change representative experience* modeling actions are presented to the modeler.

Figure 5.5 shows the calculation equation of the *count score* for count based recommendation. For each modeling action, we take the number of times it has been performed and divide it by the total number of modeling actions performed with respect to the particular meta.

$$\text{Count Score} = \frac{\text{Number of Times Action Performed}}{\text{Total Number of Times Actions Performed}}$$

Figure 5.5. Count Score Calculation.

3. **History Based Recommendation**: In history based recommendation, the instance in terms of when the modeling action was selected is considered with respect to objects of a particular meta. We think of time in terms of modeling actions.

   So, if 10 modeling actions were performed, and if *change representative id* was action number 3 performed on *Representative1* and *change representative salary* was action number 8 performed on *Representative1* then it is considered that *change representative salary* action was performed in more recent past as compared to *change representative id* modeling action. However, since modeling actions can repeat for objects of the same meta, these most recent one is considered and the past ones are discarded.

   So, as in the previous example, if action number 10 was *change representative id* performed on *Representative2*, then *change representative id* action was performed in more recent past as the older action number 3 of *change representative id* is discarded. So, with respect to, action number 8 of *change representative salary*, the *change representative id* modeling action is considered to be a more recent one. Figure 5.6 shows the calculation equation of the *history score* for history based recommendation. For each modeling action, we take the latest number it has been performed, and then, divide it by the summation of the latest number for all the modeling actions performed with respect to the particular meta.

$$\text{History Score} = \frac{\text{Latest Performed Action Number}}{\sum \text{All Actions Latest Performed Action Numbers}}$$

Figure 5.6. History Score Calculation.

4. **Action Context Based Recommendation**: In action context based recommendation, the action which is selected after a particular action is considered. It is based on the context of other actions. It is also considered with respect to objects of a particular meta.

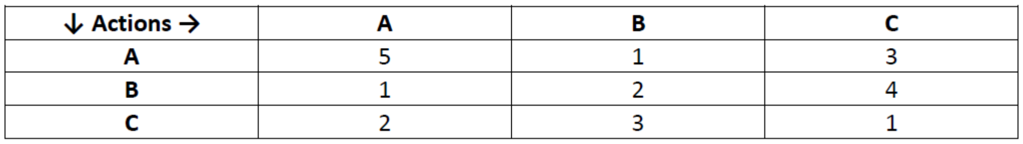
   For example, if for *Representative1*, the modeler selects *change representative id* action and then either for *Representative1* or *Representative2*, the selection of *change representative salary* action is made, then whenever *change representative id* modeling action is performed for any Representative object then *change representative salary* modeling action is recommended. We perform this by maintaining with respect to each modeling action, the number of times all the other modeling actions are selected.

   Figure 5.7 shows the calculation equation of the *Action Context Score* for action context based recommendation. For each modeling action, we take how many times after a particular modeling action, it has been performed, and then, divide it by the summation of the count for all the modeling actions performed after the particular modeling action with respect to a particular meta.

$$\text{Action Context Score} = \frac{\text{Action Context Count after particular Action}}{\sum \text{Action Context Count for all Actions after particular Action}}$$

Figure 5.7. Action Context Score Calculation.

An example table is shown in Table 5.8. Suppose we have 3 modeling actions A,B and C, with respect to a particular Representative object. Then after action B, action A has been performed 1 time, action B has been performed 2 times and action C has been performed 4 times. In order to obtain *action context score* for action C, we divide 4, which is the number of times it was selected after action B, by 7 which represents the summation of counts for all other actions performed after action B.



↓ Actions →	**A**	**B**	**C**
A	5	1	3
B	1	2	4
C	2	3	1

Figure 5.8. Action Context Count Table.

5. **Context Based Recommendation**: In context based recommendation, we consider 3 contexts: attribute actions, connection actions and general actions. Attribute actions are the ones which are related to changing the attributes of the selected object. For example, for any Representative object of type Representative meta, the 3 actions of *change representative salary*, *change representative id* and *change representative experience* are the attribute modeling actions. Connection actions are the actions in which the selected object acts as a source for a particular connection which is recommended to the modeler. For example, if the Representative object is selected, the *add connection approves* is a connection action. General actions are the actions which are the ones mentioned in the `Other Modeling Actions Selection Box`.

   For example, if we select *Representative1*, and using general actions, add another Representative object, then when we select any Representative object, it will be shown as a recommended action, *i.e.*, we will see an action to add Representative object to *Garage* model. This action will be treated as a general action. So, as shown in Figure 5.1, if we perform *change representative id* for selected

Representative object, then, when we select the same or any other Representative object, all the attribute actions would be given a higher score, and therefore, should get recommended towards the top for the modeler to perform. This is useful assuming modelers tend to work with one context at a time like working on all attributes of an object first, and then, adding connections.

6. **Action Presence Based Recommendation**: Since modeling actions are shown to the modeler due to constraints not being satisfied, once the modeling actions are completely satisfied as per the constraints specified, the modeling action no longer appears in the `Recommended Actions Selection Box`.

   For example, if for *Representative1*, the modeling action *change representative experience* is performed and the representative experience constraint is satisfied by specifying the representative experience as 2, if the modeler selects *Representative1* again *change representative experience* action will not appear again. However, if the modeler specifies the experience value as 1 which does not satisfy the constraint then upon selection of *Representative1*, the action is recommended again. Assuming we input the value correctly and satisfy the underlying constraint, when we perform other actions for *Representative1* object, a count is maintained to check how many times that modeling action has not come up. This is how action presence based recommendation is implemented.

   Figure 5.9 shows the calculation equation of the *action presence score* for action presence based recommendation. For each modeling action, we divide the *action presence count* (count how many times it does not come up) by the summation of all the action presence counts for all actions with respect to particular meta.

$$\text{Action Presence Score} = \frac{\text{Action Presence Count}}{\sum \text{Action Presence Count for all Actions}}$$

Figure 5.9. Action Presence Score Calculation.

5.2.2 Combining all the Recommendation Parameters

As we have seen in Section 4.2.2, we have combined all the recommendation parameters using weights to determine the order of recommendations. Also, if the first action is based on sequence based recommendation, then that action will be displayed first followed by the rest of the recommended actions. If the sequence based recommendation and the first recommended action is both the same then the score for the sequence based recommendation will be still calculated. However, it will just be shown as sequence based recommendation. Also, the recommendation scores are with respect to the context of a particular meta. This is to maintain context aware recommendation, so that, the actions performed will be added to the list of that meta.

If a particular meta is selected for the first time, and, there are no recommendation scores available yet, then the default setting is that the attribute actions are recommended first to the modeler. This is also done by intuition, as there is no existing data available, and we assume the modeler proceeds to work with the attributes of an object first. We just activate the context based recommendation with respect to attributes for a particular object. However, there is no specific order to the modeling actions recommended, in the attribute modeling actions, as again, there is no additional data available. If modeling actions have the same score, then they are presented in alphabetically sorted order.

5.3 Retrieval of Recommendations

The recommendation information is stored in a database. We have used *ADBC framework* [44] which uses *SQLite3* [45] to facilitate the necessary database CRUD (create, read, update and delete) operations. The modeler activity information is stored for each project of a particular DSML, the modeler creates. This helps in reusing the recommendation information from the previous projects of a particular DSML. In short, the recommendation information for the new project is based on the previous projects for a particular DSML. If there are multiple projects created for a DSML, the recommendation scores are aggregated and then presented to the modeler. This is done for each modeling action and the number of times it occurs in the projects. Database storage of recommendation information is persistent storage which can be retrieved later by the recommender system which is useful if the modeler saves and closes the project. So, the modeler can continue working by reopening the project as per their convenience and the recommendation information of the previous activity of the modeler is retrieved by the recommender system.

As storing to a database is a file Input/Output (I/O) operation, it is slow as disk operations are slow [45]. To overcome this drawback, we have used an LRU cache mechanism [46], which aims to overcome this drawback. In LRU cache, the recommendation information is tied to each meta which is stored. So, upon selection of common metas within the specified limit, the cache stores all of the information. However, if we make a selection of a new meta which is more than the cache limit, the *least used* object is written to the database and the recommendation information for the new meta is loaded from the database. This is similar to LRU page replacement policy algorithm [46], if the cache limit is exceeded. In the end, while closing the model, the information from the cache will be written to the database. This ensures that the system does not cause any unnecessary delays while the modeler is performing the modeling actions.

In the next chapter, an evaluation of our system is performed.

6 RESULTS

In this chapter, we will use describe the approaches to evaluate our recommender system. *Average Reciprocal Hit-Rank (ARHR)* [29] is the standard approach for evaluating our system. *System Usability Scale (SUS)* [47] is another way to evaluate the usability of our system. ARHR analysis is performed by us on our system based on modeler activity whereas in SUS the modelers based on their experience rate or evaluate our system for completing the given tasks.

6.1 System Evaluation Study

In order to evaluate our system, we conducted a study where 5 modelers of varying levels of GME modeling expertise were given tasks to perform using our system. The modelers had to identify their level of *expertise* on a scale of 1 to 5. Expertise level determines how familiar the modeler is with using GME.

So, a level 1 modeler has never used GME is a *novice* modeler, a level 3 modeler considered as an *average* modeler has used the software occasionally and the level 5 modeler is considered an *expert* modeler. There was one modeler for each level. The modelers were given modeling exercises on CRS and LMS paradigms. The modelers were provided with two tutorials to install and understand the features of the recommender system:

1. The modelers had to install the GAME Microsoft Installer (MSI) by referring to `https://github.iu.edu/SEDS/GAME/wiki/GAME-for-GME-Installation`, which provides the tutorial for installation of the `Model Intelligence` add-on for GME.

2. The modelers had to learn the different features of the recommender system by referring to `https://github.iu.edu/SEDS/GAME/wiki/Tutorial-on-Working-with-GAME-Model-Intelligence`, which provides the tutorial for understanding the various features of the recommender system. The tutorial does not indicate to the modelers that our system recommends modeling actions based on modeling activity.

The details of the software and the system installation as well as usage is given in Appendix C. The tasks were as follows:

1. **For CRS**: The modelers had to create and complete a model with certain conditions towards completion. This activity was intended to check how the modelers perform with simple modeling actions.

2. **For LMS**: The modelers had to create 3 models which had more modeling elements as compared to CRS. This was to evaluate how the system behaves if more number of modeling actions are to be analyzed. Also, the modelers were asked again to complete two more models in a `New Project` based on LMS. This activity was used to evaluate for scalability and the ability of the system to consider the recommendations from the previous model applied in these models.

The modeling exercises are described in detail in the Appendix E. The modelers were not informed that the system has recommendations integrated into it. We have one modeler for each expert level.

6.2 Number of Modeling Actions performed towards Model Completion

The modelers have completed a certain number of modeling actions towards model completion as per the modeling exercises given on CRS and LMS scenarios. These modeling actions are the non-deterministic actions that required modeler intervention.

6.2.1 Garage Paradigm

For the Garage paradigm, we can see in Table 6.1, the total number of modeling actions performed by each modeler of each expert level towards model completion. As per Table 6.1, the average for each modeler is 140 modeling actions towards model completion.

Table 6.1.
Number of Modeling Actions performed towards Model Completion in CRS Paradigm

Modeler Expert Level	**Number of Modeling Actions**
1	161
2	143
3	156
4	102
5	138
Total	700

6.2.2 Library Paradigm

For the Library paradigm, we can see in Table 6.2, the total number of modeling actions performed by each modeler of each expert level towards model completion. As per Table 6.2, the average for each modeler is 342 modeling actions towards model completion.

6.2.3 Considering Garage and Library Paradigms

If we consider both the CRS and LMS exercises, then each modeler on an average had to perform 482 modeling actions (the summation of average modeling actions in

Table 6.2.
Number of Modeling Actions performed towards Model Completion in LMS Paradigm

Modeler Expert Level	**Number of Modeling Actions**
1	355
2	335
3	351
4	343
5	325
Total	1709

CRS and LMS scenario for each modeler) towards both model completion exercises. The total number of modeling actions performed by all 5 modelers is 2409 modeling actions.

6.3 Average Reciprocal Hit-Rank (ARHR)

First, we only consider, if the modeling action is selected from the modeling actions with respect to a particular meta which is displayed is the *first selection dialog box*. These are the modeling actions which are recommended to the modeler. If a modeler selects a modeling action from these recommended modeling actions then it is considered as a *hit*. We use average reciprocal hit-rank measure that rewards each hit based on where it occurred in the *top-N list* [29].

So, if a modeling action is performed which was at position 1, it will increase the ARHR whereas if it was performed from the last position, it will not be as significant as position 1. Also, if an action is performed from the general modeling actions, the position is not considered for the ARHR but the action is considered as part of the ones performed by the modeler. We calculate the ARHR [48] as shown in Figure 6.1. We

$$\text{ARHR} = \frac{1}{\text{Total Modeler Actions performed}} \sum_{i=1}^{\#\text{hits}} \frac{1}{\text{Position of selected Modeling Action}}$$

Figure 6.1. ARHR Score Calculation.

first take the summation of the reciprocal of the positions at which the hits occurred. Then we divide this result with the number of modeling actions the particular modeler completed.

This is done as the recommended modeling actions are variable, *i.e.*, it can reduce if particular constraints of an object are satisfied as we have seen in action presence as shown in Section 5.2.1 or they can increase if the modeler performs modeling actions from the general modeling actions as shown in Section 5.1.1. Finally, we take an average of all the ARHR by dividing them by the number of modelers. Thus, we obtain the final ARHR which is taken for each metamodel paradigm.

6.3.1 Results and Analysis

We consider the ARHR for both the Garage as well as Library paradigms separately.

Garage Paradigm

1. **ARHR** As shown in Table 6.3, we have calculated the ARHR for each of the modelers. In the end, we have also given an average of the ARHR for the CRS paradigm model.

2. **Analysis of High ARHR** We will see a detailed analysis of how the ARHR is so high.

