Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Tanumoy Pati

Entitled
Auto-Generating Models From Their Semantics and Constraints

For the degree of Master of Science

Is approved by the final examining committee:

James H. Hill
Chair

Rajeev Raje

Mohammad Al Hasan

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): James H. Hill

Approved by: Shiaofen Fang 07/02/2012
Head of the Graduate Program Date

Graduate School Form 20
(Revised 9/10)

PURDUE UNIVERSITY
GRADUATE SCHOOL

Research Integrity and Copyright Disclaimer

Title of Thesis/Dissertation:

Auto-Generating Models From Their Semantics and Constraints

For the degree of Master of Science

I certify that in the preparation of this thesis, I have observed the provisions of *Purdue University Executive Memorandum No. C-22,* September 6, 1991, *Policy on Integrity in Research.**

Further, I certify that this work is free of plagiarism and all materials appearing in this thesis/dissertation have been properly quoted and attributed.

I certify that all copyrighted material incorporated into this thesis/dissertation is in compliance with the United States' copyright law and that I have received written permission from the copyright owners for my use of their work, which is beyond the scope of the law. I agree to indemnify and save harmless Purdue University from any and all claims that may be asserted or that may arise from any copyright violation.

Tanumoy Pati

Printed Name and Signature of Candidate

07/02/2012

Date (month/day/year)

*Located at http://www.purdue.edu/policies/pages/teach_res_outreach/c_22.html

AUTO-GENERATING MODELS FROM THEIR SEMANTICS AND CONSTRAINTS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Tanumoy Pati

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

August 2012

Purdue University

Indianapolis, Indiana

This work is dedicated to my family and friends.

ACKNOWLEDGMENTS

I am heartily thankful to my advisor, Dr. James H. Hill, for his encouragement, guidance and support throughout my research work. I am not only grateful to him for giving me the opportunity to work for him, but also for inspiring me to hone my skills every moment.

I also want to thank Dr. Rajeev Raje and Dr. Mohammad Al Hasan for agreeing to be a part of my Thesis Committee.

My gratitude is also extended to Dennis Feiock for helping me with the results section and Nicole Wittlief for helping me edit this manuscript.

Thank you to all my friends and well-wishers for their good wishes and support. And most importantly, I would like to thank my family for their unconditional love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABBREVIATIONS

DSML	Domain-Specific Modeling Language
GME	Generic Modeling Environment
MDE	Model Driven Engineering
GEMS	Generic Eclipse Modeling System
D&C	Deployment and Configuration
DSL	Domain-Specific Language
PME	Proactive Modeling Engine
EMF	Eclipse Modeling Framework
CLP	Constraint Logic Program
LMS	Library Management System
OCL	Object Constraint Language
AST	Abstract Syntax Tree
UML	Unified Modeling Language
OMG	Object Management Group
QVT	Query/Views/Transformations
PICML	Platform Independent Component Modeling Language
FCO	First Class Object
EBNF	Extended Backus Naur Form
DSEL	Domain-Specific Embedded Language

ABSTRACT

Pati, Tanumoy. M.S., Purdue University, August 2012. Auto-Generating Models From Their Semantics and Constraints. Major Professor: James H. Hill.

Domain-specific models powered using domain-specific modeling languages are traditionally created manually by modelers. There exist model intelligence techniques, such as constraint solvers and model guidance, which alleviate challenges associated with manually creating models, however parts of the modeling process are still manual. Moreover, state-of-the-art model intelligence techniques are—in essence—reactive (*i.e.*, invoked by the modeler).

This thesis therefore provides two contributions to model-driven engineering research using domain-specific modeling language (DSML). First, it discusses how DSML semantic and constraint can enable proactive modeling, which is a form of model intelligence that foresees model transformations, automatically executes these model transformations, and prompts the modeler for assistance when necessary. Secondly, this thesis shows how we integrated proactive modeling into the Generic Modeling Environment (GME). Our experience using proactive modeling shows that it can reduce modeling effort by both automatically generating required model elements, and by guiding modelers to select what actions should be executed on the model.

1 INTRODUCTION

Model-Driven Engineering (MDE) [1], powered by domain-specific modeling languages (DSMLs), allows developers to define the semantics of a given domain using intuitive graphical representations, and define constraints to govern the interactions of domain constructs. DSMLs are then used by modelers to model concepts for the target domain. For example, a librarian can use a library DSML to model books owned by their library, along with monitoring patron borrowing activity. Likewise, a system developer can use a component-based system DSML to model the interfaces and attributes of components in their system, the composition of the modeled components, and the deployment and configuration (D&C) of the modeled compositions. Lastly, model interpreters transform constructed models into concrete artifacts (*e.g.*, a book audit or complex D&C descriptor files). Examples of MDE tools that use DSMLs include: 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 using DSMLs to create models is primarily a manual process. This means that it is the responsibility of the modeler to manually craft and manage their models, by actions such as adding and deleting model elements, setting attributes, and ensuring constraints are not violated. Since creating a model can be a tedious and time-consuming process—especially when dealing with complex DSMLs and large models—model intelligence techniques (*e.g.*, constraint solvers [6–8] and model guidance [9, 10]) have emerged as approaches to alleviate this concern. For example, modelers can manually create a *partial model* and use constraint solvers to automatically generate a complete solution. Likewise, modelers can select a model element, and model guidance engines can either highlight valid associations (*e.g.*,

connections and references) with respect to the selected model element, or provide valid editing operations that a modeler can perform.

Although model intelligence is improving the usability of DSMLs, it is still plagued by manual processes. As highlighted above, it is the responsibility of the modeler to manually create a partial model before constraint solvers can be invoked. Likewise, model guidance techniques engage the modeler only after they have made a selection. It can, however, be difficult for the modeler to know what actions can occur next—especially if the modeler is not familiar with the DSML. Further, "fixing" a model implies the modeler has to first create a model; this is typically a manual process completed through trial-and-error, even with current state-of-the-art model guidance techniques. Finally, it is the responsibility of the modeler to manage model consistency and correctness above and beyond manually, or even automatically, evaluating constraints after completing actions (*i.e.*, reactive constraint checking).

Due to these challenges, there is need for improved model intelligence techniques that better assist modelers in the modeling process. Based on this understanding, the main contributions of this thesis are as follows:

- It introduces proactive modeling, which is a form of model intelligence that foresees plausible model transformations, executes these model transformations automatically, and prompts the modeler for assistance when needed; and

- It shows how proactive modeling is implemented in GME as a GME add-on (*i.e.*, a domain-independent event handler) named the *Proactive Modeling Engine (PME)*.

Our experience in applying the PME to a simple DSML and other DSMLs shows that it can reduce modeling effort by automatically generating required model elements, and also guide modelers to select what actions to execute on a model. It is, however, necessary to provide mechanisms that allow modelers to control how engaged proactive modeling is with the model and modeler.

2 RELATED WORKS

This section compares our work on proactive modeling to other works; more specifically, we compare our work with others from the areas of partial model creation, decision making, and semantic- and constraint-driven automation.

Partial model creation. Sen et al. [8] presented a framework for generating model completion recommendations in model editors. In their approach, the metamodel is transformed into a *constraint logic program* (CLP) [8], and is processed by a Prolog engine. The processed CLP is then able to complete a partial model (*i.e.*, one that has been manually created by the modeler). Our approach extends their effort in that proactive modeling can assist in either automatically creating the partial model, or recommending what actions to take on the model. Once the modeler has a valid partial model created using PME, the modeler can use their approach to complete it.

Hessellund et al. [7] created an extension of the Eclipse Modeling Framework called *SmartEMF*. SmartEMF provides support for representing, checking, and maintaining four kinds of consistency constraints: well-formedness of individual artifacts, referential integrity across artifacts, references with additional constraints, and style constraints. Similar to Sen et al., SmartEMF provides editing guidance to the modeler by evaluating precondition constraints that exist on editing operations. Our work therefore extends the work done by Hessellund et al. in that it can not only provide modeling guidance to modelers but it can also automatically perform model transformations, such as automatically adding/deleting of valid model elements in accordance with the constraints.

White et al. [6] created a Domain-Specific Intelligence Framework (DSIF) that provides model guidance for large and complex models. In their approach, a DSIF is first created for a DSML, which is then used to parametrize a Prolog knowledge base for each model, utilizing the entities and roles specified in the metamodel. DSIF also

allows modelers to specify domain-specific constraints in Prolog that can complete a partially specified model through modeling suggestions. Our approach extends their effort in that it can assist with creating the partial model–which is currently done manually–that is needed to auto-generate the complete model.

Decision making. Janota et al. [10] improved modeling experience with their work on Interactive model derivation, which is a process of constructing models and metamodels with the help of automatic adaptive guidance. This guidance system assists a modeler by providing a list of valid edit operations to choose from. The guidance algorithms, developed for concrete modeling languages, identify transformations that *refine* the model and provide suggestions to the modeler. Our approach extends their work in that proactive modeling can not only provide decision-making capabilities but also auto-generate model elements when a model element is first created (*i.e.*, automatically perform multiple editing operations). Moreover, proactive modeling also provides modelers with a sequence of valid operations to choose from after it has finished auto-generating model elements.

Constraint-driven model intelligence. White et al. [9] developed a model intelligence mechanism that guides modelers towards correct models. In their approach, the modeler first selects a relationship type and an element for the new relationship. The model intelligence then evaluates constraints associated with the selected element and presents a list of valid elements that can be associated with the selected element. Our work extends the work done by White et al. in that proactive modeling automates the modeling process based on the metamodel's semantics and constraints, rather than simply its constraints. Once the proactive modeling reaches a point where it needs human intervention, it prompts the modeler for the next action. At that point, our approach is similar to the approach of White et al.

3 A RUNNING EXAMPLE: THE LIBRARY MANAGEMENT SYSTEM

This chapter introduces the Library Management System (LMS) example, which is a system that assists librarians in tracking the book inventory in their library and monitoring the book borrowing process of the patrons.

3.1 Overview of the LMS

This section provides a detailed discussion of the core components of the library management system: the metamodel or the modeling elements, constraints and the model itself.

3.1.1 Modeling elements

There are multiple ways to compose a metamodel for the LMS; Figure 3.1 shows one simple example metamodel for the LMS. As shown in this figure, the root element is a Library model element. This Library model element contains four basic model elements: `Book`, which represents books present in a library; `Patron`, which represents people who borrow books from the library; `Librarian`, which represents librarians working in the library; and `HRStaff`, which represents the human resource staff that hires a librarian. The Library model element also contains another model element named Shelf, which represents the shelf location of a book in the library.

The Book model element has the following attributes:

- **Title** – a single-line text field that allows the modeler to specify the book's title;
- **Author** – a multi-line text field that allows the modeler to list the authors of the book;

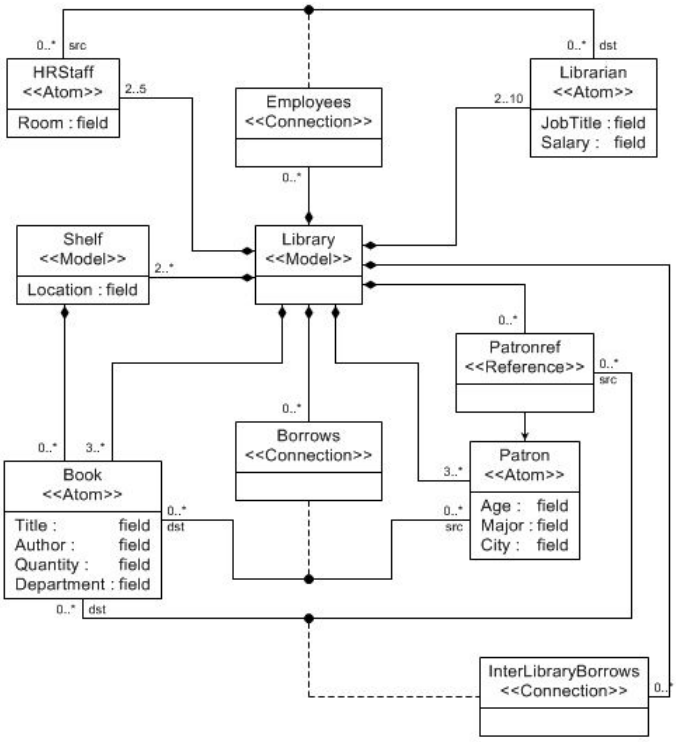

Figure 3.1. An example GME metamodel for the Library Management System.

- **Quantity** – a numerical field that allows the modeler to set the number of book copies owned; and
- **Department** – a single-line text field that allows a modeler to categorize books by a specific department, such as Computer Science and Electrical Engineering.