   Total Number of Recommended Actions In Figure 6.2, we can see how many modeling actions were presented to all the modelers. However, it is important to note that depending upon the general action selection of the

Table 6.3.
ARHR for the CRS Model completed by Modelers

Modeler Expert Level	**ARHR**
1	0.626708
2	0.868298
3	0.927564
4	0.869374
5	0.916667
Average	0.8417222

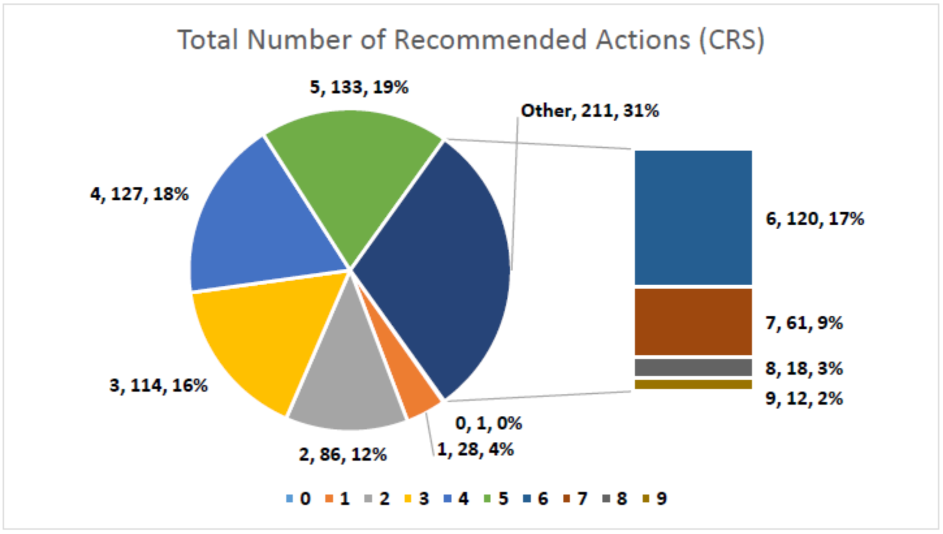


Figure 6.2. Total Recommended Actions (CRS).

modelers, the recommended modeling actions will increase for particular metas accordingly. Also, we can see that there is a scenario, in which, 0 modeling actions were recommended once to a modeler.

This means that the modeler has selected a general action as no recommended modeling actions were presented, as the modeler satisfied all constraints. We can also see that, 68% of the modeling actions were 4 or more and that 50% of the modeling actions were 5 or above.

Modeling Action Selection Position In Figure 6.3, we can see that 77% of the modeling actions selected by the modelers are at position 1 and 85% of the modeling actions selected are at positions 1 and 2. This has resulted in a high

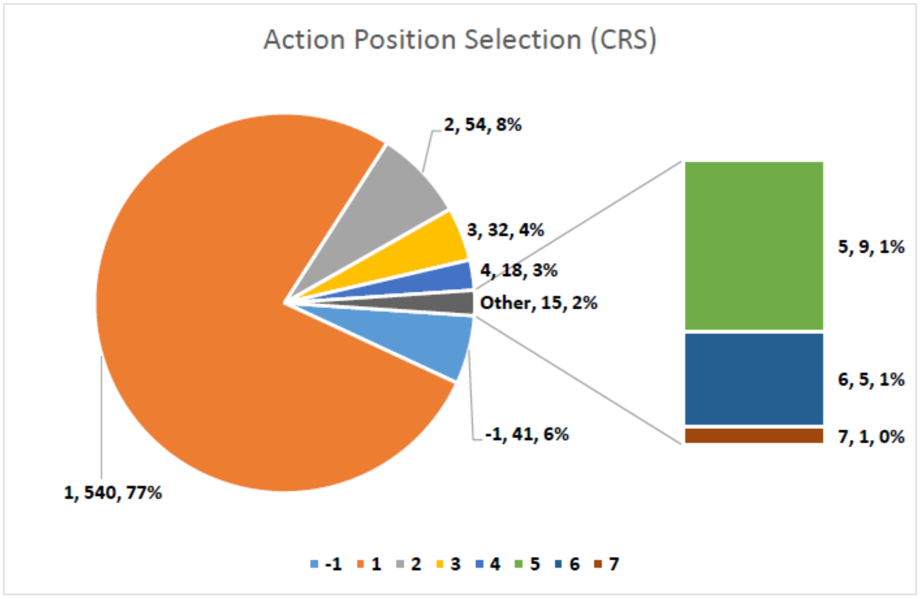


Figure 6.3. Action Position Selection (CRS).

ARHR score. However, 6% of the modeling actions come from general actions which are considered for the total number of modeling actions but their position value is discarded. This has reduced the ARHR slightly.

First Position Action Selection As shown in Figure 6.4, we can see that 69% of the modeling actions selected at position 1 are from 4 or more recommended modeling actions which increases to 84% if 3 recommended modeling actions are also considered. Only 4% of the modeling actions are where only one recommended modeling action was presented to the modelers.

This shows that the modelers were not given only one modeling action to chose from and it has only a minute impact of the high ARHR. Obviously, when 0 or no modeling action was presented, no selection was made at position 1.

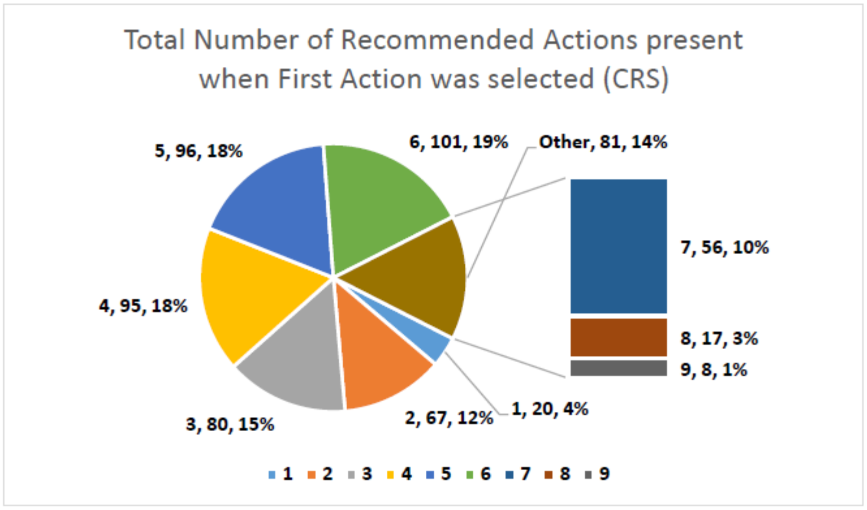


Figure 6.4. First Action Selection (CRS).

Second Position Action Selection As shown in Figure 6.5, we can see that 48% of the time a modeling action was selected at position 2, the actions presented to the modelers were 4 or higher. If we add 3 modeling actions presented to the modelers then it increases to 80%. Only at 20% were 2 modeling actions presented to the modelers and selection made at position 2.

This shows that majority of the modeling actions presented to the modelers for selection at position 2, is more than 2. Obviously, if only 0 or 1 modeling action was presented to the modelers, there cannot be a selection at position 2.

Library Paradigm

1. **ARHR** As shown in Table 6.4, we have calculated the ARHR for the modelers for the library paradigm modeling exercise. In the end, we have also given the average ARHR.

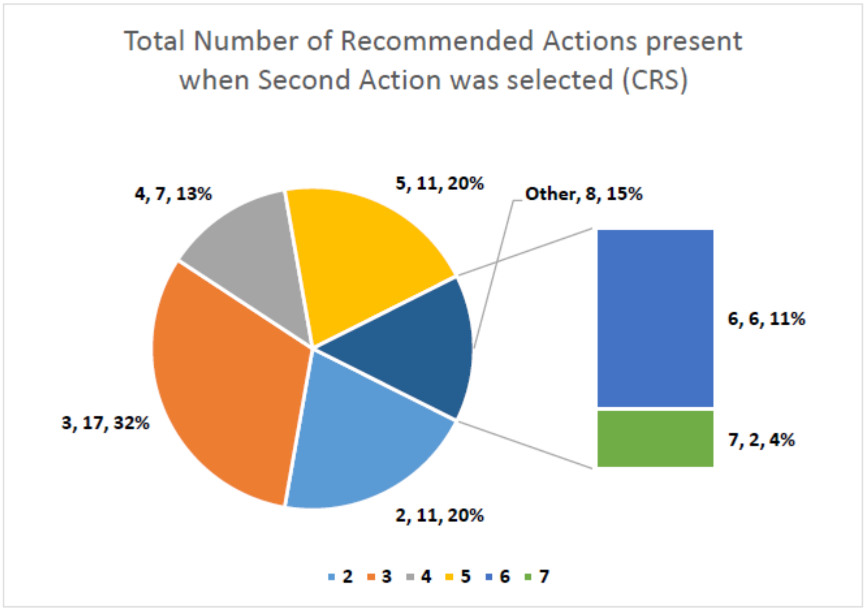


Figure 6.5. Second Action Selection (CRS).

Table 6.4.
ARHR for the LMS Model completed by Modelers

Modeler Expert Level	**ARHR**
1	0.819531
2	0.885174
3	0.937749
4	0.936589
5	0.919179
Average	0.8996444

2. **Analysis of High ARHR** As in the Garage paradigm, we will see an analysis of how ARHR is so high.

   Total Number of Recommended Actions Similar to the CRS, in Figure 6.6 we can see that total number of recommended modeling actions to the modelers which is 4 or more is 63% and if we include even 3 recommended modeling actions it becomes 82%.

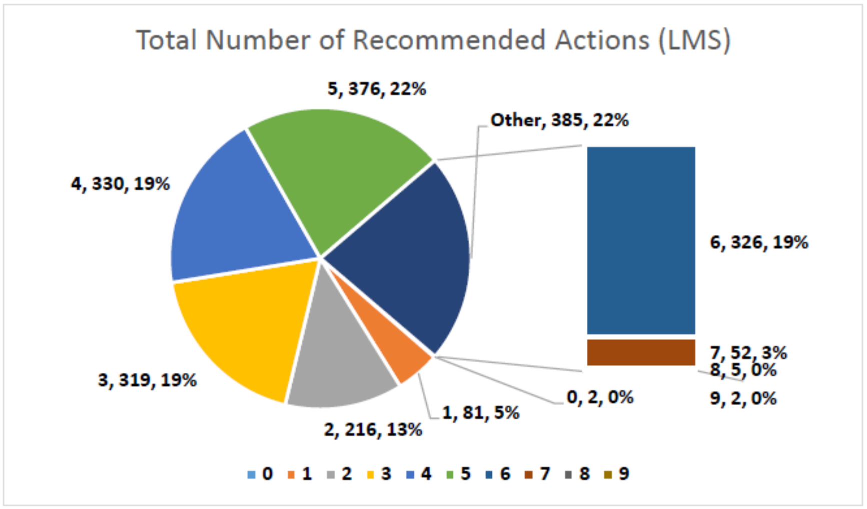


Figure 6.6. Total Recommended Actions (LMS).

There are only 18% of actions which are 2 or fewer out of which only 5% is one modeling action. Here also, 0 actions have been recommended 2 times to modelers. As in CRS, modelers have chosen an action from the general modeling list for that scenario.

Modeling Action Selection Position As shown in Figure 6.7, we can see that 85% of the modeling actions were selected at position 1 and including position 2 it becomes 92%.

The rest form only 8% out of which the selection from general modeling actions is only 3%. Therefore, due to these factors, the ARHR is quite high, as most modelers have selected at position 1.

First Position Action Selection In Figure 6.8, we can see that for position 1 selection 81% of the modeling actions are 3 or more.

Only a marginal 5% of the modeling actions were, where only 1 modeling action was recommended to the modeler. So, it can be inferred that the modelers were

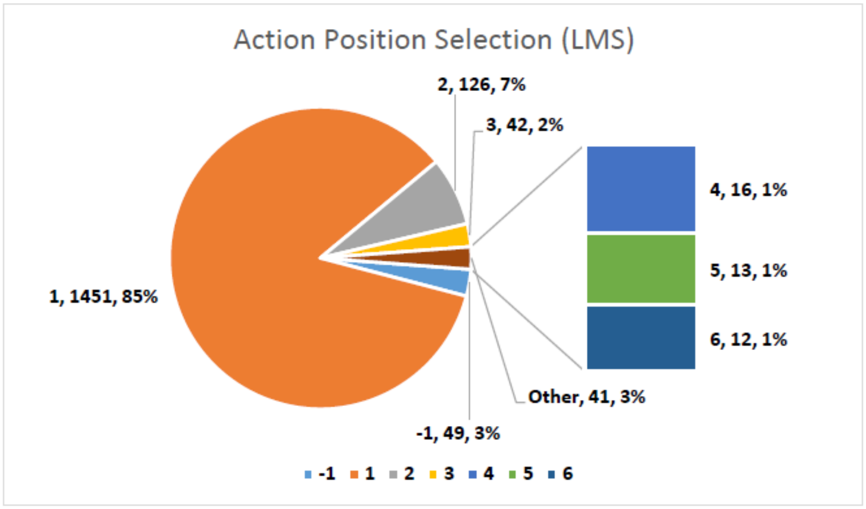


Figure 6.7. Action Position Selection (LMS).

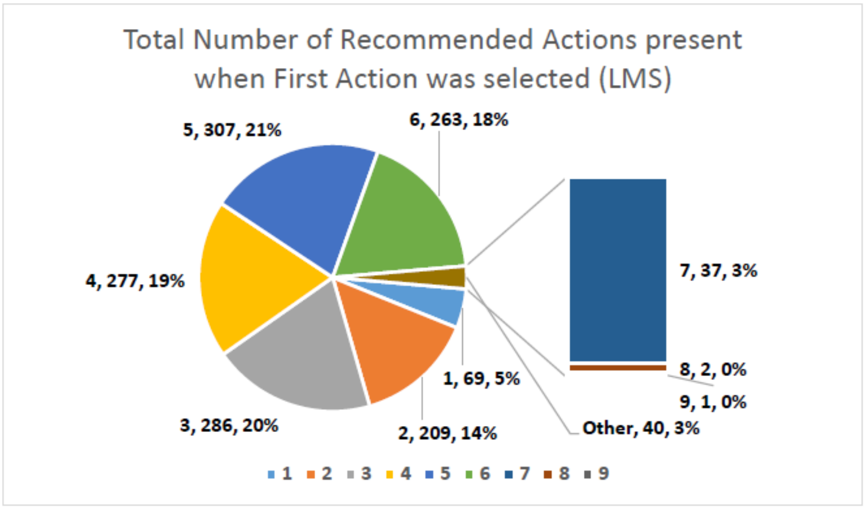


Figure 6.8. First Action Selection (LMS).

given almost always more than 1 recommended modeling action to chose from while they were choosing the first one.

Second Position Action Selection As shown in Figure 6.9, for position 2 selection, 79% of the modeling actions were from 4 or more modeling actions and if we consider 3 modeling actions it becomes 97%.

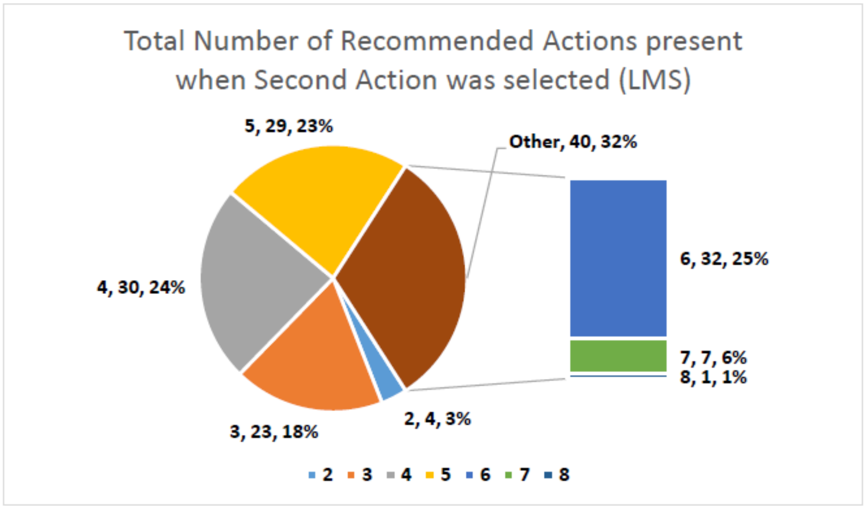


Figure 6.9. Second Action Selection (LMS).

Only a marginal 3% of the selections were from 2 modeling actions. This also shows, as from the previous observations, that the modelers almost always were given more than 2 modeling actions to choose from.

6.3.2 Analysis of ARHR Results

As we can see from both the CRS and LMS examples, the ARHR is high in both cases. We have also seen that the modelers have always selected the recommended modeling action at position 1 from the available varying list of recommended modeling actions. These modeling actions vary depending upon the modeler satisfying constraints and selection of modeling actions from general actions. In CRS scenario, we can observe that the modeling actions were fewer as the modelers had to create only one Garage model. So, the selection of actions at position 1 was only 77%. Whereas, in the case of LMS scenario, the modelers had to create 1 Library model and then again in a different project 2 more Library models.

This resulted in an increase in the selection of modeling actions at position 1. We can see that all of the recommendation parameters have worked in favor for the modelers and the recommendations were presented in an effective manner. The recommender system was successful in recognizing which action the modelers wanted to do next and has recommended appropriately. Otherwise, the modelers would not have selected the first action so many times.

6.4 System Usability Scale (SUS)

The modelers who have completed the specified tasks were given the SUS survey to evaluate the system. The SUS survey was created by John Brooke which was used to measure the effectiveness, efficiency and satisfaction of a particular system which can be software [47]. It can also be used to measure *usability* and *learnability* [49, 50]. It can also be used reliably on a small sample size (say 8-12) of users and be fairly confident that a good assessment of how people see the system [50, 51].

The modelers are given the SUS which is a 10 question survey having the odd questions are positively worded and the even numbered item including 0 are negatively worded. The modelers are asked to rate from *strongly disagree* to *strongly agree*. For SUS calculation, for odd items, the score contribution is the scale position minus 1. For even items, the contribution is 5 minus the scale position. We then, multiply the sum of the scores by 2.5, to obtain the overall value of SU [47].

6.4.1 Results

As shown in Table 6.5, we have calculated the SUS rating for all the modelers. We have also taken an average of all the SUS Ratings and displayed in the end.

6.4.2 Analysis of Results

As we can see, in the percentile ranking, our system fairs *Good* [50] and our system is scores in a traditional school grade setting C [52] or a B grade [49]. The SUS rating

Table 6.5.
SUS Rating Calculated based on Modeler Responses

Modeler Expert Level	**Calculated SUS Rating**
1	67.5
2	72.5
3	87.5
4	85
5	72.5
Average	77

is valid and reliable, however, not diagnostic and it does not tell us what makes a system usable or not [49, 50].

SUS ratings have a modest correlation with task performance, but it is not surprising that people's subjective assessments may not be consistent with whether or not they were successful using a system as they are only one component of the overall construct of usability [50].

6.5 User Feedback

As we have seen in SUS, it does not tell us what makes a system usable and neither does it point out ways to improve the same. To overcome this drawback, we have also given a feedback form to the modelers, which contains 3 questions and the modelers give subjective replies. The questions are as follows:

1. **Question 1**: *Any Benefits/Advantages of Using this System over the Current one?* This helps us understand the various positive aspects of our system from the modelers. The modelers have to describe the benefits of using the recommendation system over the existing one which does not provide any auto-completion or recommendations.

2. **Question 2**: *Any Criticisms/Shortfalls/Disadvantages of Using this System over the Current one?* This helps us understand the various negative aspects of our system from the modelers. The modelers have to describe any shortcomings of using the recommendation system over the existing one which does not provide any auto-completion or recommendations.

3. **Question 3**: *Feedback/Suggestions:* This is a general feedback section in which the modelers can give any suggestions which may potentially improve the system. The modelers may just describe subjectively any features they desire in the system in a non technical language.

6.5.1 User Excerpts and their Analysis from the Feedback Form

Here, we present some excerpts from modelers as given in their feedback form. We classify these into the responses for the 3 questions as follows:

1. **Question 1 : Advantages** Here, we discuss in general the advantages from using our system over modeling without it.

    - **Useful for Novice Modelers**: Modeler Level 1 wrote *The system is very easy to learn even for a novice user.* Modeler Level 5 wrote *The GUI interface for GAME made GME feel more user friendly for people with little programming background.*

      This shows the modelers feel that the system along with the Graphical User Interface (GUI) would be great for novice modelers.

    - **Easy to use**: Modeler Level 2 wrote *Ease of Use. Intuitive and easy to understand interface.* Modeler Level 3 worte *The system was very adaptive to my usage. I felt using the system got easier the more I used it.* Modeler Level 4 wrote *It is easy to use and it is useful to display all the functionality in one dialog box.*

This shows that the system is useful, easy to use and the modelers are comfortable with it.

- **Reduce Modeling Time**: Modeler Level 5 wrote *Having the system remember past selections/actions is great. I think this can potentially decrease the time it takes to construct a Model.* Modeler Level 1 wrote *It is a really useful tool and it is very fast too.* Modeler Level 3 wrote *It prioritized suggestions based on my most recent actions and this saved a lot of time for me.*

  This shows that the modelers felt that the their modeling time and effort is reduced somewhat using this system as it has personalized their modeling actions by analyzing their modeling activity.

- **Simplify Modeling Actions**: Modeler Level 2 wrote *Adding new atoms and listing down their properties is simple.*

  This shows that the modeling actions are simplified since they are presented to the modeler.

- **Modeling paradigm independent**: Modeler Level 1 wrote *I found it useful since you can model any scenario like a parking system or restaurant system.*

  This shows that the modelers can use our system independent of any modeling paradigm.

2. **Question 2 : Disadvantages** Here, we discuss in general the advantages from using our system over modeling without it.

   - **Repetitive Modeling Actions should be simplified**: Modeler Level 1 wrote *The only disadvantage I felt was the cumbersome process of creating objects every time. So, if there are 100 objects you have to click 100 times and create those which can be problematic for huge systems.* Modeler Level 5 wrote *The GUI interface was nice but it required repetitive actions/button-*

clicks even when using previously selected options. This made the process of creating a model still feel tedious.

The modelers have found it difficult for scaling their modeling actions like adding 10 same objects. For solving the problem of repetitive modeling actions, we can combine MTBD along with our system. Thus, the modelers specify their modeling actions by showing a demonstration and the system can generate the number of objects needed.

- **Initial Addition of Actions**: Modeler Level 2 wrote *Could simplify the process of starting from scratch. Sometimes a little slow for initial addition of actions.*

  The modelers need to select through various dialog boxes they want to perform a general action at first before it gets added to the recommended actions list. This would result in a slow down of the initial modeling process. So, we could as shown in Chapter 7, reuse modeling actions from expert modelers who are trusted and thus will be presented in the recommended actions of other modelers. So, they would not need to go through the entire process of adding specific general actions.

- **Dialog Box pop up on Object Select always**: Modeler Level 5 wrote *It would be nice to have some way to force GAME to not open the dialog box every time I click on an object. Sometimes I just want to move objects around and having to close the box every time also becomes tedious.*

  This means the modeler does not want the dialog box to pop up every time an object is selected. We can change this by keeping a switch or certain hot key combinations which would disable the triggering of the dialog box.

- **Attribute Values have to be entered each time**: Modeler Level 3 wrote *I had to enter in the attribute values for each and every entity. This felt very cumbersome as the model got larger.*

The modeler felt that the attribute values to enter repeatedly became cumbersome as the model size increased. A possible solution is discussed in next Section *Suggestions* for *attributes of objects*.

3. **Question 3 : Suggestions** Here, we discuss in general the suggestions from using our system over modeling without it.

    - **Attributes of Objects**: Modeler Level 1 wrote *One should be able to set attributes through another say attribute menu which should be like form filling process for each object.* Modeler Level 2 wrote *Could allow users to add properties to atoms via the input section.* Modeler Level 3 wrote *May be the system could enter default values in the rest of the entities based on the values that I enter in the initial few entities.*

      This means that the modelers would prefer the attributes to be separate with all the constraint information present. The present GME model does not show the constraint information. So, we might need to come up with an effective way as to show the constraints for the attributes and also guide the modeler accordingly. Also, we could make use of MTBD for filing up default values for one sample object and then repeat it for other objects.

    - **Object Creation**: Modeler Level 1 wrote *I feel for the object creation there can be a user input menu which would ask user to input how many objects one has to create and it would create them instantly.* This is similar to the problem of *repetitive modeling actions should be simplified* from the previous *Disadvantages* section.

      We can to use MTBD for showing a demonstration of how many objects and/or how to add in which paradigm.

    - **Focus on Modeling Action Selection**: Modeler Level 5 wrote *I feel enabling the use of hot keys to accomplish tasks and/or navigate the GAME dialog boxes could make the modeling process smoother. When the GAME dialog boxes appear, perhaps placing the default focus onto the menu options* *instead of the buttons would be nice. This would allow the modeler to use the arrow keys on the keyboard to navigate the menu immediately rather than having to click into the menu first.*

This means that the modeler wants to use special hot keys, on the keyboard, to navigate the modeling actions including the general actions and put focus on the first modeling action. That would be helpful for making the modeling process smoother. This can be easily completed by updating the functionality of the `Recommendations Dialog Box`.

7 CONCLUDING REMARKS

In this thesis, we introduced recommender systems into domain-specific modeling languages with the goal of improving the modeling experience. Based on our experiments, we had a SUS Rating Score of 77, which is classified as *Good.* Based on our experience, we have learned the following lessons:

- **Combining recommender systems with model transformation by demonstration.** Our user feedback results (see Section 6.5.1), highlighted that certain actions that need to be performed repeatedly can be accomplished using one modeling action (*e.g.*, adding 10 objects). Currently under our system, the modeler must perform manually select repetitive modeling actions. We therefore believe this is a scenario where we can integrate model-based demonstrations by transformation into our current approach to further improve user experience.

- **Synthetic dataset generation.** The unavailability of data in DSMLs to support recommender systems research (refer Section 4.2.3) led to assigning weights by intuition for the recommendation factors. A potential solution would be to generate synthetic datasets, which mimic real world datasets [53] for training the weights for recommendation parameters. Item-based top-N recommendation algorithm also used synthetic dataset SDG for evaluating their system [29]. Future work will therefore involve generation of synthetic datasets, specific to each DSML, to train the weights of recommendation parameters subsequently improving the ARHR and the SUS rating.

- **Identifying user modeling expertise levels based on modeling actions** In our current approach, modelers have self identified their expertise level on modeling. This is a problem, as modelers may wrongly identify themselves as expert modelers whereas they may just be intermediate modelers. This can cause errors in data collection, which will get propagated if used in user trust where modeling actions of expert modelers will be recommended and given priority to novice modelers. The recommender system therefore must be able to dynamically identify different modeler levels based on the modeling activity of the modelers.

Lastly, here is a list of future research directions, which we believe will improve the current SUS Rating Score:

- **Using machine learning techniques for dynamically learning weights of recommendation parameters.** As explained in Section 4.2.3, we have assigned weights based on our intuition. Although this is feasible, it can become problematic because we expect that each modeler performs modeling actions differently. When the weights are learned based on individual modelers for every DSML, we expect that it should be representative of the pattern a particular modeler is modeling. So, the weights cannot be the same for all modelers in all DSMLs. Future research, therefore, includes using machine learning techniques to learn weights on an individual modeler basis.

- **Incorporating user trust.** User trust [54], which leverages preferences of communities of similar users, has been shown to improve the predictive accuracy of recommender systems [54, 55]. Future work therefore includes incorporating user trust into our recommendation system for DSMLs. This will allow novice modelers to trust modeling actions of expert modelers, which will also help decrease learning time for novice modelers.