Likewise, the Patron model element has the following attributes:

- **Age** – a numerical field that captures the patron's age;
- **Major** – a single-line field that captures the patron's major, such as Computer Science or Biology; and
- **City** – a single-line field that contains the patron's city.

Similarly, the Librarian model element has the following attributes:

- **JobTitle** – a single-line text field that captures the librarian's job title, such as Junior Librarian, Senior Librarian, or Book Inventory Manager; and
- **Salary** – a numerical field for specifying the librarian's salary.

The HRStaff model element has the following attributes:

- **Room** – a numerical field that shows the room number of the human resources staff member.

Lastly, the Shelf model element contains a single attribute named `Location`, which represents the location of the book shelf in the library.

The Library model element also contains `Borrows` connection elements such that a connection between a Patron and a Book means the patron borrowed the connected book from the library. It also contains `Hires` connection elements such that a connection between a Librarian and HRStaff means that the librarian was hired by the connected HR staff. The Library model element can also contain `Patronref` model elements that refer to patrons belonging to other libraries. This allows the modeler to model patrons borrowing books from another library. Therefore, the Library model also consists of an `InterLibraryBorrows` connection between Patronref and Book elements.

3.1.2 Constraints

The LMS has several constraints that govern its model. Since the LMS is created in GME, the Object Constraint Language (OCL) [11] is used to express the LMS's constraints. The constraints of the LMS are as follows:

- **Minimum number of books required.** This constraint is used to enforce the minimum number of books that a library must contain. As shown in Figure 3.2, this constraint checks that a Library model element has at least 3 Book model

elements. The constraint shown in this figure is automatically generated by GME from the cardinality expressed on the containment connection between the Library and Book model elements in Figure 3.1.

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

Figure 3.2. OCL constraint showing minimum number of books required.

- **Minimum number of patrons required.** This constraint is used to enforce the minimum number of patrons that a library must contain. As shown in Figure 3.3, the constraint checks that a Library model element has at least 3 Patron model elements. The constraint shown in this figure is automatically generated by GME from the cardinality expressed on the containment connection between the Library and Patron model elements in Figure 3.1.

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

Figure 3.3. OCL constraint showing minimum number of patrons required.

- **Minimum number of shelves required.** This constraint is used to enforce the minimum number of shelves that a library must contain. As shown in Figure 3.4, the constraint checks that a Library model element has at least 2 Shelf model elements. The constraint shown in this figure is automatically generated by GME from the cardinality expressed on the containment connection between the Library and Shelf model elements in Figure 3.1.

- **Number of HR staff required.** This constraint checks that a Library model contains at least 2 and at most 5 HRStaff elements as shown in Figure 3.5. The

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

Figure 3.4. OCL constraint showing minimum number of shelves required.

constraint shown in this listing is automatically generated by GME from the cardinality expressed on the containment connection between the Library and HRStaff model elements in Figure 3.1.

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

Figure 3.5. OCL constraint showing the number of required HR staff.

- **Number of librarians required.** This constraint checks the minimum and maximum number of librarians that work at the library. As shown in Figure 3.6, the constraint checks that a Library model element has at least 2 and at most 10 Librarian model elements. The constraint shown in this figure is automatically generated by GME from the cardinality expressed on the containment connection between the Library and Librarian model elements in Figure 3.1.

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

Figure 3.6. OCL constraint showing the number of required librarians.

- **Required city.** This constraint checks that all patrons who are members of the library are from a pre-determined city. As shown in Figure 3.7, all patrons must be from Indianapolis in order to join the library. This is a domain-specific constraint that is added manually by the person who created the metamodel.

```
self.City = "Indianapolis"
```

Figure 3.7. OCL constraint showing the required city.

- **Required age.** This constraint checks the age of each patron that is a member of the library. As shown in Figure 3.8, a patron must be at least 18 years of age. This is a domain-specific constraint that is added manually by the person who created the metamodel.

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

Figure 3.8. OCL constraint showing the required age.

- **Salary range.** This constraint checks that a librarian's salary is within the accepted salary range. As shown in Figure 3.9, the salary should be in the range $30,000 to $40,000. This is a domain-specific constraint that is added manually by the person who created the metamodel.

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

Figure 3.9. OCL constraint showing the salary range of a librarian.

- **Book borrowing condition.** This constraint validates that a patron can only borrow books that are relevant to his/her field. For example, a Computer Science student can only borrow books that are relevant to the field of Computer Science. As shown in Figure 3.10, this constraint checks the patron's major against the book's department. This is a domain-specific constraint that is added manually by the person who created the metamodel.

```
let conn = self.connectedFCOs(Borrows) in
conn->forAll(p:Book | self.Major = p.Department)
```

Figure 3.10. OCL constraint showing the book borrowing condition.

- **Book borrowing limit.** This constraint validates that a patron can only borrow a certain number of books. As shown in Figure 3.11, a patron can borrow only 5 books from the library.

```
let conn = self.attachingConnections(Borrows)->size in
conn <= 5
```

Figure 3.11. OCL constraint showing the book borrowing limit.

- **Patron referencing condition.** This constraint checks that it refers to a patron that belongs to another library, as shown in Figure 3.12. This is a domain-specific constraint that is added manually by the person who created the metamodel.

```
self.refersTo().parent () < > self.parent ()
```

Figure 3.12. OCL constraint showing patron referencing condition.

- **Inter-library book borrowing condition.** This constraint validates that a patron from another library (*i.e.*, Patronref) can only borrow books that are relevant to their field. For example, a Patronref element referring a Computer Science patron can only borrow books that are relevant to the field of Computer Science. As shown in Figure 3.13, this constraint checks the referring patron's major against the book's department. This is a domain-specific constraint that is added manually by the person who created the metamodel.

```
let p:Patron = self.refersTo () in
let conn = self.connectedFCOs () in
conn->forAll (b:Book | b.Department = p.Major)
```

Figure 3.13. OCL constraint showing the inter-library book borrowing condition.

- **Inter-library book borrowing limit.** This constraint validates that a patron belonging to other libraries can only borrow a certain number of books. As shown in Figure 3.14, a patron can borrow only 2 books from the library.

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

Figure 3.14. OCL constraint showing the inter-library book borrowing limit.

3.1.3 An example model

Using the metamodel for the LMS discussed in the previous sections, modelers (*e.g.*, librarians) can model all the books in the library. Likewise, modelers can model patrons and the books they borrow from the library. Figure 3.15 shows an example model for the LMS illustrating this usage.

As shown in this figure, the Library model element contains both Book and Patron model elements. Likewise, the box in the lower right-hand corner of Figure 3.15 shows the attributes for the selected Patron element named *Tanumoy*. The connections between the Patron elements and the Book elements represent the patron borrowing the associated book, and the connections between a Librarian model element and a HRStaff model element represent the HR staff member hiring the connected librarian.

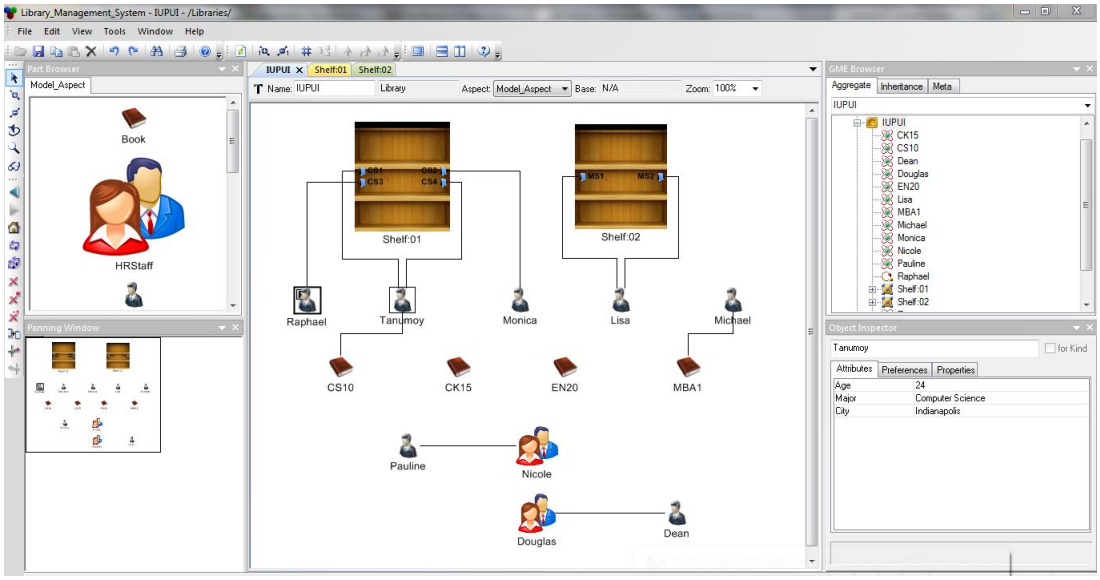

Figure 3.15. An example model for the Library Management System.

3.2 Current Limitations in Context of the LMS

It is possible to use current state-of-the-art approaches in model intelligence to assist with creating this model. For example, model guidance can be used to highlight which books a patron can borrow when the modeler selects a particular patron. Likewise, it is possible to use model guidance to assist with correcting a model that has a constraint violation, such as creating too many connections from a patron to a book.

There is, however, another aspect of this example that is not addressed by current model intelligence techniques: the process of proactively managing the model, and actions that occur on it. For example, a Library must contain certain elements, such as 3 Books and 2 Patrons. When a Library model element is created, it should automatically contain these elements (*i.e.*, the modeler should not have to prompt the model intelligence to engage with the model). Likewise, the modeler has many *valid* actions to select from, which can be inferred from the metamodel; this is because the metamodel is static and well-defined.

4 METHODOLOGY

This chapter puts forth the theoretical and implementation aspects of achieving proactive modeling. Section 4.1 provides a detailed overview of proactive modeling and Section 4.2 discusses the design and implementation of proactive modeling in GME.

4.1 An Overview of Proactive Modeling

This section provides a detailed overview of proactive modeling in DSMLs. It discusses the aspects that are to be automated in the modeling process, the process for achieving proactive modeling, and the role of mutable and immutable constraints in proactive modeling.

4.1.1 The Goal of Proactive Modeling

The term *proactive modeling* translates directly to foreseeing modeling. The main goal of proactive modeling therefore is to automate—as much as possible—the modeling process by foreseeing valid model transformations (*i.e.*, those that must be executed manually by a modeler), and automatically executing them. If there are optional model transformations, proactive modeling then queries the modeler for what model transformation to execute, and executes the selected one (similar to model guidance).

Proactive modeling therefore helps reduce modeling effort since it removes many tedious and error-prone modeling actions from the modeling process, such as manually creating required model elements. Likewise, proactive modeling can assist modelers who are not familiar with a DSML. With this in mind, proactive modeling focuses on automating the following aspects of the modeling process:

1. **Automated model creation.** This aspect of proactive modeling involves automatically creating different model elements when a related model element is first created. For example, when a Library model element is added to the model, all its child model elements (*e.g.*, Book, Patron, and Librarian) should be automatically added to the Library model element, up to the required quantity. This is different from current model intelligence techniques in that they do not support auto-generating elements in the required quantity, or they do not support auto-generation at all, unless the modeler manually creates a partial model.

2. **Decision-making.** This form of proactive modeling involves presenting the modeler with a list of valid model transformations or actions (*e.g.*, create a connection, add a reference, or add a new model element) that can occur based on the current state of the model. After selecting an action, proactive modeling executes the action. For example, when a modeler wants to add a Patronref model element, proactive modeling presents the modeler with a list of all the possible Patrons that the Patronref model element can reference. Upon selecting a Patron model element, the reference is auto-generated. This form of automation–which requires human-intervention–is different from current model intelligence techniques in that it is triggered automatically when automated modeling reaches a stopping point.

Figure 4.1 provides a high-level overview of the proactive modeling process where proactive modeling resides between the modeler and the model. A proactive modeling engine automatically adds modeling elements to the model (1 in the figure); when the proactive modeling engine reaches a stopping point, it then interacts with the modeler to select which transformations to apply to the model (2 in the figure). Ultimately, the modeler does not interact directly with the model; instead, the modeler interacts with the model through the proactive modeling engine.

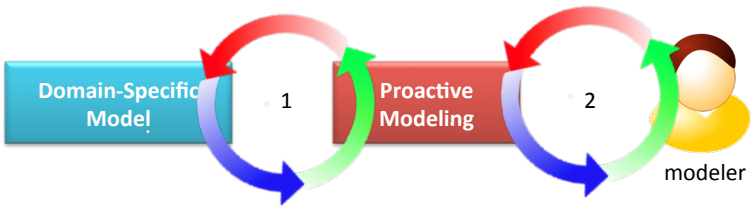

Figure 4.1. Conceptual overview of the proactive modeling process.