- **Recommender systems as an educational tool for DSMLs.** Recommender systems enable people to share their opinions and benefit from the experience of one another [56]. Likewise, recommender systems has been implemented in educational tools [57, 58]. Future research therefore can explore as to how our recommendation system for DSMLs can facilitate the education of novice users on a given DSML. This will help in identifying and suggesting more novel solutions to improve the existing system other than the points discussed in this chapter.

All work from this thesis has been integrated into GAME. It is freely available in open-source format from the following location: `github.com/SEDS/GAME`.

REFERENCES

REFERENCES

[1] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.

[2] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The Generic Modeling Environment, 2001.

[3] Jules White, Douglas C. Schmidt, and Sean Mulligan. The generic eclipse modeling system. In *Model-Driven Development Tool Implementers Forum, TOOLS*, volume 7, 2007.

[4] Hans Vangheluwe, Ximeng Sun, and Eric Bodden. Domain-Specific Modelling with $AToM^3$. In *In Proceedings of the th OOPSLA Workshop on Domain-Specific Modeling*, 2004.

[5] Steve Cook, Gareth Jones, Stuart Kent, and Alan Wills. *Domain-specific development with visual studio dsl tools.* Addison-Wesley Professional, first edition, 2007.

[6] Jules White, Douglas C. Schmidt, Andrey Nechypurenko, and Egon Wuchner. Domain-Specific Intelligence Frameworks for Assisting Modelers in Combinatorically Challenging Domains. *GPCE4QoS (October 2006)*, 2006.

[7] Anders Hessellund, Krzysztof Czarnecki, and Andrzej Wąsowski. Guided development with multiple domain-specific languages. *Model Driven Engineering Languages and Systems*, pages 46–60, 2007.

[8] Sagar Sen, Benoit Baudry, and Hans Vangheluwe. Domain-specific model editors with model completion. *Models in Software Engineering*, pages 259–270, 2008.

[9] Jules White, Douglas C. Schmidt, Andrey Nechypurenko, and Egon Wuchner. Model intelligence: an approach to modeling guidance. *UPGRADE*, 9(2):22–28, 2008.

[10] Mikoláš Janota, Victoria Kuzina, and Andrzej Wąsowski. Model construction with external constraints: An interactive journey from semantics to syntax. *Model Driven Engineering Languages and Systems*, pages 431–445, 2008.

[11] James H. Hill. Measuring and reducing modeling effort in domain-specific modeling languages with examples. In *2011 18th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, pages 120–129, April 2011.

[12] Krishnakumar Balasubramanian, Jaiganesh Balasubramanian, Jeff Parsons, Aniruddha Gokhale, and Douglas C. Schmidt. A Platform-Independent Component Modeling Language for Distributed Real-Time and Embedded Systems. In *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium (RTAS'05)*, San Francisco, CA, March 2005.

[13] Tanumoy Pati, Dennis C. Feiock, and James H. Hill. Proactive modeling: Auto-generating models from their semantics and constraints. In *Proceedings of the 2012 Workshop on Domain-specific Modeling*, DSM '12, pages 7–12, New York, NY, USA, 2012. ACM.

[14] Tanumoy Pati. Auto-generating models from their semantics and constraints. Master's thesis, Purdue University, 2012.

[15] Object Management Group. *Object Constraint Language*, 2006.

[16] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.

[17] Dániel Varró. *Model Transformation by Example.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[18] Gerti Kappel, Philip Langer, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. Conceptual modelling and its theoretical foundations. chapter Model Transformation By-example: A Survey of the First Wave, pages 197–215. Springer-Verlag, Berlin, Heidelberg, 2012.

[19] Yu Sun, Jules White, and Jeff Gray. *Model Transformation by Demonstration.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[20] Robin Burke. *Hybrid Web Recommender Systems*, pages 377–408. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[21] Tariq Mahmood and Francesco Ricci. Improving recommender systems with adaptive conversational strategies. In *Proceedings of the 20th ACM Conference on Hypertext and Hypermedia*, HT '09, pages 73–82, New York, NY, USA, 2009. ACM.

[22] Paul Resnick and Hal R. Varian. Recommender systems. *Commun. ACM*, 40(3):56–58, March 1997.

[23] Francesco Ricci, Lior Rokach, and Bracha Shapira. *Introduction to recommender systems handbook.* Springer, 2011.

[24] Paul Resnick and Hal R Varian. Recommender systems. *Communications of the ACM*, 40(3):56–58, 1997.

[25] J Ben Schafer, Joseph Konstan, and John Riedl. Recommender systems in e-commerce. In *Proceedings of the 1st ACM conference on Electronic commerce*, pages 158–166. ACM, 1999.

[26] Benjamin A Clegg, Gregory J DiGirolamo, and Steven W Keele. Sequence learning. *Trends in cognitive sciences*, 2(8):275–281, 1998.

[27] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 207–216, New York, NY, USA, 1993. ACM.

[28] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, and A. Inkeri Verkamo. Advances in knowledge discovery and data mining. chapter Fast Discovery of Association Rules, pages 307–328. American Association for Artificial Intelligence, Menlo Park, CA, USA, 1996.

[29] Mukund Deshpande and George Karypis. Item-based top-n recommendation algorithms. *ACM Transactions on Information Systems (TOIS)*, 22(1):143–177, 2004.

[30] Yehuda Koren. Collaborative filtering with temporal dynamics. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 447–456, New York, NY, USA, 2009. ACM.

[31] Guy Shani, Ronen I. Brafman, and David Heckerman. An mdp-based recommender system. In *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence*, UAI'02, pages 453–460, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[32] Gediminas Adomavicius and Alexander Tuzhilin. Context-aware recommender systems. In *Recommender systems handbook*, pages 191–226. Springer, 2015.

[33] Lior Rokach. Ensemble-based classifiers. *Artificial Intelligence Review*, 33(1):1–39, 2010.

[34] R. Polikar. Ensemble based systems in decision making. *IEEE Circuits and Systems Magazine*, 6(3):21–45, Third 2006.

[35] David Opitz and Richard Maclin. Popular ensemble methods: An empirical study. *Journal of Artificial Intelligence Research*, 11:169–198, 1999.

[36] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*, pages 285–295. ACM, 2001.

[37] Thierry Bertin-Mahieux, Daniel PW Ellis, Brian Whitman, and Paul Lamere. The million song dataset. In *ISMIR*, volume 2, page 10, 2011.

[38] Yoav Freund, Robert E Schapire, et al. Experiments with a new boosting algorithm. In *icml*, volume 96, pages 148–156, 1996.

[39] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.*, 55(1):119–139, August 1997.

[40] Robert E Schapire, Yoav Freund, Peter Bartlett, Wee Sun Lee, et al. Boosting the margin: A new explanation for the effectiveness of voting methods. *The annals of statistics*, 26(5):1651–1686, 1998.

[41] Robert E Schapire and Yoram Singer. Improved boosting algorithms using confidence-rated predictions. *Machine learning*, 37(3):297–336, 1999.

[42] Thomas G Dietterich. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*, pages 1–15. Springer, 2000.

[43] Andreas Töscher, Michael Jahrer, and Robert M Bell. The bigchaos solution to the netflix grand prize. *Netflix prize documentation*, pages 1–52, 2009.

[44] Adbc framework. `https://github.com/DOCGroup/ADBC`. Last Accessed: February, 2017.

[45] Mike Owens and Grant Allen. *SQLite*. Springer, 2010.

[46] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *ACM SIGMOD Record*, 22(2):297–306, 1993.

[47] John Brooke et al. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

[48] Xia Ning and George Karypis. Slim: Sparse linear methods for top-n recommender systems. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 497–506. IEEE, 2011.

[49] James R Lewis and Jeff Sauro. The factor structure of the system usability scale. In *International Conference on Human Centered Design*, pages 94–103. Springer, 2009.

[50] John Brooke. Sus: A retrospective. *J. Usability Studies*, 8(2):29–40, February 2013.

[51] Thomas S Tullis and Jacqueline N Stetson. A comparison of questionnaires for assessing website usability. In *Usability professional association conference*, pages 1–12, 2004.

[52] Aaron Bangor, Philip Kortum, and James Miller. Determining what individual sus scores mean: Adding an adjective rating scale. *Journal of usability studies*, 4(3):114–123, 2009.

[53] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[54] John O'Donovan and Barry Smyth. Trust in recommender systems. In *Proceedings of the 10th International Conference on Intelligent User Interfaces*, IUI '05, pages 167–174, New York, NY, USA, 2005. ACM.

[55] Mohsen Jamali and Martin Ester. A matrix factorization technique with trust propagation for recommendation in social networks. In *Proceedings of the fourth ACM conference on Recommender systems*, pages 135–142. ACM, 2010.

[56] Loren Terveen and Will Hill. Beyond recommender systems: Helping people help each other. *HCI in the New Millennium*, 1(2001):487–509, 2001.

[57] Olga C Santos. *Educational Recommender Systems and Technologies: Practices and Challenges: Practices and Challenges*. IGI Global, 2011.

[58] Nikos Manouselis, Hendrik Drachsler, Riina Vuorikari, Hans Hummel, and Rob Koper. Recommender systems in technology enhanced learning. In *Recommender systems handbook*, pages 387–415. Springer, 2011.

[59] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1997.

[60] Gme manual. `https://forge.isis.vanderbilt.edu/gme/GME\%20Manual\%20and\%20User\%20Guide/GME\%20Manual\%20and\%20User\%20Guide.pdf`. Last Accessed: February, 2017.

[61] Wix toolset. `http://wixtoolset.org/`. Last Accessed: February, 2017.

APPENDICES

A PME HANDLERS AND MODELING ACTIONS FROM OCL EXPRESSION FAILURES

This chapter gives an overview about how the proactive modeling handlers work in completing any model created. It also gives the classification of the OCL expression failures which help in identifying the types of modeling actions.

A.1 PME Handlers

There are 5 handlers [14] in PME which are listed below with a example to illustrate the working and drawbacks (if any) of each handler:

1. **Containment Handler**: This handler handles all the containment constraints. It ensures that at least the minimum required elements but not exceeding the maximum limit are present in the model created. The elements and sub-elements created as per the constraints mentioned in the metamodel. For example, in the LMS metamodel, the Library model must contain 2 to 15 Librarians. Therefore, the containment handler adds at least 2 Librarians when a Library model is created.

   A drawback of this handler is that if the modeler wants to add more Librarians, the user guidance handler must be used.

2. **Containment Check Handler**: This is the handler which checks that the containment constraints are not *violated* if the number of elements present falls short or is more than the required number. For example, in the LMS metamodel, when a Library model is created then at least 2 Librarians are added. So, if we delete a Librarian, this handler will add another Librarian. Also, if we add 15 Librarians and then try to add 1 more Librarian then the maximum limit of 15

Librarians would be violated as there would be 16 Librarians. Therefore, this handler would prompt the modeler to make a selection to delete a Librarian from the available set of Librarians.

3. **Association Handler**: This is the handler which enables associations or connections between valid elements as specified in the metamodel. It is used to handle association constraints. This handler is triggered for only those objects which act as source of the connection. For example, in LMS, if the Borrows connection needs to be made between a Patron and a Book present, we would select the particular Patron which would trigger the association handler as Patron is the source of the connection Borrows. It will display to select the type of connection, which is Borrows, and upon this selection it would display a list of valid Books which is the target of Borrows connection based on constraints specified.

   There are two drawbacks with this approach. Firstly, this handler is triggered along with the model guidance handler due to which if the modeler uses only one handler, then the other handler must be closed. This increases the modeling effort. Secondly, if the association is made for Borrows, still the modeler must make two selections of first the connection Borrows and then of a valid Book every time the modeler needs to make Borrows connection for the Patron. There is no mechanism for remembering or automatically detecting this action in PME.

4. **Reference Handler**: This is the handler which handles all the reference constraints. It takes care of the creation of aliases as specified in the metamodel. When a reference element is dragged into the modeling area, the reference handler is triggered. It will display the list of valid objects that the reference object can refer to. For example, in the LMS, when a PatronRef object is dragged into the model, the reference handler is triggered and it gives a valid list of Patron objects which the PatronRef object can refer to.

   A disadvantage of this Handler is that, the modeler needs to find the reference object in the `Part Browser` (refer to Appendix C) and adding a reference object can be done using the rodel guidance handler also.

5. **User Guidance Handler**: This is the handler which is used to display all the general modeling actions. The modeler can use the dialog box provided to perform general modeling actions such as "Add Model contained by Rootfolder", "Add an Atom or Model", "Delete an Atom or Model", "Add a Reference", "Delete a Reference" and "Create a Connection". [14].

   However, once a modeler performs any of these actions, PME does not store the actions for future use. Also, the modeler needs to carry out the same number of steps to perform that modeling action. So, if the modeler needs to add 10 Patrons in the Library model in LMS, the modeler needs to perform the action using the model guidance handler 10 times. This, also, increases the modeling effort.

6. **Attributes Handler**: This handler is used to handle all the attribute constraints. It is used to set the equal expression failures when the elements are created. For example, in CRS, if the salary attribute of Mechanic must be equal to 35000, the attributes handler sets the salary attribute of Mechanic to 35000 whenever a Mechanic element is added to the model. However, for a non deterministic action like representative salary must be in the range of 40000 to 50000, the modeler must decide the correct value for that attribute.

   PME does not give any prompt for entering the values in a dialog box. The modeler must manually go over the object constraints so that the value can be changed accordingly.

A.2 OCL Expression Failures in Modeling

The modeling constraints specified in OCL are classified as the following failures. These failures need to be resolved so that the model can eventually be completed.