4.1.2 Insights for Realizing Proactive Modeling

As explained in the previous section, proactive modeling handles two types of automation. In order for proactive modeling to perform automation, however, it must obtain information about the model . Since a DSML is well-defined, it is possible for proactive modeling to gain its insights from a DSML's metamodel. More specifically, it can gain its insight from the following types of analysis of a DSML's metamodel:

- **Semantic analysis.** Semantic analysis is the process of analyzing a DSML's metamodel at run-time to discover information about its model elements. For example, when adding a Patronref element to the model, semantic analysis of the LMS metamodel (see Figure 3.1) will identify that a Patronref model element can reference a Patron model element. By performing semantic analysis, proactive modeling can collect any type of information that is relevant to a model element without being bound to the target DSML.

- **Constraint analysis.** Constraint analysis is the process of parsing and analyzing a DSML's constraints collected during the semantic analysis process. For example, semantic analysis of Patronref returns the constraint shown in Figure 3.12, which is then parsed and evaluated to generate the list of possible Patrons that can be referenced. By performing constraint analysis, proactive modeling can not only evaluate constraints, but also use them to provide modeling guidance and auto-generate model elements.

Other types of analysis can be integrated into proactive modeling, such as layout analysis where actions are performed based on the current layout of modeling element, and user-intent analysis where actions are performed based on past knowledge of how a modeler creates a model, but we have limited our work to these two forms of analysis. This is because both semantic and constraint analysis are based on well-defined information that is static; they can also provide a foundation for building other types of analysis previously mentioned.

4.1.3 Mutable vs. Immutable Constraints

As explained above, it is possible to analyze a DSML's constraints and determine what elements should be added to the model, or generate a list of valid modeling actions. For example, specifying that the number of Patrons must equal 3 means that proactive modeling can automatically ensure the number of patrons is always 3 since 3 does not change. On the other hand, saying that the number of patrons must equal the number of books means that proactive modeling needs modeler intervention because both the number of patrons and books can be modified.

Based on the two examples above, constraints can be classified as either *mutable* or *immutable*. A mutable constraint is a constraint that evaluates two variable expressions. An immutable constraint is a constraint that evaluates a variable expression and a constant expression. Since an immutable constraint has constant values, it is possible to automatically execute actions that transform the model towards the constant value. Mutable constraints, on the other hand, require modeler intervention because the modeler must select the model element that acts as the constant value in the constraint evaluation.

4.2 The Design and Implementation of Proactive Modeling in GME

This section discusses how proactive modeling is realized in GME via the *Proactive Modeling Engine*.

4.2.1 Mapping Proactive Modeling into GME

As explained in Section 4.1.2, semantic analysis allows proactive modeling to learn the elements of a DSML and their interactions; constraint analysis allows proactive modeling to learn how to govern the interactions between elements. Both types of analysis can be integrated into GME at run-time without being bound to a specific DSML. Before describing the implementation details of proactive modeling in GME, it is first necessary to understand how the analysis maps into GME—especially the constraint analysis. Based on the functionality of GME, we have classified constraints into the following four categories:

- **Containment constraints.** GME allows modelers to define *multiplicity* specification (also known as cardinality) on containment relationships. The multiplicity specification determines the acceptable number of containment relationships allowed between a parent model element and a child element. For example in Figure 3.1, the containment relationship between Book model element and Library model element has a multiplicity of `3..*`. This means that a Library model element should have at least 3 Book model elements at all times.

- **Attribute constraints.** GME allows modelers to define constraints that validate attribute values with respect to expected values or other elements. For example, Figure 3.7 illustrates that the expected city for a patron is "Indianapolis". Likewise, Figure 3.8 shows that the expected age of a patron is at least 18.

- **Association constraints.** GME allows modelers to define association relationships between two model elements using a Connection model element. For

example, Figure 3.1 shows three Connection elements: Borrows, which associates Book model elements with Patron model elements; Hires, which associates HRStaff model elements with Librarian model elements; and InterLibraryBorrows, which associates Patronref model elements with Book model elements. Modelers can refine association relationships using constraints and these constraints determine the possible destination model elements of a connection. For example, Figure 3.10 shows an association constraint imposed on Patron that governs the connection between a Book and Patron model element.

- **Reference constraints.** GME allows modelers to define aliases (or pointers) to other model elements by using the Reference model element. For example, Figure 3.1 shows how the LMS metamodel defines a Patronref model element that refers to Patron model elements. Modelers can also impose constraints on references, which validate that the referenced model element meets a condition. For example, Figure 3.12 shows a reference constraint imposed on Patronref, which specifies that a Patronref can only reference Patrons from another library.

4.2.2 The Proactive Modeling Engine

Based on the understanding of how to map proactive modeling into GME, Figure 4.2 provides an overview of the Proactive Modeling Engine (PME), which implements proactive modeling. As shown in this figure, the PME is a GME add-on; a GME add-on is a domain-independent event handler that receives events dictating what model actions have occurred. For example, when a new model element is added to the model, all loaded add-ons then receive the `OBJEVENT_CREATED` event and the event provides a reference to the newly created object. Likewise, when a model element is selected, all loaded add-ons receive the `OBJEVENT_SELECTED` event and the event provides a reference to the selected object. It is worth noting that if an add-on

modifies the model, then the event that corresponds to the modification is sent to all loaded add-ons—including the add-on that modified the model. Lastly, all GME add-ons are stateful.

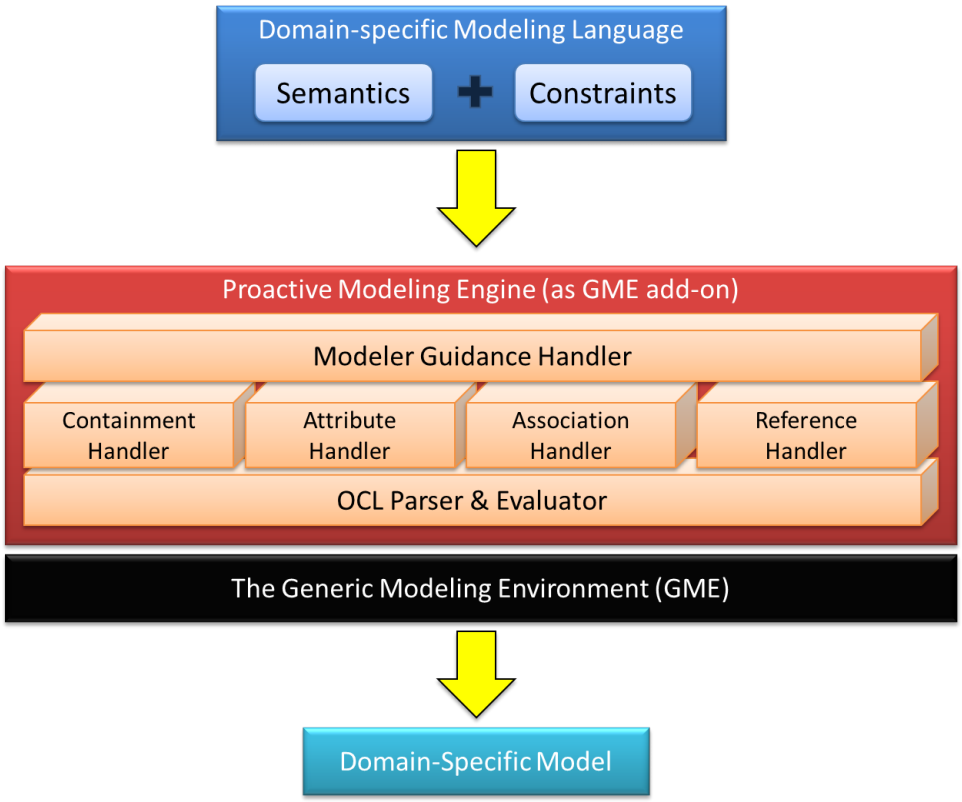

Figure 4.2. Architectural overview of the Proactive Modeling Engine.

As shown in Figure 4.2, the PME is composed of the following key components:

- **OCL parser and evaluator.** The OCL parser is responsible for parsing OCL constraints and dynamically creating an abstract syntax tree from the parsed OCL constraints. Since a GME add-on is stateful, the parsed OCL expressions are cached for later retrieval. The OCL parser in the PME is designed and implemented using the *Boost Spirit Parser Framework* (`boost-spirit.com`), which is an object-oriented recursive descent parser generator framework implemented using expression templates and template meta-programming techniques [12].

This parser works only with constraints defined in the DSML and is independent of the DSML's metamodel. This allows the OCL parser to be used as a standalone OCL parser for other application domains.

The OCL evaluator for the PME works as follows: First, the OCL evaluator is invoked by handlers on the root node of the abstract syntax tree (AST). The individual objects that form the AST are responsible for evaluating a certain aspect of the constraint (*e.g.*, a method or expression). The evaluation control then traverses the AST in a top-down fashion and each object returns the evaluated result back to its parent, stopping at the root. Based on the evaluated value and the information collected during semantic analysis, the PME transforms the model accordingly.

A detailed discussion of the structure of the OCL parser and evaluator has been presented in Appendix C.

OCL vs. Prolog. Prolog is a declarative programming language that is based on the logic programming paradigm. In Prolog, program logic is expressed in terms of relations, and a computation is initiated by running a query over these relations. Prolog has been widely used in modeling intelligence [6–8]. Model elements can be transformed into facts in Prolog, which can be checked using Prolog rules and queried with Prolog queries [13].

The OCL, on the other hand, is a declarative language for describing rules that apply to Unified Modeling Language (UML) models. OCL is an Object Management Group (OMG) (`www.omg.org`) standard for constraint specification and is the most widely used constraint specification language [14]. It also acts as the basis for a number of different languages [15] such as Query/Views/-Transformations (QVT) standard. Moreover, OCL is widely supported by both commercial and non-commercial modeling tools [13].

Opoka et al. [13] compared the performance of query engines that used Prolog and OCL. Their experiments show that queries collecting, or selecting, elements

based on direct properties (*e.g.*, determining model size) have an evaluation time that is linear for both Prolog and OCL, but Prolog is faster. For complex queries based on properties of relationships between elements, however, OCL performs significantly better than Prolog. More specifically, the evaluation time in OCL is linear, while in Prolog it is quadratic. This is because OCL allows hierarchical representation (reflecting original structure) of models together with navigation abilities, and Prolog uses a non-hierarchal list representation of models.Taking into account the results presented by Opoka et al. involving performance, we parse and evaluate OCL expressions directly instead of translating them into Prolog rules since the PME operates in real-time.

- **Containment handler.** The containment handler is responsible for automating the model element creation process by resolving the containment relationships between model elements. For example, when a Library model element is added to the example model shown in Figure 3.15, the containment handler first analyzes the LMS's metamodel to identify what model elements a Library model can contain. In this case, the containment handler will identify the Book, Patron, Borrows, Shelf, HRStaff, Librarian, Hires, and Patronref model elements. This is the semantic analysis portion of proactive modeling.

  After the containment handler completes its semantic analysis, the containment handler collects each constraint associated with the newly created model element, and forwards it to the OCL parser and evaluator. If a constraint is a containment constraint, and is violated, then the containment handler auto-generates the model element associated with each constraint until all containment constraints are valid. This is the constraint analysis portion of proactive modeling. Using the LMS example, when a modeler add a Library model element to the model, then the PME will auto-generate 3 Book, 3 Patron, 2 Shelf, 2 HRStaff, and 2 Librarian model elements, as shown in Figure 4.3.

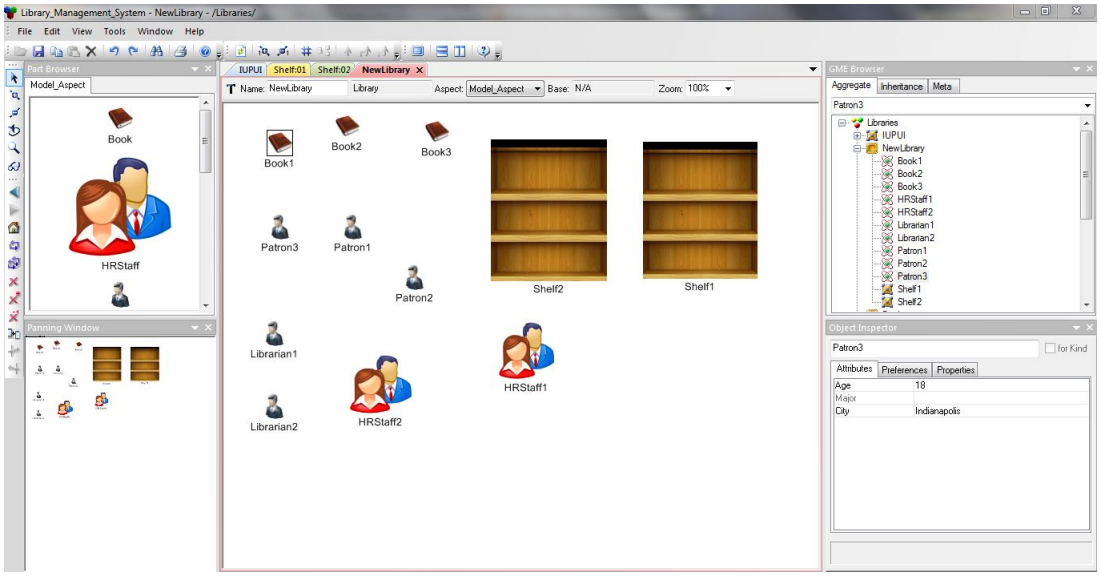

Figure 4.3. Example of containment handler auto-generating other model elements when a new Library model element is added to the model.

- **Attributes handler.** The attributes handler is responsible for handling a model element's attribute values during the creation process, *i.e.*, ensuring the created object does not violate any attribute constraints. This, however, does not mean that a modeler cannot change an attribute's value after the model has been created.

  In the context of the LMS, when a Patron model element is added to the model shown in Figure 3.15, the attributes handler first analyzes the LMS's metamodel to identify its attributes (*i.e.*, semantic analysis). The attribute handler then collects the constraints associated with the Patron model element and forwards them to the OCL parser and evaluator. The attributes handler, however, evaluates only the attribute constraints associated with Patron model element (shown in Figure 3.7, Figure 3.8, and Figure 3.9). In this example, the value of the City attribute is automatically set to "Indianapolis" and the value of the Age attribute is set to 18. The lower right window in Figure 4.3 shows that the value of these two attributes for *Patron3* Patron model element

is automatically set when it is added to the model. Likewise, when a Librarian model element is added to the model, the value of the Salary attribute is set to \$30,000 (*i.e.*, the minimum allowed salary per the constraint in Figure 3.9).

- **Association handler.** The association handler is responsible for identifying valid destination model elements for a given source model element when making a connection between two model elements. For example in Figure 3.15, to create a connection between a Patron model element and a Book model element, the modeler first selects a Patron model element (*e.g.*, *Tanumoy*). The selection of *Tanumoy* then triggers the association handler, which first analyzes the meta-model to identify all valid connection types associated with the selected model element. This list, if any exists, is presented to the modeler. Once the modeler selects a connection type, the association handler identifies all valid endpoint model elements for the selected connection type. In this example, when *Tanumoy* is selected, the association handler auto-selects the Borrows type, and then returns a list of Book model elements (*i.e.*, CS1, CS2, CS3, CS4, CS10, MS1, MS2, CK15, EN20, and MBA1).

  The handler then collects, one at a time, the constraints associated with Patron model element (*i.e.*, the source model element) and forwards them to the OCL parser. The association handler subsequently evaluates the association constraints, which allows it to filter any model elements that will violate its constraints. In this example, the connection handler will filter out books that are not relevant to the *Tanumoy* patron (a Computer Science student), which are MS1, MS2, CK5, EN20, and MBA1. Lastly, Figure 4.4 shows how the PME displays a list of valid destination model elements. This is similar to current state-of-the-art techniques in model guidance.

- **Reference handler.** The reference handler is responsible for identifying valid model elements that can be referred to by a reference model element. For example in Figure 3.15, when a modeler adds a Patronref model element to

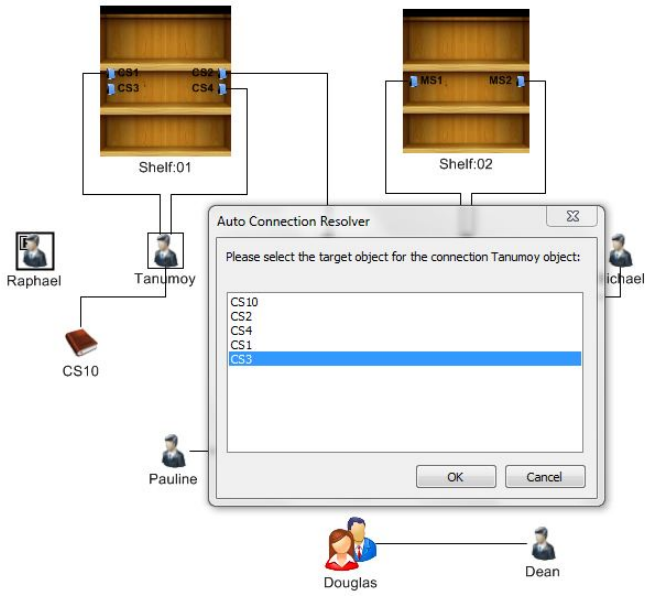

Figure 4.4. Example of association handler presenting the modeler with a list of valid Book model elements when the modeler selects a Patron model element.

the model, the reference handler first analyzes the LMS's metamodel, and then gathers a list of valid model element types that can be referenced by a Patronref model element. In this example, the reference handler will have a list that contains Patron model element type. The reference handler then uses the type information to gather a list of all elements that are of the identified model element type. The reference handler will, therefore, list the following Patron model elements: Tanumoy, Monica, Lisa, Michael, Joe, Raphael and Sam.

The handler then collects, one at a time, the constraints associated with selected reference model element and forwards them to the OCL parser and evaluator. The reference handler evaluates each OCL constraint with the goal of filtering the initial list of plausible model elements such that no element in the final list violates any constraints. Figure 4.5 shows an example of how the PME displays the Patron set when the modeler adds a Patronref model element to the model.

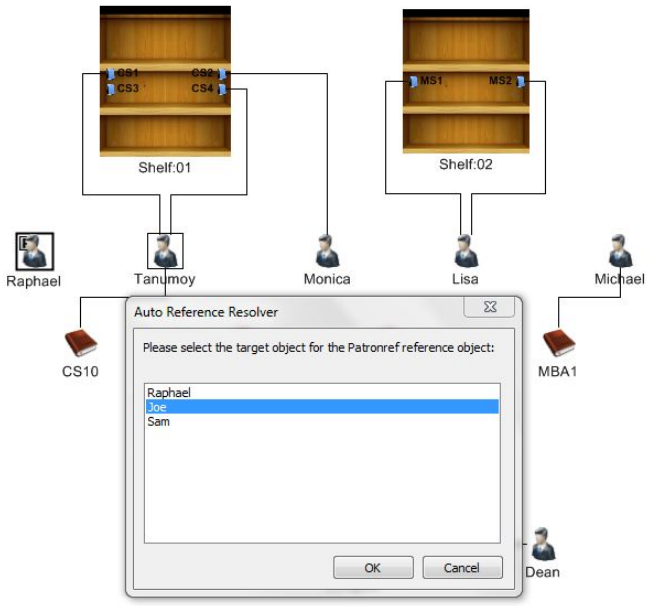

Figure 4.5. Example of reference handler showing the valid list of Patrons when a Patronref model element is added to the model.

As shown in Figure 3.12, this PME presents to the modeler a list of Patron model elements that are from a different library (*i.e.*, Joe, Raphael, and Sam).

- **Modeler guidance handler.** The modeler guidance handler is responsible for providing a modeler with a list of possible operations to choose from when the proactive modeling engine has finished auto-generating model elements. The operations that are presented to the modeler are in compliance with both the semantics and the constraints of the DSML. For example, when a modeler starts a new project, the modeler guidance handler presents the modeler with the list of all the model elements that can be added to the `RootFolder`. Likewise, the modeler guidance handler prompts the modeler to select a model to operate. Upon selection, the modeler is presented with a list of operations that are specific to the selected model.

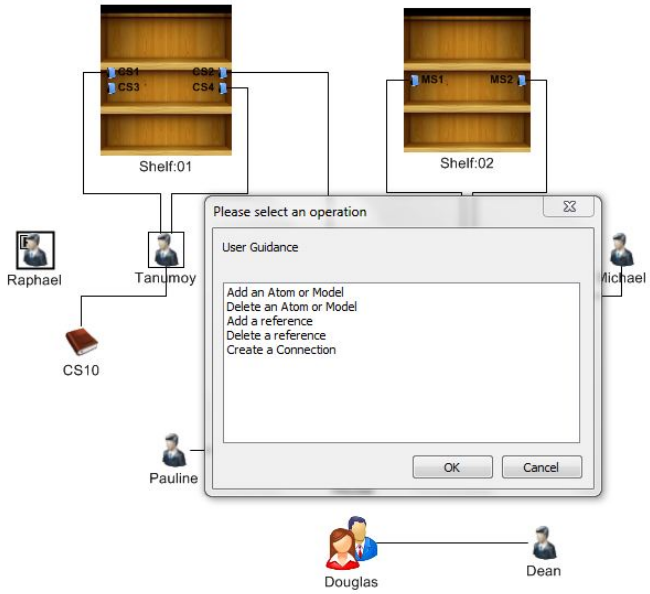

Figure 4.6. Example of modeler guidance handler showing the list of valid operations for Library model element.

Figure 4.6 shows the list of operations that a modeler can select from for the `IUPUI` model. The complete list of modeler guidance operations currently supported by the PME is as follows:

- **Add a model element.** This operation is used to add a model element to a parent model. When this operation is selected, the PME presents the modeler with a list of all the model element types that can be added to the parent model (*e.g.*, Book, Patron, HRStaff, Shelf, and Librarian model element). After the modeler selects the model type, the PME passes control to the containment and attributes handler to complete the automation process.
- **Add a reference model element.** This operation is used to add a reference model element to the selected parent, if allowed. When this action is selected, the PME provides the modeler with a list of all reference model element types that can be added to the parent element. When the

modeler selects the model element type, the PME passes control to the reference handler to complete the automation process.

- **Delete a model element.** This operation is used to delete a model element from the selected parent model element, if deleting a model element is allowed.
- **Create a connection.** This operation is used to create a connection between two model elements within the selected parent model element. When this operation is selected, the PME presents the modeler with a list of valid connection types that can be added to the parent model. When the modeler selects the connection type, the modeler is then presented with a list of valid source model elements that can be associated with the selected connection type. Finally, after the model selects the source model element, the PME passes control to the association handler to complete the automation process.

The modeler guidance handler therefore provides relevant and valid operations to modelers, which reduces the modeler's decision set and can improve their modeling experience. Likewise, we can easily extend the modeler guidance handler to support other operations as we learn about them.

4.2.3 Chain Reactions in PME

As stated above, the PME is a GME add-on, and a GME add-on in turn is a re-entrant component. This means that when the PME modifies the model, the PME will receive an event associated with the latest modification. Upon receiving the new event, the PME handles the new event similar to how it handled the previous event. If there is no decision-making need on the modeler's part, then the PME automatically handles the event (per the discussion above). If the PME requires user input, then it queries for it and proceeds accordingly.

In the best case scenario, the first model modification (*e.g.*, starting a new project, or opening an existing project) triggers the PME and the PME auto-generates all the elements required in the model. In this scenario, modeling effort is very low since the modeler's responsibility is reduced. In the worst case scenario, the modeler is prompted by the PME after each modification to the model since the PME is not able to automatically generate anything. In this scenario, the PME is similar to manually creating a model except for the fact that the PME ensures the modeler only executes valid actions.

5 RESULTS

This chapter contrasts the modeling effort of a modeler with/without PME for LMS and PICML DSMLs.

5.1 Modeling effort in DSMLs

Hill [16] defined `modeling effort` as the number of actions taken by a modeler to complete a high-level modeling goal or task. For example, in LMS a high-level modeling goal is for a patron to borrow books from a different library. This high-level goal consists of several actions:

- Representing the patron in the different library.
- Deduce the books that can be borrowed by this patron.
- Associate a specific book with this patron.
- Validate the borrowing conditions associated with the patron.

In modeling terms, the actions mentioned above relate to activities such as adding a model element, updating an existing element, setting attribute values for model elements, and associating model elements. In the software engineering domain [17] [18], tasks that require a human operator (*e.g.*, modeling) are measured with respect to two segments: measurable computer portion and variant human portion. The human portion of the task is referred to as *think time* [17] [18] and it is represented by a non-negligible variable Z. The amount of think time associated with each action depends heavily on the experience of the modeler.

Using the principles mentioned above, Hill [16] presented an equation to measure modeling effort $M(T)$ as:

$$M(T) = \sum_{t \in T} Z_t + time(t) \tag{5.1}$$

As shown in Equation 5.1 the modeling effort for each action t in task T is measured as the summation of the think time for a given action Z_t and the time taken by the computer to complete its given portion of the task, *i.e.*, $time(t)$. Therefore, $M(n)$ is interpreted as "the modeling effort of the task is n actions" [16]. Table 5.1 shows the modeling effort calculated for some tasks that can be performed during the modeling process.