The types of failures identified are given below:

1. **Equal Expression Failure**: Constraints that fail due to *equal* expression not being satisfied. An example of the equal expression as shown in Figure A.1, if

the representative salary attribute must be 35000, it is a deterministic action as there is no possible value, other than 35000, that must be assigned.

```
(self.RepSalary = 35000)
```

Figure A.1. Representative Salary Equal Expression Constraint

2. **Not Equal Expression Failure**: Constraints that fail due to *not equal* expression not being satisfied. An example of the not equal expression as shown in Figure A.2, if the representative salary attribute cannot be 35000, then it is a non-deterministic action, as the representative salary attribute can have any value below 35000 or any value above 35000. Therefore, this value must be set by the modeler.

```
(self.RepSalary <> 35000)
```

Figure A.2. Representative Salary Not Equal Expression Constraint

3. **Greater Expression Failure**: Constraints that fail due to *greater* expression not being satisfied. An example of the greater expression as shown in Figure A.3, if the representative salary attribute must be greater than 35000, then it is a non-deterministic action, as the representative salary attribute can have any value above 35000. Therefore, this value must be set by the modeler.

```
(self.RepSalary > 35000)
```

Figure A.3. Representative Salary Greater Expression Constraint

4. **Greater Equal Expression Failure**: Constraints that fail due to *not greater* expression not being satisfied. An example of the greater equal expression as shown in Figure A.4, if the representative salary attribute must be 35000 or greater, then it is a non-deterministic action, as the representative salary attribute

can have any value as 35000 or above. Therefore, this value must be set by the modeler.

```
(self.RepSalary >= 35000)
```

Figure A.4. Representative Salary Greater Equal Expression Constraint

5. **Lesser Expression Failure**: Constraints that fail due to *lesser* expression not being satisfied. An example of the lesser expression as shown in Figure A.5, if the representative salary attribute must be lesser than 35000, then it is a non-deterministic action, as the representative salary attribute can have any value below 35000. Therefore, this value must be set by the modeler.

```
(self.RepSalary < 35000)
```

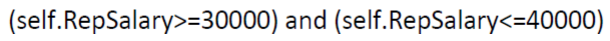
Figure A.5. Representative Salary Lesser Expression Constraint

6. **Lesser Equal Expression Failure**: Constraints that fail due to *lesser equal* expression not being satisfied. An example of the lesser equal expression as shown in Figure A.6, if the representative salary attribute must be 35000 or lesser then it is a non-deterministic action, as the representative salary attribute can have any value 35000 or below. Therefore, this value must be set by the modeler.

```
(self.RepSalary <= 35000)
```

Figure A.6. Representative Salary Lesser Equal Expression Constraint

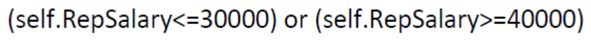
7. **And Expression Failure**: Constraints that fail due to *and* expression not being satisfied. And expression failures are combinations of the above comparison failures. An example of the and expression as shown in Figure A.7, if the representative salary attribute must be 30000 or greater but not more than 40000. Thus, it can be any value in the range of 30000 to 40000 which has to be set by the modeler.



```
(self.RepSalary>=30000) and (self.RepSalary<=40000)
```

Figure A.7. Representative Salary And Expression Constraint

8. **Or Expression Failure**: Constraints that fail due to *or* Expression not being satisfied. Or expression failures are combinations of the above comparison failures. An example of the or expression as shown in Figure A.8, if the representative salary must be 30000 or lesser, or else, 40000 or greater. Thus, it can be any value below and 30000, or else, 40000 or above which has to be set by the modeler.



```
(self.RepSalary<=30000) or (self.RepSalary>=40000)
```

Figure A.8. Representative Salary Or Expression Constraint

A.3 Types of Modeling Actions

The modeling actions are divided into deterministic and non-deterministic modeling actions.

1. **Deterministic Modeling Actions**: These are the modeling actions which do not require modeler intervention. An equal expression failure evaluates to a deterministic modeling action. For example, representative salary has OCL constraint as shown in Figure A.1, then it is a deterministic modeling action as the value must be 35000 which is an equal expression failure. Also, the at least containment condition related to addition of elements is considered as deterministic modeling action. Another example is, if a Garage model is created from the CRS metamodel, the at least containment condition that the Garage must have at least 1 Customer is a deterministic modeling action. The deterministic modeling actions are automatically completed by proactive modeling technique.

2. **Non-Deterministic Modeling Actions**: These are the modeling actions which require modeler intervention. Any expression failure, other than equal expression, evaluates to non-deterministic modeling actions with the exception of at least containment condition. Adding connections to valid target elements, addition of new elements and deletion of existing elements are considered as non-deterministic modeling actions as it must be determined by the modeler. Attribute modeling actions in which the attribute can have any value is also considered as a non-deterministic modeling action. For example, if representative salary can have any value, then it is left to the modeler to decide an appropriate value. This means that the representative salary can only be decided based on modeler intervention.

The non-deterministic modeling actions are the ones to be completed by the modeler. The modeler needs to have specific domain knowledge as to which are the non-deterministic modeling actions which are not getting satisfied, and accordingly, perform the same towards model completion as required.

B RECOMMENDATIONS CLASSES AND SYSTEM DESIGN

This chapter of the thesis gives an overview of how the recommender system was programmed, *i.e.*, the classes and their functionality, how they interact with one another and how each of those components are integrated into making our system work.

B.1 Class Design

B.1.1 Addition of Failure Classes in OCL Project

In the GAME OCL project, we have added the classes for each of the failures as described in Section A. A pictorial representation of the class hierarchy is shown in Figure B.1. They are described as follows:

- **Expression Failure**: This class acts as the base class for any expression failures.
- **Equality Expression Failure**: This class is derived from `Expression Failure`. This class deals with the equality expression failures like *Conjunction* and *Comparison* type of expression failures.
- **Comparison Expression Failure**: This class is derived from `Equality Expression Failure`. This class forms the base class of failures for comparisons like equal to, greater than equal to and lesser than equal to.
- **Conjunction Expression Failure**: This class is derived from `Equality Expression Failure`. This class forms the base class for conjunction failures for, "and", and "or" failures.
- **Equal Expression Failure**: This class is derived from `Comparison Expression Failure`. This class deals with the failure of equality between two variables.

- **Not Equal Expression Failure**: This class is derived from `Comparison Expression Failure`. This class deals with the failure of non equality between two variables.
- **Greater Expression Failure**: This class is derived from `Comparison Expression Failure`. This class deals with the failure of greater than relation between two variables.
- **Greater Equal Expression Failure**: This class is derived from `Comparison Expression Failure`. This class deals with the failure of greater than equal to relation between two variables.
- **Lesser Expression Failure**: This class is derived from `Comparison Expression Failure`. This class deals with the failure of less than relation between two variables.
- **Lesser Equal Expression Failure**: This class is derived from `Comparison Expression Failure`. This class deals with the failure of less than equal to relation between two variables.
- **And Expression Failure**: This class is derived from `Conjunction Expression Failure`. This class deals with the failure of and operation.
- **Or Expression Failure**: This class is derived from `Conjunction Expression Failure`. This class deals with the failure of or operation.
- **Expression Failure Visitor**: This acts as a base class for any expression failure visitors. It makes use of the *Visitor Pattern* [59].

When any failure occurs in the OCL expression, failure objects are created, as per the failure, and stored to resolve them. The resolution of these failures take place in the derived class of the `Expression Failure Visitor` where the programmer can specify what specific actions are to be done to address those failures.

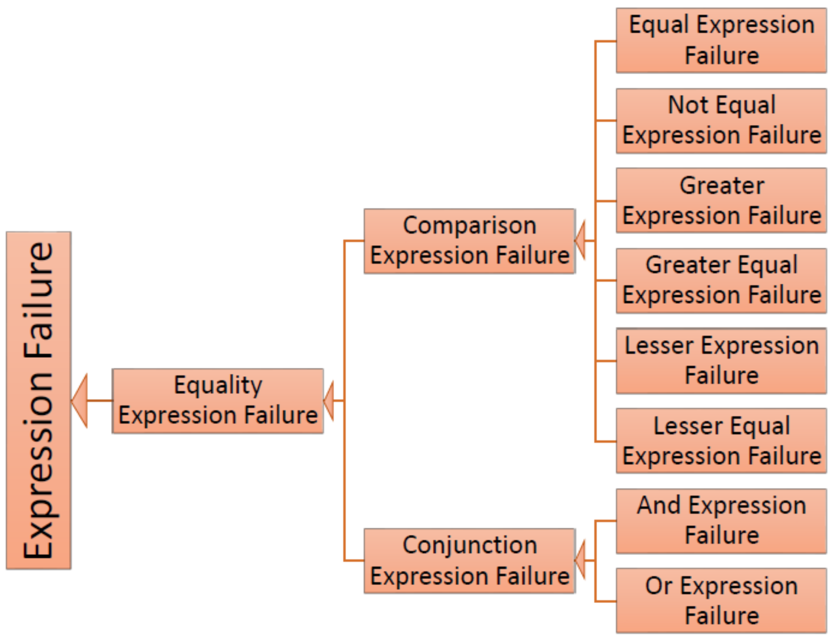

Expression Failure
Equality Expression Failure
Comparison Expression Failure
Conjunction Expression Failure
Equal Expression Failure
Not Equal Expression Failure
Greater Expression Failure
Greater Equal Expression Failure
Lesser Expression Failure
Lesser Equal Expression Failure
And Expression Failure
Or Expression Failure

Figure B.1. Expression Failure Hierarchy.

B.1.2 Addition of Recommendations in Model Intelligence Project

We have created classes for the recommender system in the `GAME Model Intelligence` project. The `Model Intelligence` project already contains the handlers used by PME as described in Section A. We first add a `Recommendation Handler` class which will get triggered on any object select event.

- **Model Intelligence Expression Failure Visitor**: This is the class derived from `Expression Failure Visitor` class which is responsible for handling all the OCL expression failures as described in Section A. It performs the completion of the model as per the deterministic modeling actions auto completed by PME.

- **Recommendation Handler**: This is the class which acts as the main starting point for the recommender system. It is triggered, *via*, object select events. When the modeler clicks on any object in the model editor in GME, this class will be called.

- **Attribute Recommendation Analyzer**: This class is used to analyze all the attribute constraints on the selected object. The non-deterministic modeling actions with respect to the attributes of the selected objects are analyzed and the `Attribute Change Recommendation Command` objects are created for attributes which do not satisfy the constraints. These are then stored in a container which was passed from the `Recommendations Handler`.

- **Attribute Recommendation Visitor**: This class is used to get the attribute name for the non-deterministic modeling action constraint which fails for the attribute of the selected object. It visits all the expression failures as mentioned in Section A. It is used by the `Attribute Recommendation Analyzer` class.

- **Connection Recommendation Analyzer**: This class is used to analyze the connection constraints of a selected object which acts as the source of the connection. It is used by the `Recommendation Handler` to analyze the connection constraints on the selected object. It will create `Connection Recommendation Command` and store in the container provided by `Recommendation Handler`. It is also used to create `Connection General Recommendation Command` for connections created using the `General Modeling Actions Selection Dialog Box`.

- **General Actions Recommendation Analyzer** This class is used to analyze the general actions on the selected object. If any object is selected, the list of general actions displayed is provided by this class. If the modeler performs any general modeling action then this class takes control and creates the appropriate `General Actions Recommendation Command`. It also stores this command in the object provided by the `Recommendation Handler` and the handler then executes this modeling action.

- **Model Action Recommendation Command**: This is the base class for all the modeling actions command objects. It is modeled on *Command Pattern* [59] where it is used to calculate the recommendation scores using the equation as shown in Figure 4.2. It is used as a *Template Method Pattern* so that all the other subclasses will be able to inherit that functionality. A pictorial representation of the class hierarchy is shown in Figure B.2.

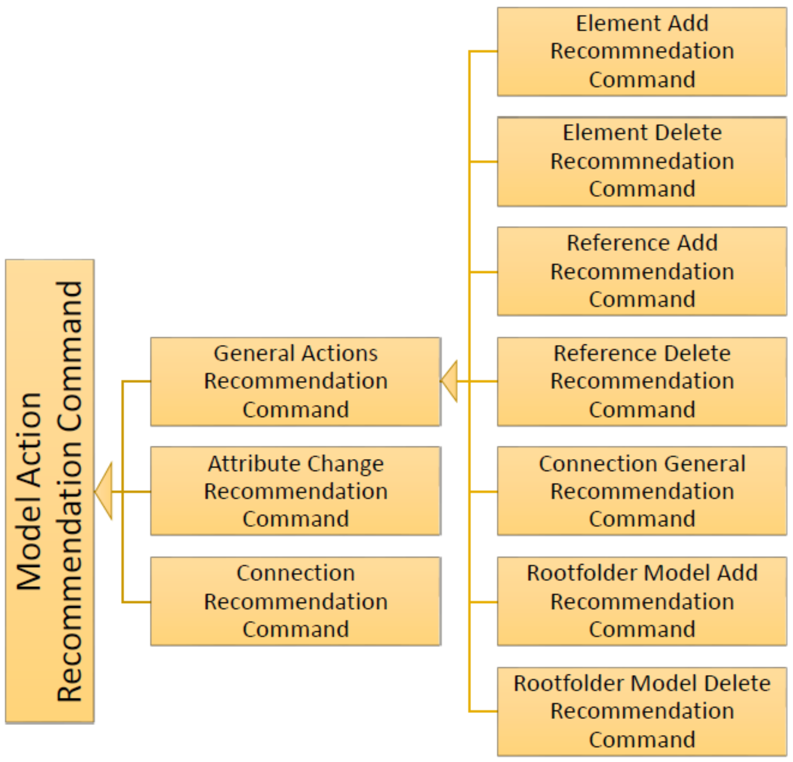

Model Action Recommendation Command
General Actions Recommendation Command
Attribute Change Recommendation Command
Connection Recommendation Command
Element Add Recommnedation Command
Element Delete Recommnedation Command
Reference Add Recommendation Command
Reference Delete Recommendation Command
Connection General Recommendation Command
Rootfolder Model Add Recommendation Command
Rootfolder Model Delete Recommendation Command

Figure B.2. Model Action Recommendation Command Hierarchy.