Table 5.1
Modeling effort for typical tasks which can be performed in GME.

Task	**Modeling effort**	
	without PME	**with PME**
Adding a model element such as Model, and Atom	$M(1)$	$M(1)$
Setting a reference	$M(mn+2)$	$M(n+1)$
Creating a connection between two elements	$M(n+3)$	$M(2)$

As shown in Table 5.1, the modeling effort for adding a model element such as Model, Atom, and Reference is $M(1)$ with/without using PME. However, when a modeler wants to set a reference without using PME they need to perform three actions. First, they have to find the model element that they want to refer; the modeling effort for this is $M(n)$, since the modeler must go through m models and n model elements per model to find the valid element. Second, they must select the model element; the modeling effort for this is $M(1)$. Finally, drag the selected element back onto the reference element; the modeling effort for this is $M(1)$. The

total modeling effort for setting a reference is, therefore, $M(mn + 2)$. On the other hand, when the modeler is guided by PME, the modeling effort is simply $M(n + 1)$ because now the modeler is presented with a list of all the valid elements that can be referenced. The modeler has only to select a particular element and the reference is created, the modeling effort for which is $M(n)$. However, if there exists only one element that can be referred then the modeling effort without using PME is $M(m+2)$ and with using PME is $M(1)$; this is because if there is only one element then the modeler is not presented with a list of model elements, instead the reference gets created automatically. Similarly, when creating a connection between two elements, without using PME, the modelers must perform four actions. First, they need to check the source node for its properties; the modeling effort for this is $M(1)$. Next they must find the model element that is to be connected (*i.e.*, destination node); the modeling effort for this is $M(n)$. Thirdly, they have to select the source node for the connection; the modeling effort for this is $M(1)$. Finally, the destination node must be selected in order to establish the connection; the modeling effort for this is $M(1)$. The total modeling effort for creating a connection is, therefore, $M(n + 3)$. On the other hand, when the modeler is guided by PME they have to perform only two actions. First, select the source node, the modeling effort for which is $M(1)$. Second, select a destination node from the list of all valid elements which can be connected, the modeling effort for which is $M(1)$. The total modeling effort for creating a connection is now reduced to $M(2)$. Moreover, PME also relieves the modeler from the responsibility of invoking the constraint checker in GME whenever a model transformation is undertaken.

Table 5.2
Modeling effort for creating a model using the LMS DSML.

High-level goals	**Actions**	**Modeling effort**	
		without PME	**with PME**
Creating Library model	Adding 1 Library element + adding 3 Book elements + adding 2 HRStaff elements + adding 2 Librarian elements + adding 2 Shelf elements	$M(1) + 3M(1) + 2M(1) + 2M(1) + 2M(1) = M(10)$	$M(1)$
Creating and setting Patronref	Adding 1 Patronref element + finding and dragging the element to be referred	$M(1) + M(mn + 2) = M(mn + 3)$	$M(n + 2)$
Creating a Borrows connection	Selecting a Patron element and checking his/her major + selecting all the books (say n) in the library and checking their department + creating a connection between the two elements	$M(1) + M(n) + M(2) = M(n + 3)$	$M(2)$
Creating a Hires connection	Finding the librarians + creating a connection between Librarian and HRStaff elements	$M(n) + \mathrm{M}(2) = M(n + 2)$	$M(2)$
Creating an InterLibraryBorrow connection	Finding and checking the major of the referenced element + selecting all the books (say n) in the library and checking their department + creating a connection between the two elements	$M(n) + M(1) + M(n) + M(2) = M(2n + 3)$	$M(2)$

5.2 Measuring modeling effort in LMS

Table 5.2 compares the modeling efforts of a modeler in creating a Library model (refer Figure 3.15) with/without the use of PME. As shown in the table, proactive modeling significantly reduces the modeling effort. Moreover, PME ensures that the allowed transformations on the model adhere to the constraints of the DSML. This aspect of PME relieves the modeler from the duty of manually invoking the constraint checker in GME after completing any model transformation, such as adding elements, setting references, and creating connections to ensure a correct model. Therefore, the PME governs the entire book borrowing process of a patron.

5.3 Measuring modeling effort in PICML

The Platform Independent Component Modeling Language (PICML) is a DSML that enables developers to define component interfaces, QoS parameters and software building rules, and also generates descriptor files that facilitate system deployment [19]. It is a large scale DSML defined in GME and consists of approximately 930 modeling elements and 130 constraints [16]. The model abstractions and constraints in PICML are based on OMGs CORBA Component Model [20].

PICML has been used for many case studies in both academic and industry settings [16]. In academic settings, PICML has been used to support the research and development of deployment and configuration tools [21], system execution modeling tools [22], and optimization techniques [23] for component-based distribution system. Similarly, in industry settings, PICML has been used to support development of large-scale component based distributed systems such as shipboard computing environments, avionics systems, and global communications systems [16]. Since PICML is based on the OMGs CORBA Component Model and Deployment and Configuration specification, it has many high-level goals such as modeling of component interfaces, inter-component communication, network structures (*e.g.*, hosts) and mapping components to hosts [16].

This section, therefore, presents the modeling effort calculated during creation of a button component model using PICML. The initial modeling work, without using proactive modeling, was done by Dennis Feiock. The primary motive for this work was to extend UPPAAL TRON to support distributed system domains via the CORBA component model. The enormous number of functionalities possessed by PICML makes it very difficult for a novice modeler to complete a component. Primarily, the button component model can be segregated into 6 components as described below:

- **Interface Definitions.** It describes the interface for the button component. It defines ports and attributes associated with the component using the CORBA 3.x IDL feature provided by CCM. Ports define the incoming and outgoing events that the component consumes and produces. The button component interface has 2 OutEventPorts, `SingleClick` and `DoubleClick`, and 1 InEventPort, `Click`. Additionally, it also defines two Attributes, `ReporterTimeout` and `ReporterTimeunit`. The modeling effort for creating this component been discussed in Table 5.3.

- **Targets.** This section of the model provides the domains or nodes on which the button component will be deployed. In the button component example, there exists only 1 target node called `MainNode`. The modeling effort for creating this component has been discussed in Table 5.4.

- **Deployment Plans.** This part of the model is responsible for modeling component groups that will be deployed to a particular node. The modeling effort for creating this component has been discussed in Table 5.7.

- **Implementation Artifacts.** It defines the artifacts that are to be generated. In the button component example, there are 2 button artifacts to be generated called `ButtonImpl` and `ButtonSvnt`. The modeling effort for creating this component has been discussed in Table 5.5.

- **Component Implementations.** This part of the model consists of two segments: `ButtonImpl` and `ButtonAsm`. The ButtonImpl model element describes the implementation details of the button component and also links the required artifacts to the implementation. The ButtonAsm allows the modeler to provide actual values to the component attributes such as ReporterTimeout and ReporterTimeunit. The modeling effort for creating this component has been discussed in Table 5.6.

- **Predefined Datatypes.** This section of the model provides all the built in data types of PICML such as Boolean, Byte, and Char.

Table 5.3
Modeling effort for creating interface definition in button component.

High-level goal	**Actions**	**Modeling effort**	
		without PME	**with PME**
Creating the Interface Definition	Adding Folder `InterfaceDefinitions`	$M(1)$	$M(1)$
	Adding Model `File`	$M(1)$	$M(1)$
	Adding Event element `Notify`	$M(1)$	$M(1)$
	Adding Component element `Button`	$M(1)$	$M(1)$
	Creating InEventPort `Click` : adding an InEventPort element + setting its reference	$M(1) + M(m + 2)$	$M(1) + M(1)$
	Creating 2 OutEventPorts `SingleClick` and `DoubleClick`: adding OutEventPort elements + setting their references	$2M(1) + 2M(mn + 2)$	$2M(1) + 2M(n + 1)$
	Creating 2 Attributes `ReporterTimeout` and `ReporterTimeunit` : adding Attribute elements + setting their references	$2M(1) + 2M(mn + 2)$	$2M(1) + 2M(n + 1)$
	TOTAL	$M(4mn + m + 19)$	$M(4n + 14)$

Table 5.4
Modeling effort for creating targets in button component.

High-level goal	**Actions**	**Modeling effort**	
		without PME	**with PME**
Creating the Targets	Adding Folder `Targets`	$M(1)$	$M(1)$
	Adding Model `Domain`	$M(1)$	$M(1)$
	Adding Node element `MainNode`	$M(1)$	$M(1)$
	TOTAL	$M(3)$	$M(3)$

Table 5.5
Modeling effort for creating implementation artifacts in button component.

High-level goal	**Actions**	**Modeling effort**	
		without PME	**with PME**
Creating the Implementation Artifacts	Adding Folder `ImplementationArtifacts`	$M(1)$	$M(1)$
	Creating Artifact container `ButtonArtifacts`: adding ArtifactContainer model element + adding ImplementationArtifact element `ButtonImpl`	$M(1) + M(1)$	$M(1)$
	Adding ImplementationArtifact element `ButtonSvnt`	$M(1)$	$M(1)$
	TOTAL	$M(4)$	$M(3)$

Table 5.6
Modeling effort for creating component implementation in button component.

High-level goal	**Actions**	**Modeling effort**	
		without PME	**with PME**
Creating the Component Implementations	Adding Folder `ComponentImplementations`	$M(1)$	$M(1)$
	Creating ComponentImplementationContainer element `ButtonImpl`: adding ComponentImplementationContainer element `ButtonImpl` + adding ComponentRef model element+ setting its reference + adding 2 ComponentImplementationArtifacts + setting their references + adding MonolithicImplementation element + creating connections	$M(1) + M(1) + M(m + 2) + 2M(1) + 2M(mn + 2) + M(1) + 3M(n + 3)$	$M(1) + M(1) + 2M(n + 1) + 2M(1) + M(1) + 3M(n + 3)$
	Creating ComponentImplementationContainer element `ButtonAsm`: adding ComponentImplementationContainer element `ButtonAsm` + adding ComponentAssembly element + adding ComponentInstance element (adding ComponentInstanceType element + adding 2 OutEventPort elements + setting their references + adding 2 Attribute elements + setting their references) + adding 2 simple properties + setting their references + connecting the properties with the ComponentInstance element	$M(1) + M(1) + M(1) + (M(1) + 2M(1) + 2M(mn + 2) + 2M(1) + 2M(mn + 2)) + 2M(1) + 2M(mn + 2) + 2M(n + 3)$	$M(1) + M(1) + M(1) + (2M(1) + 2M(n + 1) + 2M(1) + 2M(n + 1)) + 2M(1) + 2M(n + 1) + 2M(n + 3)$
	TOTAL	$M(8mn + m + 5n + 49)$	$M(13n + 38)$

Table 5.7
Modeling effort for creating deployment plans in button component.

High-level goal	**Actions**	**Modeling effort**	
		without PME	**with PME**
Creating the Deployment Plans	Adding Folder `DeploymentPlans`	$M(1)$	$M(1)$
	Creating DeploymentPlan element `ButtonDeployment`: adding DeploymentPlan model element + adding NodeReference element + setting its reference + adding ComponentInstanceRef element + setting its reference + adding CollocationGroup element + connecting it to NodeReference element + adding 3 simple properties `TronParams`, `TronAdapterParams` and `StringIOR` + setting their references for the added properties + connecting these properties to CollocationGroup element and NodeReference element	$M(1) + M(1) + M(m + 2) + M(1) + M(m + 2) + M(1) + M(n + 3) + 3M(1) + 3M(mn + 2) + 3M(n + 3)$	$M(1) + M(1) + M(1) + M(1) + M(1) + M(n + 3) + 3M(1) + 3M(n + 1) + 2M(n + 3) + M(2)$
	TOTAL	$M(3mn + 4n + 2m + 30)$	$M(6n + 23)$

6 CONCLUDING REMARKS

Model-Driven Engineering (MDE) facilitates building solutions in many application domains through the systematic use of graphical languages called domain-specific modeling languages (DSMLs). Traditionally, it is the responsibility of the modelers to create domain-specific models using DSMLs. As domain-specific models increase in both size and complexity, it will be hard for modelers to cope. It is, however, necessary to go beyond existing model intelligence solution approaches to continue to improve the experience of modelers.

This thesis,therefore, presents a model intelligence approach called proactive modeling. We believe that it holds the potential to open new areas of research. Based on our experience implementing proactive modeling in GME, and applying it to several DSMLs, the following is a list of lessons learned and future research directions:

- **Assists novice modelers with learning a new DSML.** Proactive modeling guides a modeler throughout the modeling process by providing list of valid operations to choose from. Moreover, proactive modeling also enhances modeling experience through actions like auto-generation of elements, auto-reference resolving, auto-connection resolving, and automatic value entry for constrained attributes. These features of proactive modeling make it suitable for novice modelers because they are prevented from violating constraints, and helps them get through the tedious, labor-intensive, time-consuming process of manually creating a model.