- **Attribute Change Recommendation Command**: This is a subclass of the *Model Action Recommendation Command*. It is used to create commands of non-deterministic modeling actions with respect to object attributes. When this particular command is executed, the object attribute associated with this command can be changed. It makes use of the `Attribute Value Dialog` for this purpose. It is used by the `Attribute Recommendation Analyzer`.
- **Connection Recommendation Command**: This is a subclass of the `Model Action Recommendation Command`. It is used to create commands of non-deterministic modeling actions with respect to object associations where the selected object is the source of a connection. When this particular command is

executed, the object association with any valid destination, associated with this command can be set. It is used by the `Connection Recommendation Analyzer`.

- **General Actions Recommendation Command**: This is a subclass of the `Model Action Recommendation Command`. It again serves as a base class for the `General Action Commands` selected by the modeler. The `General Action Commands` are always specific to a particular model. For example, in CRS scenario, if a new Customer is added by selecting a Customer object, then the general action with respect to the Customer object will be stored as *adding a Customer object to a Garage model*. The modeler must choose a specific *Garage* model from the available ones. It also acts as a `Template Method Pattern` where this selection of the parent model is common for all the sub classes.
- **Element Add Recommendation Command**: This is a subclass of the `General Actions Recommendation Command`. It deals with the creation and addition of a new Atom or Model. This command is generated using the `General Actions Recommendation Analyzer`.
- **Element Delete Recommendation Command**: This is a subclass of the `General Actions Recommendation Command`. It deals with the deletion of an existing Atom or Model. This command is generated using the `General Actions Recommendation Analyzer`.
- **Connection General Recommendation Command**: This is a subclass of the `General Actions Recommendation Command`. It deals with the creation of any connection between a valid source and destination object. This command is generated using the `General Actions Recommendation Analyzer`.
- **Reference Add Recommendation Command**: This is a subclass of the `General Actions Recommendation Command`. It deals with the creation and addition of a new Reference object. This command is generated using the `General Actions Recommendation Analyzer`.

- **Reference Delete Recommendation Command**: This is a subclass of the `General Actions Recommendation Command`. It deals with the deletion of an existing Reference. This command is generated using the `General Actions Recommendation Analyzer`.
- **Rootfolder Model Add Recommendation Command**: This is a subclass of the `General Actions Recommendation Command`. It deals with the creation and addition of a new Model contained by the Rootfolder. This command is generated using the `General Actions Recommendation Analyzer`.
- **Rootfolder Model Delete Recommendation Command**: This is a subclass of the `General Actions Recommendation Command`. It deals with the deletion of an existing Rootfolder Model. This command is generated using the `General Actions Recommendation Analyzer`.
- **Object Action Sequence State**: This class is used to maintain information pertaining to the sequence based recommendation. It acts like a `Wrapper (Facade)` [59] which stores all the sequence actions, their order and their information, and gives out the action which must be recommended to the modeler, when the sequence based recommendation is triggered.
- **Model Intelligence Recommendation Factory**: This class is the base class for creating the `Model Action Recommendation Command` sub class objects. It acts like an interface whose methods must be implemented by the sub classes. It is based on `Abstract Factory Pattern` [59] used for creation of `Model Action Recommendation Command` objects.
- **Recommendation Command Factory**: This is a sub class of the `Model Intelligence Recommendation Factory` interface. It implements all the methods for creating default as well as parameterized `Model Action Recommendation Command` sub class objects.

- **Model Action Recommendation Visitor**: This class is based on the `Visitor Pattern` [59]. It acts as a base class to visit all the `Model Action Recommendation Command` sub classes to perform specific operations.
- **Model Locator**: It is a sub class of the `GAME Mga Visitor` class which is used to traverse all the model elements in a hierarchical order. This class is specifically used to locate models inside Rootfolders, Folders and other Models.
- **Rootfolder Model Locator**: It is a sub class of the `GAME Mga Visitor` class which is used to traverse all the model elements in a hierarchical order. It locates Rootfolder models in a project. This visitor will traverse the entire Rootfolder and sub folders from where it is started. All models are stored internally, which can be retrieved later.
- **Rootfolder Meta Model Locator**: It is a sub class of the `GAME Mga Visitor` class which is used to traverse all the model elements in a hierarchical order. It locates Rootfolder meta models in a project. This visitor will traverse the entire Rootfolder and sub folders from where it is started. All models are stored internally, and can be retrieved later.
- **Rootfolder Meta Model Folder Locator**: It is a sub class of the `GAME Mga Visitor` class which is used to traverse all the model elements in a hierarchical order. It locates folder of Rootfolder meta model in a project. This visitor will traverse the entire Rootfolder and sub folders from where it is started. All models are stored internally, and can be retrieved later.
- **Recommendation Dialog**: This class is used to display the `Recommendations Dialog box`. It is called by the `Recommendation Handler`. This is described in section 5.1.1.
- **Attribute Value Dialog** This class is used to display the `Attribute Value Dialog Box`. This class is called when the `Attribute Change Recommendation Command` is executed. Figure C.2 shows the a sample `Attribute Value Dialog Box`.

- **Object Constraints Dialog**: This class is used to display the `Object Constraints Dialog Box` which shows all the constraints on the selected object. Section 5.1.1 point 3 of `View Object Constraints` describes the same.
- **Object Name Change Dialog** This class is used to display the `Object Name Change Dialog Box` which is used to change the name of the selected object. Section 5.1.1 point 4 of `Change Object Name` describes the same.
- **Recommendation Object**: This class is used to store information about recommendations pertaining to a specific meta. It stores the list of modeling actions with respect to the particular meta, the action context information, sequence recommendation information as well as other state information.
- **Recommendation Object Cache**: This is the class which contains all the `Recommendation Objects` for quick retrieval. It makes use of the LRU page replacement algorithm as described in section. It contains a cache list of the objects as well as cache map for faster access. It also contains other state information required to maintain the cache information.
- **Recommendation Object Pair** This is a pair to represent name of meta in string format and the `Recommendation Object`.
- **Recommendation Object Cache List**: This is a user defined container using list from `C++ Standard Template Library` to store the `Recommendation Object Pairs`.
- **Recommendation Object Cache Map**: This is a map used to store location of elements in recommendation object cache list for `O(1)` retrieval time. When a particular meta is specified by the string name, the location of the corresponding `Recommendation Object` is retrieved by checking with the map and from that location we get the position in `Recommendation Object Cache List` directly.
- **Recommendation Score Parameters Constants**: This class is used to define the weights assigned to each of the recommendation parameters.

- **Action Score Object**: This class is used to store the name of the modeling action and its associated score.
- **Action Score Descending Sort**: This class sorts `Action Score Objects` in descending order based on their recommendation score. This class is used as a comparator for the `std::sort` function of `Standard Template Library of C++` on recommended actions list in `Recommendation Handler`.
- **Recommendations Handler Database**: This class handler all the database *Data Definition Language (DDL)* and *Data Manipulation Language (DML) CRUD (Create, Read, Update and Delete)* operations. It creates a database if not present inside database file. This uses the *ADBC framework* for *SQLite3* [44].
- **Model Action Recommendation Database Insert Visitor**: This class is a sub class of the `Model Action Recommendation Visitor`. It is used to specify the database insertion statements for each of the `Model Action Recommendation Command` class objects.

B.2 System Design

B.2.1 Interaction of Recommendation Classes

In this section, we describe, how each of the classes work towards making the recommendations system work. The steps are described below:

1. **Recommendations Handler working**: As mentioned before, the main handler is the `Recommendation Handler` which presents the modeler a list of recommended actions. We will see below how each step is performed. The description is also shown in Figure B.3.

   (a) The `Recommendation Handler` checks for the existence of a database file, when the project is opened. If it exists, it uses the same, else it creates a new one. This database file is common to all the projects, under the same paradigm. When the project finishes opening, the handler calls initialize function of the `Recommendation Handler Database` class which in turn checks if the tables are created, and creates them, if absent.

   (b) Now, the 3 `Recommendations Analyzers`, *i.e.*, `attributes, connection and general actions` are used to identify which modeling actions are to be performed for satisfying the failed constraints.

   (c) The recommendation scores are retrieved from the cache and calculated by the `Recommendation Object`.

   (d) We then display the Recommended actions in the descending order of their scores and also the general actions are displayed in the `General Actions Selection Dialog Box`.

   (e) The modeler selects a modeling action and once it is performed, the scores and other parameters such as sequence based recommendation are updated.

   (f) Depending upon the conditions, if the project is closed or if the `Recommendation Object` is removed from the cache, the recommendation information is stored into the database.

   (g) This process is repeated for each object selection.

2. **Least Frequently Used (LRU) optimization**: Here, we describe how the storing of the recommendation information works in order to avoid the delay caused by disk I/O operations as mentioned in Section 5.3. Figure B.4 shows as to how this workflow takes place.

   (a) First we load the recommendation information of the meta selection from the database. So, if we click on *Arvind* which is of the meta *Customer* from CRS in Figure 3.15, we load the recommendation information pertaining to the *Customer* object. We store this into the `Recommendation Cache`.

   (b) We then work with this object and if we perform modeling actions with respect to this object, then this object along with its information stored in the cache. Similarly, we can work with other objects also.

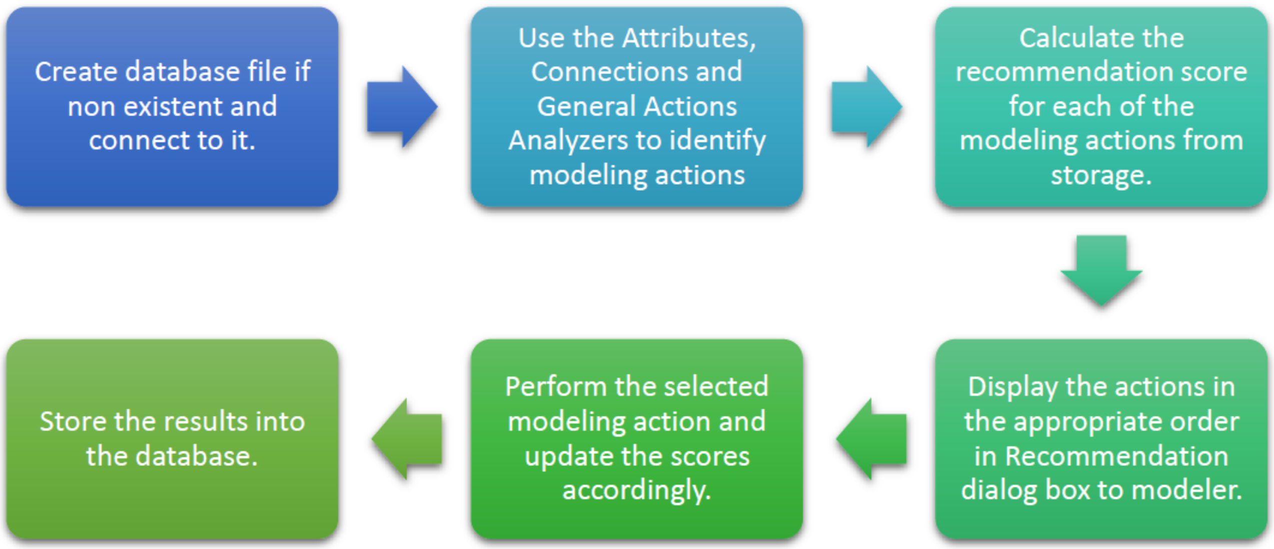

Create database file if non existent and connect to it.
Use the Attributes, Connections and General Actions Analyzers to identify modeling actions
Calculate the recommendation score for each of the modeling actions from storage.
Display the actions in the appropriate order in Recommendation dialog box to modeler.
Perform the selected modeling action and update the scores accordingly.
Store the results into the database.

Figure B.3. Workflow of Recommendation Handler.

(c) As per the LRU policy, the object which is *least used* is removed from the cache. When this object is removed from the cache, then, all of its updated information is updated into the database. So, if the modeler works with a particular set of objects, and then, moves on to other objects, then only the database operation takes place. In the end, when the project is closed the recommendation information of all the other objects in the cache are stored into the database.

3. **Score Calculation working**: Here, we discuss how the calculation of the scoring mechanism takes place. We describe the steps as shown in Figure B.5.

   (a) We first load the initial scores from the database for the meta of the selected object. This would be based on the previous selections or projects.

   (b) We then generate the modeling actions which need to be performed using the `analyzers` as mentioned in *workflow* of `Recommendations Handler`.

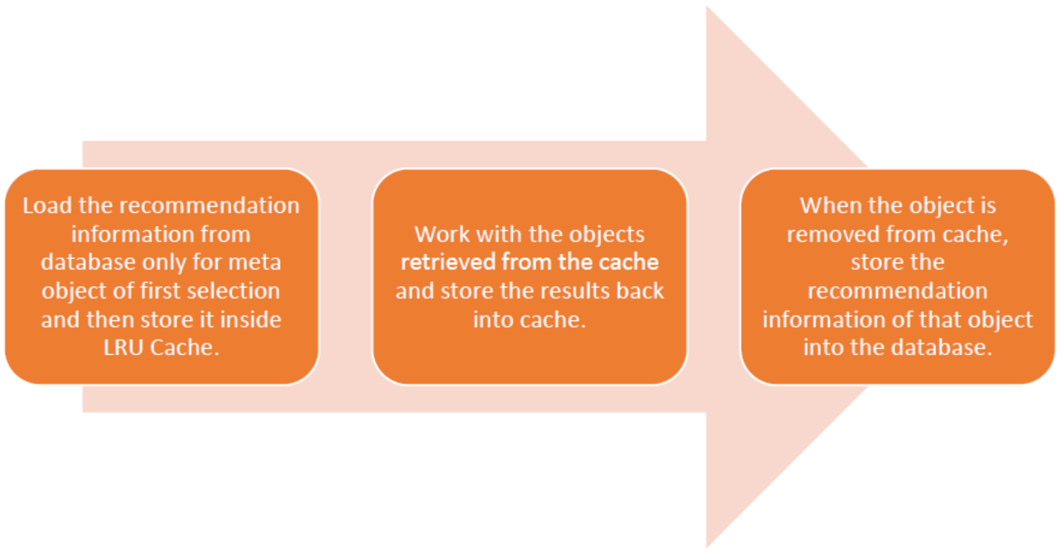

Load the recommendation information from database only for meta object of first selection and then store it inside LRU Cache.
Work with the objects retrieved from the cache and store the results back into cache.
When the object is removed from cache, store the recommendation information of that object into the database.