- **Intelligent presentation of modeling actions needed.** Part of the proactive modeling process is presenting the modeler with a list of operations that are specific to the selected parent model. For simple DSMLs, this list is manageable. For large and complex DSMLs, the size of the list can easily be unmanageable if

presented in a haphazard way. It is therefore necessary to investigate techniques for presenting the list of valid operations so that the modeler does not become too overwhelmed, and can comfortably manage it.

- **Proactive modeling can fall victim to the "Clippy" syndrome.** Microsoft Office included an Office Assistant named "Clippy" that would try to assist the end-user based on their current actions. Unfortunately, "Clippy" was considered intrusive and annoying [24]. It is possible that proactive modeling can fall victim to this condition, which we call the "Clippy" syndrome. It is therefore critical that proactive modeling finds a way to be useful without being too intrusive. Otherwise, modelers will not want to use proactive modeling engines regardless of their benefits.

- **Discovering user-preferred modeling sequences.** There can easily be different sequences of modeling actions that achieve the same final model. Likewise, modelers may have their own order preference for executing modeling actions [10]. We believe that proactive modeling can provide the foundation for realizing this idea. Future research therefore will investigate how to support user-preferred modeling sequences in proactive modeling.

The PME discussed in this paper is freely available in open-source format, and is currently integrated in the CoSMIC tool suite. CoSMIC can be download from the following location: `www.dre.vanderbilt.edu/cosmic`.

LIST OF REFERENCES

LIST OF 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 $ATo M^3$. 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] Object Management Group. *Object Constraint Language*, 2006.

[12] David Abrahams and Aleksey Gurtovoy. *C++ template metaprogramming: Concepts, tools, and techniques from boost and beyond*. Addison-Wesley Professional, 2004.

[13] Joanna Chimia-Opoka, Michael Felderer, Chris Lenz, and Christian Lange. Querying UML models using OCL and Prolog: A performance study. In *Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on*, pages 81–88. IEEE, 2008.

[14] Jos Warmer and Anneke Kleppe. *The object constraint language: getting your models ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[15] David H. Akehurst, Gareth Howells, and Klaus D. McDonald-Maier. Supporting OCL as part of a Family of Languages. In *Proceedings of the MoDELS*, volume 5, pages 30–37, 2005.

[16] James H. Hill. Measuring and Reducing Modeling Effort in Domain-specific Modeling Languages with Examples. In *18th IEEE International Conference and Workshops on Engineering of Computer-Based Systems (ECBS)*, Las Vegas, NV, April 2011.

[17] Connie U. Smith and Lloyd G. Williams. *Performance Solutions: a practical guide to creating responsive, scalable software*, volume 1. Addison-Wesley Boston, MA;, 2002.

[18] Daniel A. Menasce, Virgilio A.F. Almeida, and Lawrence W. Dowdy. *Performance by design: computer capacity planning by example*. Prentice Hall, 2004.

[19] 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 (RTAS05)*, San Francisco, CA, March 2005.

[20] Object Management Group. *CORBA Compoenents v4.0*. Object Management Group, OMG Document formal/2006-04-01 edition, 2006.

[21] Gan Deng, Jaiganesh Balasubramanian, William Otte, Douglas C. Schmidt, and Aniruddha Gokhale. DAnCE: A QoS-enabled Component Deployment and Configuration Engine. In *Proceedings of the 3rd Working Conference on Component Deployment (CD 2005)*, pages 67–82, Grenoble, France, November 2005.

[22] John M. Slaby, Steve Baker, James H. Hill, and Douglas C. Schmidt. Applying system execution modeling tools to evaluate enterprise distributed real-time and embedded system QoS. In *Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings. 12th IEEE International Conference on*, pages 350–362. IEEE, 2006.

[23] Krishnakumar Balasubramanian. *Model-driven engineering of component-based distributed, real-time and embedded systems*. PhD thesis, Vanderbilt University, 2007.

[24] Wikipedia. Office Assistant. `http://en.wikipedia.org/wiki/Office_Assistant`.

[25] Vanderbilt University Institute for Software Integrated Systems. Gme 5 users manual (v5. 0). *Web Site: http://www. isis. vanderbilt. edu/Projects/gme/GMEUMan. pdf*, 2005.

[26] Vanderbilt University Institute for Software Integrated Systems. Gme tutorial. *Web Site: http://www. isis. vanderbilt. edu/Projects/gme/Tutorials*.

[27] N. Wirth. Extended backus-naur form (ebnf). *ISO/IEC*, 14977:2996, 1996.

APPENDICES

Appendix A: Generic Modeling Environment (GME)

The Generic Modeling Environment (GME) is a Windows-based, domain specific, model-integrated program synthesis tool for creating and evolving domain specific, multi-aspect models of large-scale engineering systems [25]. It was developed by the Institute of Software Integrated Systems (ISIS) at Vanderbilt University. The GME toolkit is configurable because it allows the representation of vastly different domains through metamodels which specify the modeling paradigm of the application domain. The modeling paradigm consists of all the syntactic, semantic, and presentation information about the domain including the concepts to be used for model construction, relationships between those concepts, guidelines to the modeler for viewing and organizing the concepts and the rules that govern the model construction [2]. The modeling paradigm behaves like a *blueprint* for generating a family of models.

A.1 GME Modeling concepts

GME encompasses a set of generic concepts that enable it to build large-scale, complex models. Figure A.1 presents an UML class diagram depicting the GME modeling concepts and the complex relationships between them.

The primary GME concepts, as shown in Figure A.1, are discussed as follows [2]:

- **Project:** A Project consists of a set of Folders.
- **Folder:** Folders act as containers for different sections of a modeling project. They are used for organizing the Models similar to how folders organize files on disk. It is compulsory that each modeling project should contain at least one root folder, located at the top of the hierarchy.
- **Atom:** Atoms are simple modeling objects that do not have an internal structure, *i.e.*, they cannot contain other entities. Atom is a type of First Class Object, or FCO, as it plays a vital role in any modeling project.
- **Model:** Models are compound, FCO entities that can contain other entities such as Atoms, other Models, References, Sets and Connections. These other

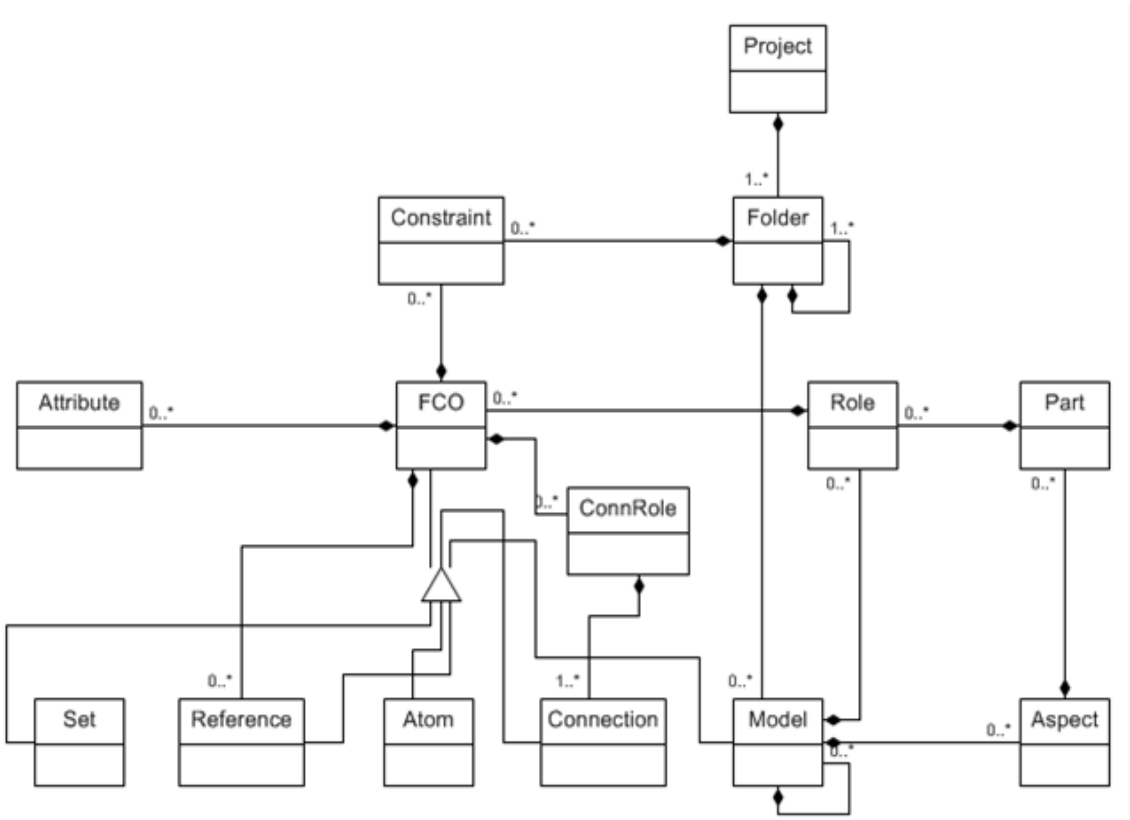

Figure A.1. GME Modeling Concepts.

entities residing in a Model are called Parts, each having its own role. Every entity or object in a paradigm must have a parent model, except for at least one Model object called *root model*.

- **Reference:** A Reference is a FCO entity with a built-in association for a single object, thereby acting as a pointer or an alias for that object.
- **Set:** A Set is a FCO entity that represents a collection of similar objects.
- **Connection:** A Connection is a FCO entity that represents a relationship between two objects contained by the same Model. Every Connection has at least two attributes: appearance and directionality.
- **Attribute:** An Attribute is an entity that represents an attribute or property of a particular FCO textually.

- **Aspect:** Aspects represent different "views" of the structure of a model. Aspect is a mechanism to maintain readability by segmenting or filtering the models.
- **Constraint:** Constraints are validity rules, expressed in a predicate language called Object Constraint Language (OCL), which can be applied to a model.

A.2 GME Interfaces

The modeling process in GME requires a modeler to primarily interact with five major interfaces as shown below:

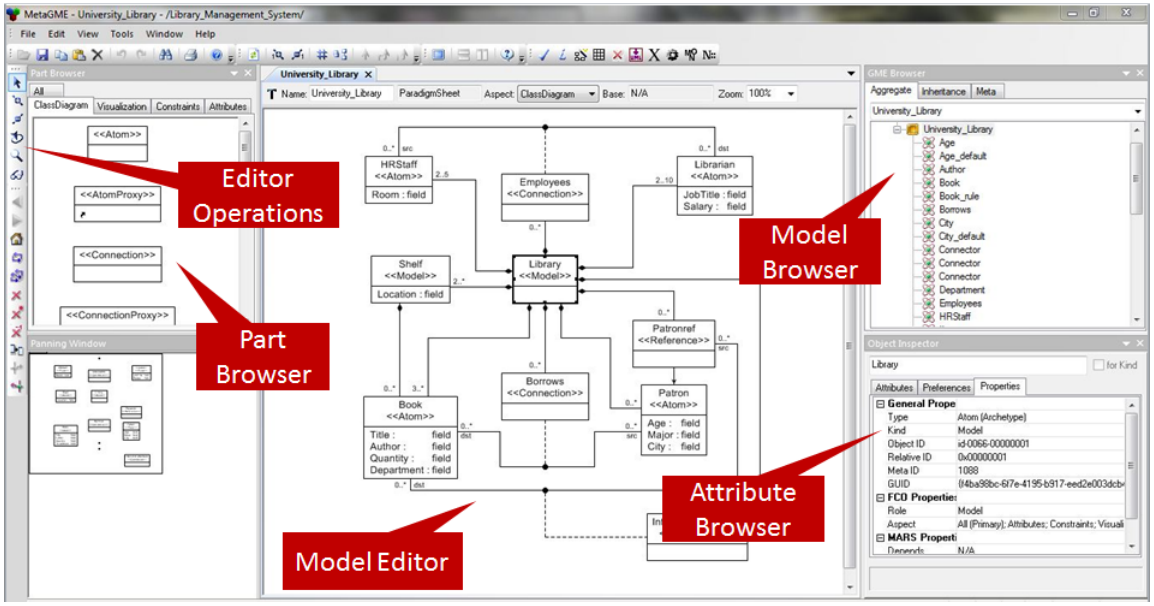

Figure A.2. GME Interfaces.

- **Part Browser:** The Part Browser window, as shown in Figure A.2, displays the parts that can be inserted into the current model in the current aspect. It includes all the modeling concepts that are used to create a metamodel or a model.
- **Model Browser:** The Model Browser is primarily used to organize the individual models that make up an overall project. As shown in Figure A.2, this browser

consists of three tabs: Aggregate tab (which contains a tree based containment hierarchy of all Folders, Models and other parts), Inheritance tab (which is used for visualizing the type inheritance hierarchy), and Meta tab (which shows the modeling paradigm specifications).

- **Attribute Browser:** The Attribute Browser allows a user to view/modify the attributes, preferences and properties of an object in the metamodel or model. As shown in Figure A.2, this browser consists of three tabs: Attributes tab, Preferences tab, and Properties tab.
- **Model Editor:** The Model Editor consists of two segments: Editor Window (which shows the contents of the selected model in one aspect at a time), and GME Menus (which displays a menu-bar for various commands such as `New Project`, `Open Project`).
- **Editor Operations:** As shown in Figure A.2, the graphical editor provides six editing modes to the user: Normal mode (which is used to add/delete /move/-copy parts within the editing windows), Add Connection mode (which allows connections to be made between the modeling objects), Delete Connection mode (which allows connections to be removed between the objects), Set mode (which allows the user to add objects into the set of a model), Zoom mode (which allows the user to view the models at different levels of magnification), and Visualization mode (which allows single objects and collections of objects to be visually highlighted with respect to other modeling objects).

A.3 The Modeling Process

The modeling process in GME is a three-tier process as shown in Figure A.3. The first step of the modeling process requires configuring GME to create a metamodel that captures all the features of an application domain. In other words, the metamodeling process can be described as modeling the modeling process [26]. Metamodeling is a repetitive process that results in a compiled set of rules, and the paradigm or the

modeling language that configures GME for a specific application domain. The domain specific metamodels are created in GME using the MetaGME paradigm, which acts like a metamodeling language or a meta-metamodel (represented using UML class diagram notations). MetaGME is a reusable framework for creating domain-specific design environments. It supports an enriched set of abstract, generic modeling concepts such as containment, module interconnection, multi-aspect modeling, inheritance, and attributes which make it suitable for a wide range of domains. The metamodels, however, only specify the syntax of the domain modeling language; the static semantics (*i.e.*, set of rules that specify the well-formedness of domain models) of the language is specified using a textual predicate logic language called Object Constraint Language (OCL). After the metamodel has been created, the user is required to register the created metamodel using the MetaGME interpreter. The registering process converts the metamodel into a GME modeling paradigm, which can be then used for creating models. The second step of the modeling process requires the mod-

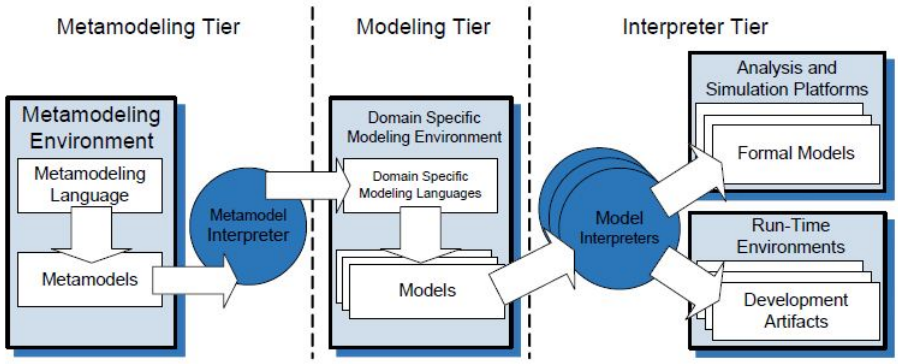

Figure A.3. The modeling process in GME.

eler to design models using the domain-specific modeling paradigm. As shown in the figure, the modeler can create several separate models capturing different aspects of the domain from a single registered paradigm or DSML.

The third and the final step of the modeling process involves the use of model-interpreters for a variety of purposes discussed in the next section.

A.4 Model Interpreters

Model transformation is an important criterion for model-driven development paradigms. It allows a single model to be used for a variety of purposes such as:

- Creating analysis tools and simulation engines.
- Running queries, generating lists, and writing reports based on the contents of the model.
- Generating program code or system configuration.
- Using the models as a data exchange format to integrate tools that are incompatible with each other.

Model transformation is achieved using Model Interpreters; these model-interpreters perform a semantic transformation which requires understanding the detailed semantics of the model. This, therefore, makes it necessary for the modeler to keep the metamodel and model-interpreter in sync, especially when the model-interpreter has hard-coded knowledge about the entities and their relationships in the DSML. The interpreter is invoked by the modeler tool either upon user request or by internally generated events such as adding a new object. Upon invocation, the interpreter traverses the entire model hierarchy or processes a part of it in the event context only to perform a wide variety of actions such as generating output, performing model transformation, and vetoing the model modification (only for event triggered interpreters). GME supports three types of model-interpreters:

- **Interpreter:** Interpreters are DSML specific components that are invoked explicitly by the user. These are the most commonly used form of model-interpreters. For example, GME has an in-built interpreter called `MetaGME Interpreter`, which converts a metamodel to a paradigm and registers that paradigm with GME.
- **Plug-in:** Plug-ins are domain independent components that are invoked explicitly by the user. Examples of plug-ins in GME are `Search plug-in`, and `XML export/import`.

- **Add-on:** Add-ons are DSML specific or domain independent, event-driven model interpreters. They are invoked by events such as *Object Added*, *Object Deleted*, and *Attribute Changed*. For example, GME consists of the `Constraint Manager`, which can be considered an interpreter as well as an add-on at the same time. This is because it can be explicitly invoked by the user and also by event-driven constraints present in the given paradigm. Any operation that causes a violation to the constraint is aborted [2].

The model-interpreters interact with the models using high-level interfaces discussed as follows:

- **Component Object Model (COM):** The architecture of GME is based on Component Object Model; therefore, COM is the primary interface used to access GME data. However, programming with COM is difficult because of its error-prone low-level details and also because the user must handle the full GME interface protocol including transactions, territories, and event handling.
- **Builder Object Network 2 (BON2):** This is a framework that provides a network of C++ objects (or builder objects), each representing a GME object. The builder objects shield the developer from the low-level COM interface, thereby making it easier to interpret and traverse models.
- **Universal Data Model (UDM):** UDM uses UML diagrams to describe data structures and to automatically generate C++ and Java classes that represent the data structures.
- **GMEs Automation Modeling Engine (GAME):** GAME is a modeling framework being developed by Dr.James Hill. Its main goal is to address shortcomings of existing backend frameworks such as BON2 and UDM which currently exist for GME, such as incorrectly implemented Visitors and incomplete extension classes. It also simplifies writing different components for GME, such as decorators, add-ons, and interpreters.

Appendix B: Add-on skeleton generator for GME

The first step of developing PME involved generating an empty add-on for GME. For this purpose, we created an add-on skeleton generator script for GME in python. This generator not only auto-generates all the necessary files for an add-on but it also registers it with specific or all paradigms existing in GME. After the auto-generation, the user can then simply create event handlers to handle certain events. The command for executing the python script is: `generate_gme_addon.py [OPTIONS]`. The options that a user can provide are shown as follows:

- **-o**: This option is used for providing the location for the newly generated add-on. If the user does not provide the location, then the add-on is generated in the same folder as the script.
- –**name**: This option is used for providing the name of the newly generated add-on. If the user does not provide a name then the add-on gets the name `Default`.
- –**component-guid**: This option is used for providing the component-unique identifier. If the user does not provide this information then the component GUID gets auto-generated.
- –**library-guid**: This option is used for providing the library-unique identifier. If the user does not provide this information then the library GUID gets auto-generated.
- –**paradigm**: This option is used for providing the paradigm with which this add-on gets registered. If the user does not provide the paradigm then the add-on gets registered with all paradigms.

Appendix C: OCL Parser and Evaluator

This chapter of the thesis presents a detailed discussion of all the key components of the OCL parser and evaluator. Figure C.1 shows an overview of the architecture of OCL parser and evaluator. As shown in the figure, the OCL parser and evaluator consists of six key components: OCL Parser, Boolean Expressions Hierarchy, Value Expressions Hierarchy, Method Hierarchy, Iterator Hierarchy, and Value Hierarchy. Each of these components have been described in more detail later in the chapter.

C.1 OCL Parser

The OCL parser is responsible for parsing OCL constraints and dynamically creating an abstract syntax tree from the parsed OCL constraints. The nodes of the abstract syntax tree (AST) are actually objects of various classes defined in the boolean-expressions hierarchy. The OCL parser in the PME is designed and implemented using the *Boost Spirit Parser Framework*, discussed later in Appendix D. The evaluation of the generated AST is triggered by the handlers on its root node. The class objects that form the AST evaluate a certain portion of the constraint. We have segregated the evaluation process into two types based on the purpose of evaluation, as described below:

- **Evaluate.** This evaluation process is invoked to check if an OCL constraint has been satisfied or not. This type of evaluation is used by PME, when dealing with containment or attribute constraints. PME evaluates these constraints to check if they have been satisfied or not. If they have not been satisfied then it auto-generates the required model elements or auto-fills the attribute values.
- **Filter Evaluate.** This evaluation process is invoked to evaluate the OCL constraint to provide modeler guidance. This type of evaluation is used by PME, when dealing with association or reference constraints. In this case, PME computes a set of target objects ahead of time, which are then used for validating the constraints. The objects which validate the constraint are then presented

to the modeler for selection. For example, for the reference constraint shown in Figure 3.12, when a reference object Patronref is added to the model, PME queries the metamodel for the object type being referred to by Patronref *i.e.*, Patron. PME then collects all the objects of type Patron present in the model and evaluates the constraint against each of the collected objects. The objects which satisfy the constraint are then presented to the modeler for selection.

One point to note here is, the fact that every class in each hierarchy holds the capability to both evaluate or filter evaluate the sub-expression that it represents.

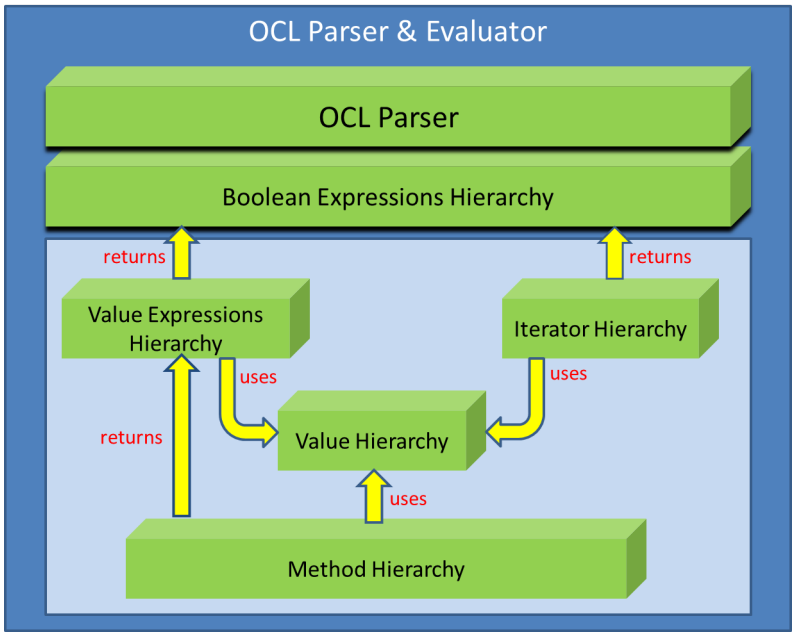

Figure C.1. Architectural overview of OCL parser and evaluator.

C.2 Boolean Expressions Hierarchy

The boolean-expressions hierarchy is composed of classes that are responsible for evaluating the basic expression types in OCL (*e.g., Let expressions*). The base class for this hierarchy is `BoolExpr`. All the classes that exist in this hierarchy return a

boolean value, which denotes the evaluation success of a particular expression. The major classes that are defined in this hierarchy are shown below [25]:

- **LocalAssignmentValueExpr.** This class is responsible for evaluating the OCL *Let Expressions*. The syntax of a let expression is:
  $let < variableName > \{ : < declarationType >\} = < sub-expression > in.$
  This expression has two parts; the first part declares and initializes a variable, while the second part declares the accessibility of the variable (although the second part is optional). The type of value stored by the variable is determined by the return type of the right hand side sub-expression, which belongs to the value-expressions hierarchy. A sample constraint showing the use of let expressions has been shown in Figure C.2.

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

Figure C.2. Sample constraint showing the use of let expressions.