Figure B.4. Optimization of Information using LRU cache.

(c) We then calculate the scores if necessary considering the previous projects and assign them to the respective modeling actions.

(d) The modeling action along with its associated score is stored in the `Action Score Object` and passed to the `Recommendations Dialog Box` in a container like `vector`.

(e) The objects are then sorted using the `Action Score Descending Sort` comparator class and `std::sort` from C++ Standard Template Library. Then, they are presented to the modeler.

(f) The modeling action performed by the modeler is recorded and retrieved by the `Recommendations Handler`.

(g) The scores and other recommendation information is recalculated and stored back into the cache.

(h) We then repeat the same process for the selection of another object.

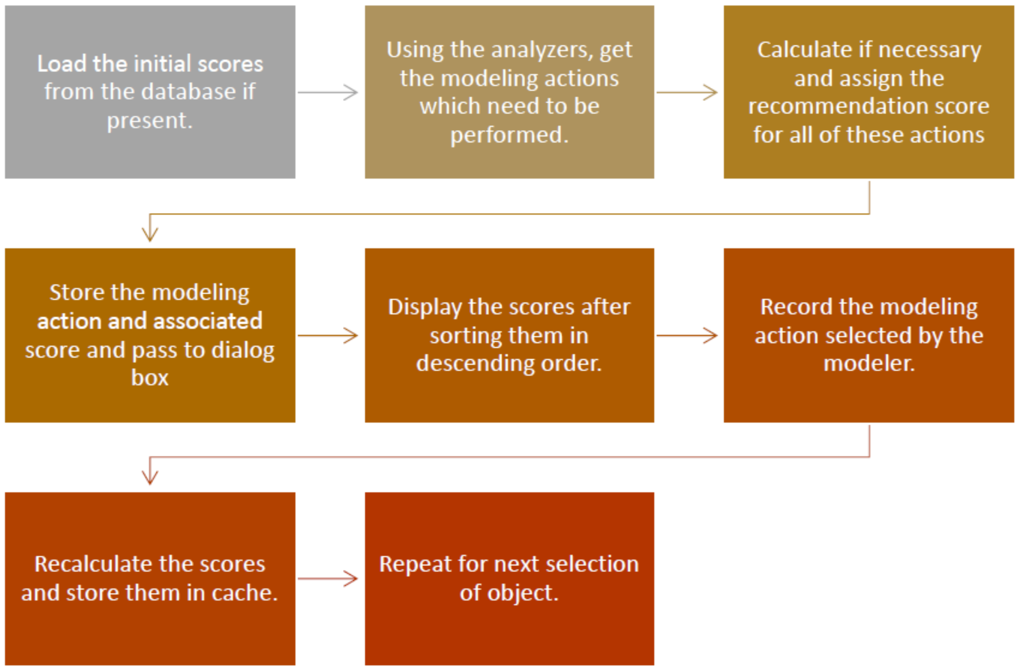

Load the initial scores from the database if present.
Using the analyzers, get the modeling actions which need to be performed.
Calculate if necessary and assign the recommendation score for all of these actions
Store the modeling action and associated score and pass to dialog box
Display the scores after sorting them in descending order.
Record the modeling action selected by the modeler.
Recalculate the scores and store them in cache.
Repeat for next selection of object.

Figure B.5. Calculation of Scoring Mechanism.

B.2.2 Database Relations for Recommendations

In this portion, we will see the database design involved in creating the tables for storing the recommendation information. The database is designed using *ADBC* framework for *SQLite3* [44] which simplifies the *SQLite3 CRUD* operations. They are described as follows:

1. **Meta Model Information Table**: This table is used to store the paradigm name and the project name which is the meta model name and model name respectively. This is used to identify a unique project based on a given paradigm.

2. **Recommendation Object Table**: This table is used to store the `Recommendation Object` class data. It uses model name and selected object name to uniquely identify the recommendation objects of a particular project.

3. **Attribute Change Recommendation Table**: This table is used to store the `Attribute Change Recommendation Command` class data. It uses the name of

the modeling action, the model name and the selected object name to uniquely identify the `Attribute Change Recommendation Command` objects of a particular project.

4. **Connection Recommendation Table**: This table is used to store the `Connection Recommendation Command` class data. It uses the name of the modeling action, the model name and the selected object name to uniquely identify the `Connection Recommendation Command` objects of a particular project.

5. **Element Add Recommendation Table**: This table is used to store the `Element Add Recommendation Command` class data. It uses the name of the modeling action, the model name and the selected object name to uniquely identify the `Element Add Recommendation Command` objects of a particular project.

6. **Element Delete Recommendation Table**: This table is used to store the `Element Delete Recommendation Command` class data. It uses the name of the modeling action, the model name and the selected object name to uniquely identify the `Element Delete Recommendation Command` objects of a particular project.

7. **Reference Add Recommendation Table**: This table is used to store the `Reference Add Recommendation Command` class data. It uses the name of the modeling action, the model name and the selected object name to uniquely identify the `Reference Add Recommendation Command` objects of a particular project.

8. **Reference Delete Recommendation Table**: This table is used to store the `Reference Delete Recommendation Command` class data. It uses the name of the modeling action, the model name and the selected object name to uniquely identify the `Reference Delete Recommendation Command` objects of a particular project.

9. **Rootfolder Model Add Recommendation Table**: This table is used to store the `Rootfolder Model Add Recommendation Command` class data. It uses the name of the modeling action, the model name and the selected object name to uniquely identify the `Rootfolder Model Add Recommendation Command` objects of a particular project.

10. **Rootfolder Model Delete Recommendation Table**: This table is used to store the `Rootfolder Model Delete Recommendation Command` class data. It uses the name of the modeling action, the model name and the selected object name to uniquely identify the `Rootfolder Model Delete Recommendation Command` objects of a particular project.

11. **Model Action Recommendation Table**: This table is used to store the `Model Action Recommendation Command` class data. It uses the name of the modeling action, the model name and the selected object name to uniquely identify the `Model Action Recommendation Command` objects of a particular project. It mainly stores the scores of the various Recommendation parameters such as the *selection count, history count, the action number* and *action presence count.* It also stores the score of the modeling action.

12. **Connection General Recommendation Table**: This table is used to store the *Connection General Recommendation Command* class data. It uses the name of the modeling action, the model name and the selected object name to uniquely identify the *Connection General Recommendation Command* objects of a particular project.

13. **Action Context Table**: This class is used to store the *Action Context data* for the action context recommendation parameter. It uses the name of the modeling action, the model name and the respective action in question to uniquely identify the action context data with respect to particular action, of a particular project.

14. **Connection Qualified Fcos Table**: This class is used to store the qualified *fco paths* for the `Connection Recommendation Command` objects. It uses the name

of the modeling action, the model name, the selected object name and the path of the qualified fco to uniquely identify the `Model Action Recommendation Command` objects of a particular project.

15. **Object Action Sequence State Table**: This class is used to store the `Object Action Sequence State` class data. It uses the model name and the selected object name to uniquely identify the `Object Action Sequence State` data for each meta.

16. **Object Action Sequence Actions Table**: This class is used to store the *sequence order* of the actions in sequence based recommendation. It uses the model name, the selected object name and the name of the action to uniquely identify each modeling action. The action order column stores the sequence of the modeling actions with respect to a particular meta.

17. **User Log Data Table**: This class is used to store the modeler data based on the modeling activity performed. It uses a database unique identifier to identify each modeling action performed by the modeler. The main functionality of this class is to store the position of the modeling action selected and the total number of the recommended actions, at that instant, for calculating the ARHR.

C GAME WITH GME

In this chapter, we discuss a basic overview of GME, how the `GAME Model Intelligence Add-on` was integrated with GME as well as the features of the recommender system work.

C.1 GME Overview

Domain-specific design environments like *Matlab/Simulink* for signal processing capture specifications and automatically generate or configure the target applications in particular engineering fields [2]. The Generic Modeling Environment (GME) developed at the Institute for Software Integrated Systems at Vanderbilt University is a Windows-based configurable toolkit for creating domain-specific modeling and program synthesis environments [2].

The modeling paradigm contains all the syntactic, semantic, and presentation information regarding the domain, such as, which concepts will be used to construct models, what relationships may exist among those concepts, how the concepts may be organized and viewed by the modeler, and rules governing the construction of models [2]. The modeling paradigm defines the family of models that can be created using the resultant modeling environment [2]. In the example of CRS, we can create various garage scenarios taking the main *Garage* paradigm as a blueprint. The important features of GME software are as follows:

C.1.1 GME Modeling Concepts

The GME modeling concepts is the vocabulary of domain specific languages based on a set of generic concepts built into GME software. They are described as follows [2]:

1. **Project**: It contains a set of Folders.
2. **Folder**: Folders are containers that help organize models similar to the way folders organize files on an operating system. Folders contain Models. It is essential that there needs to be atleast one Rootfolder, which is a special type of folder at the top of the hierarchy, in order to start modeling.
3. **First Class Objects (FCO)**: FCOs are Models, Atoms, Connections, References and Sets in GME.
4. **Model**: Models are compound objects which can contain other Models, Atoms, Connections, References and Sets. They can have parts and inner structure.
5. **Atom**: Atoms are the elementary objects, *i.e.*, they cannot contain parts.
6. **Connection**: It is used to express a relationship between objects contained by the same Model.
7. **Reference**: References are similar to pointers in object oriented programming languages. A Reference is not a "real" object, it just refers to, *i.e.*, points to one. In GME, a Reference must appear as a part in a Model. It acts as an alias for a particular object.
8. **Set**: It is used to specify a relationship among a group of objects. The only restriction is that all the members of a Set must have the same container (parent) and be visible in the same Aspect.
9. **Attribute**: It is used to specify textual information for objects. The kinds of Attributes available are text, integer, double, boolean and enumerated.
10. **Aspect**: It is used to represent different views of a model. It is used to maintain readability by filtering the models.
11. **Constraint**: Constraints are rules specified in OCL to which the objects conform to.
12. **Meta**: Meta is the blueprint from which many objects are created. For example, from CRS, Customer *Arvind* is the object created from meta Customer.

C.1.2 GME Interfaces

While modeling in GME, the modeler uses these 6 interfaces. They are described as follows [60] from Figure C.1:

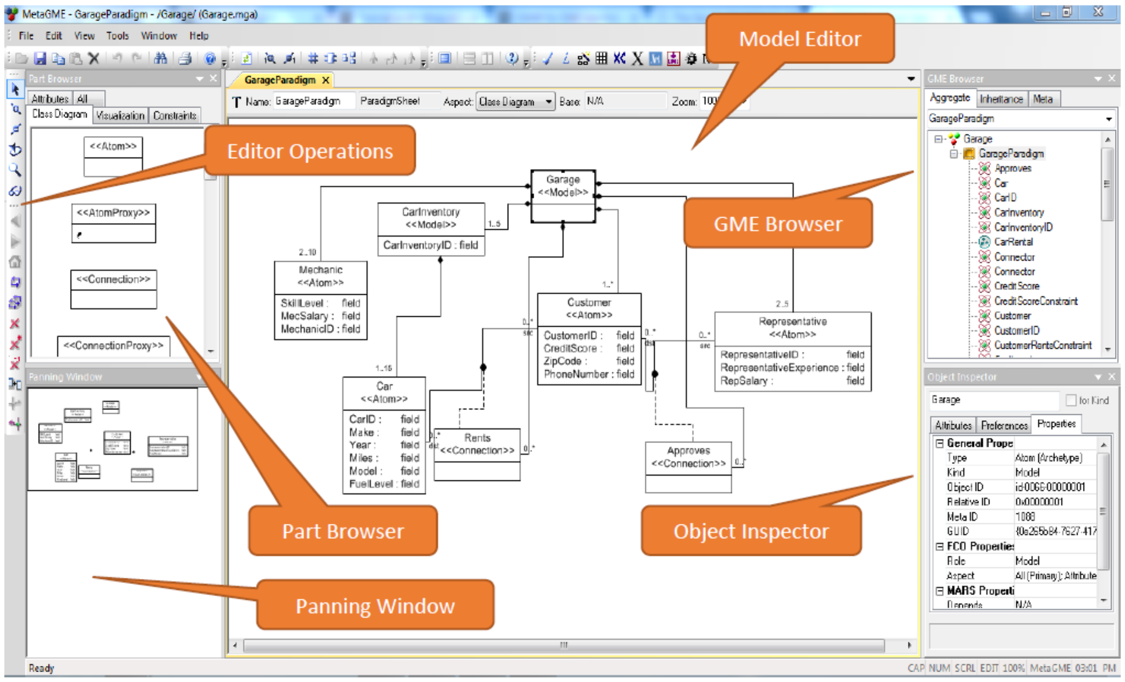


Figure C.1. GME Interfaces.

1. **Model Editor**: This is the main modeling window where the models are added by the modeler.
2. **GME Browser**: Here, we can browse for all of the elements present in the Model Editor.
3. **Object Inspector**: Here, the attributes of a selected object in the *Model Editor* can be changed.
4. **Panning Window**: This is used to pan/zoom into a particular part in the Model Editor.
5. **Editor Operations**: This provides the modeler various editing functionalities which are given below:

- **Normal Mode**: This enables adding, copying, moving and deleting parts in the *Model Editor*.
- **Add Connection mode**: This is used to add a connection between two valid objects.
- **Delete Connection mode**: This is used to delete an existing connection.
- **Set mode**: This enables addition of objects into the set of a model.
- **Zoom mode**: This is used to zoom into various parts of the model in the Model Editor.
- **Visualization mode**: This is used for visually highlighting objects with respect to other objects.

6. **Part Browser**: This is used to obtain various parts to complete modeling for a particular paradigm.

C.1.3 GME Addons

GME `Add-ons` are basically paradigm independent libraries which can be programmed externally. They are reusable components that can react to GME events sent by the `COM Mga-Layer` [60]. These components are very useful to make GME a run-time executional environment or to write more sophisticated paradigm dependent or independent extensions [60]. `Model Intelligence` is one such paradigm which is developed in the GAME project which includes PME along with recommendations which is the programmed component of this thesis.

C.2 GAME Installation for GME

Here, we describe how to install and enable `GAME Model Intelligence Add-on` for GME. The `GAME Microsoft Installer (MSI)` for our system was created using the `WiX Toolset` [61]. We perform the following steps:

1. **Operating System**: The preferred operating system for installing this software is *Windows 7*, *8* or *10* as GME is a Windows based application.

2. **Visual Studio 2012**: We need to install *Visual Studio 2012* so that the essential runtime libraries are present which is used for running `GAME Model Intelligence` successfully.
3. **GME**: We need to install GME software provided by Vanderbilt University.
4. **GAME**: We then need to download the MSI from the *GAME repository* and install it. This will install all the *GAME Model Intelligence Dynamic Linked Libraries (dlls)* needed by the `Model Intelligence Add-on`.

Once these steps are completed, we can enable the `Model Intelligence Add-on` inside GME. A complete list of steps is given at: `https://github.iu.edu/SEDS/GAME/wiki/GAME-for-GME-Installation`.

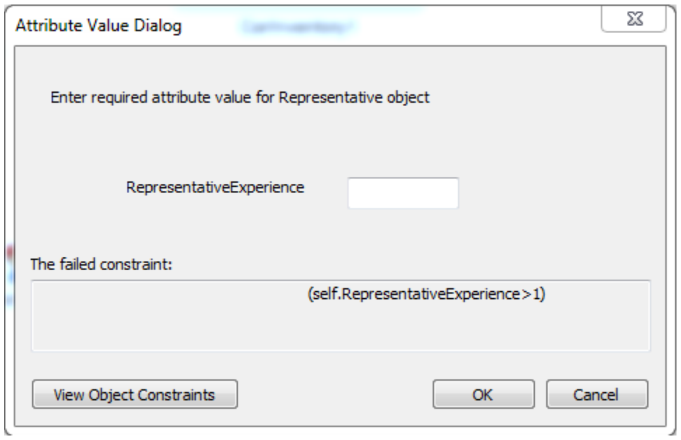
C.3 Features of the Recommender System with Modeling Actions

Here, we describe the features of the recommender system. The `Recommendation Dialog Box` and its associated features are described in Section 5.1. Here, we will describe the various modeling actions which can be performed using the same. The various modeling actions are described in detail pertaining to the *sample Garage example* of CRS paradigm in Figure 3.15 as follows:

1. **Change Object Attributes**: If suppose we want to change the representative experience attribute from Figure 5.1, we click on the Representative *Pulkit* and select modeling action "Change attribute representative experience for Representative object" and then we would get the `Attribute Dialog Box` as shown in Figure C.2.

   Here, we can also view the failed constraint and also view the other constraints on that particular Representative object. We enter a valid value, which is above 1, in the text box and press `OK`. This changes the representative experience attribute for the selected Representative object *Pulkit*.

2. **Add Association for selected source object**: If we want to add an association Approves from Representative *Pulkit* to some valid Customer object



(self.RepresentativeExperience>1)

Figure C.2. Attribute Value Dialog.

e.g., Arvind, we first click on the Representative *Pulkit* and select modeling action "Add Connection Approves for selected object Representative". Then we get a list of valid Customer objects which would include Customer *Arvind*. We select Customer *Arvind* and click `OK`. This would add an association from Representative *Pulkit* to Customer *Arvind*.

3. **Add an Atom or Model to the Main Model**: If we want to add a Customer atom to the Garage model, we would select a Customer object, for example, Customer *Arvind* and click on the "Actions relevant to existing models" from the `General Modeling Actions Selection Box`. Then, we select the parent model *Garage* from the dialog box and then select "Add an Atom or Model" from the `User Guidance Dialog Box`. Then we are presented with the target Atom or Model for creation, in which, we are given the valid meta objects in the *Garage* model.

   We select Customer and click `OK` which adds a new Customer object. We can perform the general actions from selection of any object of any meta. However, it is appropriate to use select a Customer object for adding another Customer object, *i.e.*, associate the actions for a metawith respect to that meta.

4. **Delete an Atom or Model from the Main Model**: If we want to delete a Customer atom *e.g.*, Customer *Arvind* from the *Garage* model, we would select Customer *Arvind* and click on the "Actions relevant to existing models" from the `General Modeling Actions Selection Box`. Then, we select the parent model *Garage* from the dialog box and then select "Delete an Atom or Model" from the `User Guidance Dialog Box`.

   Then we are presented with the target atom or model for deletion in which we are given the valid meta objects in the *Garage* model. We select Customer and we are presented with a list of valid Customer objects for deletion. From that list, we select Customer *Arvind* and click `OK`. This would delete Customer *Arvind* object.

5. **Add a Reference to the Main Model**: In the LMS scenario, suppose we want to add a reference to the Patron *Arvind*, we would select Patron *Arvind* and click on the "Actions relevant to existing models" from the `General Modeling Actions Selection Box`. Then, we select the parent model Library from the dialog box and then select "Add a Reference" from the `User Guidance Dialog Box`.

   We then select the target reference for creation from the list of valid reference metas for the *Library* model. Then we get the list of valid Patron objects in which we can select Patron *Arvind* and a reference to Patron *Arvind* is added.

6. **Delete a Reference from the Main Model**: In the LMS scenario, suppose we want to delete a reference to the Patron *Arvind*, we would select Patron reference *Arvind* and click on the "Actions relevant to existing models" from the `General Modeling Actions Selection Box`. Then, we select the parent model *Library* from the dialog box and then select "Delete a Reference" from the `User Guidance Dialog Box`.

   We then select the target reference for deletion from the list of valid reference metas for the *Library* model. Then, we get the list of valid Patron reference objects in which we can select Patron reference *Arvind* and the Patron reference is deleted.

7. **Creating an Association from General Actions**: Suppose, we want to create an Approves association between Representative *Pulkit* and Customer *Arvind* through the reneral actions. We would the select suppose Mechanic *Sam*. Then in the `Recommendations Dialog Box`, select “Actions relevant to existing models” and then select the parent model which is Garage. Then we select the operation “Create a Connection” and then select *Approves* connection.

   We then select the source as *Pulkit* from the valid source Representative objects and then select destination as *Arvind* from the valid Customer objects. This creates an association from Representative *Pulkit* to Customer *Arvind*.

8. **Add a Model contained by Rootfolder**: We return to the CRS scenario, in which we want to add another *Garage* model to the Rootfolder. So, we select Customer *Arvind* object click on the “Add a model contained by rootfolder” from the `General Modeling Actions Selection Box`. The we are displayed the valid models which can be added in the Rootfolder. We select *Garage* model and click `OK`. This adds a new *Garage* model in the Rootfolder.

9. **Delete a Model contained by Rootfolder**: If we want to delete an existing *Garage* model from the Rootfolder. So, we select Customer *Arvind* object click on the “Delete a model contained by rootfolder” from the *General Modeling Actions Selection Box*. The we are displayed the valid models which can be deleted from the Rootfolder. We select the appropriate *Garage* model to be deleted and click `OK`. This adds a deletes the selected *Garage* model in the Rootfolder.

The modeling actions numbered from 3 to 9 are general actions. The `User Guidance Dialog Box` is shown in Figure C.3 which is used in actions 3 to 7. If a modeler selects a general modeling action with respect to a particular meta then this general action would next time be displayed in the `Recommended Actions Selection Box`. The

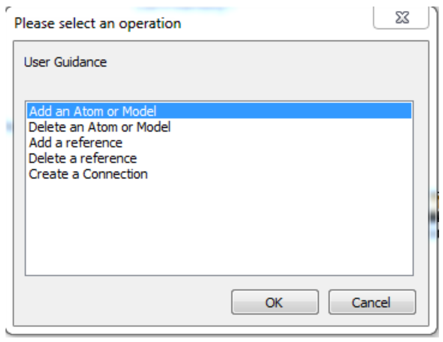


Figure C.3. User Guidance Dialog.

modeler can select from the recommended actions and the initial number of steps are reduced. The steps reduced are described below:

- **For the addition of Models, Atoms and References**: We just need to select the parent model and skip selecting the meta.
- **For the addition of Models, Atoms and References**: We just need to select the parent model and the object for deletion and skip selecting the meta.
- **For Addition of Model in Rootfolder**: We just need to select the action and skip selecting the meta.
- **For Deletion of Model in Rootfolder**: We just need to select the model for deletion.
- **For adding General Connection**: We just need to select the parent model, the source and target objects of the connection and skip the selection of connection name.

The complete *step-by-step* modeling actions tutorial for the recommender system along with screenshots can be viewed at `https://github.iu.edu/SEDS/GAME/wiki/Tutorial-on-Working-with-GAME-Model-Intelligence`.

D ATTRIBUTES OF MODELING ELEMENTS OF EXAMPLE PARADIGMS

In this section, we discuss the CRS and LMS modeling paradigm elements.

D.1 Attributes of Modeling Elements of CRS and LMS

Here, we describe the attributes of the various modeling elements of the CRS metamodel and the LMS metamodel discussed in Chapter 3.

D.1.1 Attributes of Modeling Elements of CRS

The attributes of each modeling element of CRS metamodel is as follows:

1. **Customer Attributes**
   - `CustomerID`: This is the unique identification given to the Customer.
   - `CreditScore`: This is the credit score of the Customer used to determine eligibility for renting Car.
   - `ZipCode`: This is the zip code of the Customer for location.
   - `PhoneNumber`: This is the phone number on which the customer can be contacted.
2. **Representative Attributes**
   - `RepresentativeID`: This is the unique identification given to Representative.
   - `RepresentativeExperience`: This gives us the years of experience of the Representative.
   - `RepSalary`: This is the salary that the Representative earns.
3. **Car Attributes**
   - `CarID`: This is the unique identification given to Car.

- `Make`: This is the make of the Car.
- `Year`: This is the year in which the Car was manufactured.
- `Miles`: This is the total number of miles driven by the Car.
- `Model`: This is the model of the Car.
- `FuelLevel`: This is the fuel level of the Car at the time it is present in the Garage.

4. **Mechanic Attributes**
   - `MechanicID`: This is the unique identification given to the Mechanic.
   - `MecSalary`: This is the salary that the Mechanic earns.
   - `SkillLevel`: This represents the skill level of the Mechanic.
5. **CarInventory Attributes**
   - `CarInventoryID`: This is the unique identification given to the CarInventory which contains Cars.

D.1.2 Attributes of Modeling Elements of LMS

The attributes of each modeling element of LMS metamodel is as follows:

1. **Librarian Attributes**
   - `LibrarianID`: This is the unique identification given to the Librarian.
   - `LibrarianSalary`: This is the salary paid to the Librarian.
   - `JoinYear`: This is the year in which the Librarian joined working.
   - `JobTitle`: This is the functionality which the Librarian performs like Junior and Senior.
2. **Patron Attributes**
   - `Age`: This is the age of the Patron.
   - `Major`: This gives us the major of the Patron like Computer Science or Mechanical Engineering.

- `City`: This is the city that the Patron lives in.

3. **Book Attributes**

- `ISBN`: This is the ISBN given to the book.
- `Author`: This is the author of the Book.
- `Department`: This is the department related to the Book.
- `Quantity`: This is the number of the Books available in the Library.
- `Title`: This is the title of the Book.

4. **Shelf Attributes**

- `ShelfID`: This is the unique identification given to the Shelf.
- `ShelfType`: This is the type of the Shelf based on various categories.
- `Location`: This represents the location of the Shelf in the Library.

E MODELING EXERCISES GIVEN TO MODELERS

In this section, we discuss the modeling exercises given to the modelers in order to evaluate our system as discussed in the Chapter 6.

E.1 Modeling Exercise in CRS

The CRS modeling exercise was given to the modelers so that they would be familiar with using our recommender system. There was only one exercise involving CRS. The modeling exercise is as follows:

1. Create one Garage model contained by Rootfolder.
2. It should have 4 Mechanics, 10 Customers, 4 Representatives and 2 Car Inventories with 5 cars each.
3. 5 Customers rent 5 different Cars. Try to use both Car Inventories.
4. Representatives approve 7 customers. Try to use different Representatives.
5. Complete attributes for all modeling elements.

E.2 Modeling Exercises in LMS

The LMS modeling exercise was given to the modelers so that we can determine how the recommender system remembers and recommends the modeling actions based on the modeler activity. There were two exercises involving LMS.

- **First Modeling Exercise**:
  1. Create one Library model contained by Rootfolder.
  2. It should have 5 Librarians, 3 Shelves, 3 Patron References, 10 Patrons and 14 Books.

3. It should have 4 Patrons borrow 2 Books each, 3 Patrons borrow 1 Book each. Books can be same as there are multiple copies but try to make it different. Also, Patron must not borrow same Book twice.
4. It should have each Patron Reference refer to one Patron. Let 1 Patron Reference borrow 2 books and 1 Patron Reference borrow 1 book.
5. Only one Book must be not in a Shelf, rest in different Shelves.
6. Complete attributes for all modeling elements.

- **Second Modeling Exercise**:
  1. Create one Library model contained by Rootfolder.
  2. Have 3 Librarians, 2 Shelves, 2 Patron References, 5 Patrons and 7 Books.
  3. Have 3 Patrons borrow 2 Books, 2 Patrons borrow 1 book. Books can be same as there are multiple copies but try to make it different. Also, Patron must not borrow same Book twice.
  4. Have each Patron Reference refer to one Patron. Let 1 Patron Reference borrow 2 books and 1 Patron Reference borrow 1 Book.
  5. Only one book must be outside, rest in different shelves.
  6. Repeat for another Library Model in the same RootFolder.
  7. All connections must be in the same model of RootFolder.
  8. Complete attributes for all modeling elements.