As shown in the constraint, the right hand side expression evaluates the number of Patrons contained by the invoking object, which then gets stored in the variable partCount.

```
if mySet -> isEmpty() then 0 else mySet -> size endif
```

Figure C.3. Sample constraint showing the use of if-then-else expressions.

- **IfThenElseExpr.** This class is responsible for evaluating the OCL *If Then Else Expressions*. The syntax of an if-then-else expression is:
  $if < condition > then < sub-expression > else < sub-expression > endif.$
  This expression is comprised of three parts; the first part is a condition that returns a Boolean value, while the second and third parts are sub-expressions

which have the same return type. If the condition returns true, then the first sub-expression is evaluated, otherwise the second sub-expression is evaluated. All the three parts of an if-then-else expression belong to the boolean-expressions hierarchy. A sample constraint showing the use of if-then-else expressions is shown in Figure C.3. As shown in the constraint, if a mySet is empty then the first sub-expression returns 0, otherwise the second sub-expression returns the size of the set.

- **IteratorExpr.** This class is responsible for evaluating OCL expressions containing *Iterators*. The syntax of such an expression is:
  $< expression > - > \ < iteratorName > (\{< declarator > (, \ < declarator >) * \{ : \ < declarationType > \}\} | < sub - expression >)$.
  This expression has three mandatory parts and two optional parts. The mandatory parts include the object (collection or set of objects) invoking the iterator, the name of the iterator (belongs to the iterator hierarchy) and the sub-expression which has to be evaluated for each element of the collection. The sub-expression belongs to the boolean-expressions hierarchy. The optional parts include the declarators, which are variables that refer to the current element of the iteration process, and declaration type which defines the type of the declarators. A sample constraint showing the use of iterators is shown in Figure C.4. As shown in the constraint, variable mySet stores a collection of objects whose

```
let mySet = self.parts (“Person”) in
mySet -> forAll( e1, e2 : int | e1.SSN <> e2.SSN )
```

Figure C.4. Sample constraint showing the use of iterators.

meta-type is Person. An Iterator *forAll* is applied to the collection of objects to check that no two persons have the same SSN.

- **EqualityExpr.** This class deals with evaluating equality expressions such as comparison and conjunction type of expressions. The syntax of an equality expression is:

  $< sub - expression >< equalityoperator >< sub - expression >.$

  This expression has three parts: a left hand side sub-expression, an equality operator and a right hand side sub-expression. Both the sub-expressions have the same return types. The values returned by these sub-expressions are evaluated based on the equality operators such as and, =.

- **ComparisonExpr.** This class is derived from the EqualityExpr class and it is responsible for evaluating comparison expressions. The syntax of this type of expression is similar to that of EqualityExpr. However, for comparison expressions both the sub-expressions belong to the value-expression hierarchy. Also, the equality operators that are used for comparison expressions are `=`, `>=`, `<=`, `>`, `<`, and `<>`. An example of such an expression is shown in Figure C.5. As

```
self.parts ("Book")->size >= 2
```

Figure C.5. Sample constraint showing the use of comparison expressions.

  shown in the constraint, the left hand side sub-expression calculates the number of Book type objects that exist in the model, which is compared with the constant expression on the right hand side. The constraint evaluates to be true if the number of Book objects is at least 2.

- **EqualExpr.** This class is derived from ComparisonExpr class and it is responsible for evaluating comparison expressions that have `=` as their equality operator.

- **GreaterEqualExpr.** This class is also derived from ComparisonExpr class and it is responsible for evaluating comparison expressions that have `>=` as their equality operator.

- **LesserEqualExpr.** This class is derived from ComparisonExpr class and it is responsible for evaluating comparison expressions that have `<=` as their equality operator.
- **NotEqualExpr.** This class is derived from ComparisonExpr class and it is responsible for evaluating comparison expressions that have `<>` as their equality operator.
- **GreaterExpr.** This class is derived from ComparisonExpr class and it is responsible for evaluating comparison expressions that have `>` as their equality operator.
- **LesserExpr.** This class is derived from ComparisonExpr class and it is responsible for evaluating comparison expressions that have `<` as their equality operator.
- **ConjunctionExpr.** This class is derived from EqualityExpr class and it is responsible for evaluating conjunction expressions. The syntax of this type of expression is also similar to that of EqualityExpr. However, for conjunction expressions both the sub-expressions belong to the boolean-expression hierarchy. Also, the equality operators that are used for comparison expressions are `and`, `or`. An example of such an expression is shown in Figure C.6. As shown in

```
(self.parts (“Book”)->size >= 2) and
(self.parts ("Book")->size <= 5)
```

Figure C.6. Sample constraint showing the use of conjunction expressions.

the constraint, the left hand side sub-expression determines that there should be at least 2 books present in the model; the right hand side sub-expression determines that there can be at most 5 books in the model.

- **AndExpr.** This class is derived from ConjunctionExpr and it is responsible for evaluating conjunction expressions that have `and` as their equality operator.

- **OrExpr.** This class is also derived from ConjunctionExpr and it is responsible for evaluating conjunction expressions that have `or` as their equality operator.

C.3 Value Expressions Hierarchy

The value-expressions hierarchy is responsible for evaluating the expressions that return a `Value`. `Value` is any result that is generated from computing a value-expression. There can be several types of `Value` such as BooleanValue, StringValue, ObjectValue (discussed in Appendix C.6) that can be returned by an expression. The base class for this hierarchy is `ValueExpr`. The major classes that are derived from ValueExpr are shown as below [25]:

- **MethodCall.** This class is derived from ValueExpr and it is responsible for evaluating method-calls that exist in an OCL constraint. It has the capability to evaluate single or a chain of methods, where the output value of one method acts as the input for the next method. The methods in a method-call belong to the method hierarchy. This class forms the base class for two specific types of method-calls: SelfMethodCall and LocalValueMethodCall. The syntax for a method-call is:
  $< invoking\_object >$ "." $< method >$ $*\{$"." $< method >\}$.
  As shown in the syntax, the first method is invoked by the invoking object. The result from this method is then used for invoking the next method. The final value returned by this class during evaluation depends on the return type of the last invoked method.

- **SelfMethodCall.** This class is derived from MethodCall and it is responsible for evaluating method-calls that have `self` as their invoking object. The `self` keyword indicates that the invoking object is the model element to which the constraint is associated. An example of SelfMethodCall is shown in the Figure C.7. As shown in the constraint, the `self` keyword is associated to the parent model of Patron. The first method, *i.e.*, `parts` returns back a collection

```
self.parts("Patron")->size = 2
```

Figure C.7. Sample constraint showing the use of SelfMethodCall.

of Patron type model elements that currently exists in the parent model. The second method, *i.e.*, `size` takes the collection and returns back its size, which is then used by EqualExpr for comparison with the right hand side.

- **LocalValueMethodCall.** This class is also derived from MethodCall and it is responsible for evaluating method-calls that have local variables as their invoking object. An example of LocalValueMethodCall is shown in Figure C.8. As shown in the constraint, the `attachingConnections` method is invoked by

```
let p:Patron = self.refersTo() in
p.attachingConnections()->size <= 5
```

Figure C.8. Sample constraint showing the use of LocalValueMethodCall.

the local variable that contains a Patron type object. This method returns back a collection of all the connections in which the object `p` has participated. The method `size` is then called on the returned collection to calculate its size.

- **ConstantValueExpr.** This class is derived from ValueExpr and is responsible for dealing with constant values in the expression. The types of constant values supported by this class are integer and string. During evaluation the value of the constant expression is returned. Figure 3.7 provides an example showing the use of constant expressions. When the evaluation is invoked on ConstantValueExpr, it returns the constant value, *i.e.*, string.

- **LocalValueExpr.** This class is derived from ValueExpr and is responsible for dealing with the values of local variables in the expression. During evaluation the value of the local variables is returned. Figure C.8 shows the use of a local variable `p`.
- **AttributeExpr.** This class is also derived from ValueExpr and is responsible for dealing with the Attributes of different model elements. On evaluation, this class returns the value of the attribute related to the invoking object. Figure 3.7 provides an example of an expression using attributes. As shown in the figure, the value of City attribute for a Patron is compared with a constant expression. During evaluation, the AttributeExpr will return the value of City.

C.4 Method Hierarchy

The method hierarchy is composed of classes that are responsible for evaluating the GME methods supported by OCL [25]. The base class for this hierarchy is `Method`. All the classes that exist in this hierarchy return a value, which belongs to the value hierarchy. The major classes that are defined in this hierarchy are shown below [25]:

- **Name.** This class represents the `name` method, which is a method specific to `gme::Object`. The syntax of this method is:
  name().
  This method, when evaluated, returns the name of the object.
- **KindName.** This class represents a `gme::Object` method called `kindName`. The syntax of this method is:
  kindName().
  This method, when evaluated, returns the name of the kind of the object.
- **Parent.** The Parent class represents the `gme::Object` specific `parent` method. The syntax of this method is:
  parent().

This method, when evaluated, returns the parent of the object. The metakind of this returned parent can be either `gme::Folder` or `gme::Model`. However, if the invoking object is the rootfolder of the project, then this methods returns `null`.

- **IsFCO.** This class represents the `gme::Object` specific method called `isFCO`. The syntax of this method is:
  $isFCO()$.
  On evaluation, this method returns a boolean value, which is true if the metakind of the invoking object is `gme::FCO` or its descendants.

- **ChildFolders.** This class represents the `childFolders` method, which is a method specific to `gme::Folder`. The syntax of this method is:
  $childFolders()$.
  This method, when evaluated, returns a collection of all the folders contained by the invoking folder object.

- **Models.** This class represents the `gme::Folder` specific method called `models`. The syntax of this method is:
  $models(\{kind\})$.
  On evaluation, this method returns a collection of all the models that are contained by either the invoking folder or any child folder or model contained by the invoking folder. If the user specifies a `kind`, then the returned collection will contain objects that are of type `kind`. However, the metakind (*i.e.* kind of `kind`) of `kind` should be `gme::Model`, otherwise an exception is thrown.

- **Atoms.** This class represents the `gme::Folder` specific method called `atoms`. The syntax of this method is:
  $atoms(\{kind\})$.
  On evaluation, this method behaves similar to the `models` method mentioned above. However, the returned collection consists of atoms. If the user specifies a `kind`, then the returned collection will contain objects that are of type `kind`.

- **ConnectedFCOs.** This class represents the `gme::FCO` specific method called `connectedFCOs`. The syntax of this method is:
  $connectedFCOs(\{role\}\{, kind\})$ or, $connectedFCOs(kind)$.
  This method, when evaluated, returns a collection that contains all the FCOs (refer Figure A.1) that are associated with the invoking FCO. If the user specifies a `role` (*i.e.* source or destination), then only those FCOs are returned that have the same role in the connection. Also, if the `kind` is specified then the kind of connections that are checked for connected FCOs is `kind`. However, the metakind of `kind` should be `gme::Connection`, otherwise an exception is thrown.

- **AttachingConnPoints.** This class represents the `gme::FCO` specific method called `attachingConnPoints`. The syntax of this method is:
  $attachingConnPoints(\{role\}\{, kind\})$ or, $attachingConnPoints(kind)$
  On evaluation, this method returns a collection of all the connection points *i.e.*, association ends of the FCO. If the user specifies a `role`, then the role of the connection point should match that. Also, if the `kind` is specified then the kind of connections that are checked is `kind`. However, the metakind of `kind` should be `gme::Connection`, otherwise an exception is thrown.

- **AttachingConnections.** This class represents the `gme::FCO` specific method called `attachingConnections`. The syntax of this method is:
  $attachingConnections(\{role\}\{, kind\})$ or, $connectedFCOs(kind)$.
  This method, when evaluated, returns a collection of all the connections that are linked to the invoking FCO. If the user specifies a `role`, then the connection point role in the invoking FCO's side should match that. Also, if the `kind` is specified then the kind of connections that are considered should match the `kind`. However, the metakind of `kind` should be `gme::Connection`, otherwise an exception is thrown.

- **IsConnectedTo.** This class also represents a `gme::FCO` specific method called `isConnectedTo`. The syntax of this method is:
  $isConnectedTo(fco\{, role\{, kind\}\})$ or $isConnectedTo(fco, kind)$.
  On evaluation, this method returns true if the invoking FCO is connected to `fco`. If a `role` is specified, then the role of the `fco` has to match the `role`. Also, if `kind` is specified then the kind of the regarded connections must be `kind`. However, the metakind of `kind` should be `gme::Connection`, otherwise an exception is thrown.

- **Subtypes.** This class represents the `subTypes` method, which is a method specific to `gme::FCO`. The syntax of this method is:
  $subTypes()$.
  This method, when evaluated, returns the collection of all the FCOs that are subtypes of the invoking FCO. However, if the invoking FCO is not a type, then an empty collection is returned.

- **Instances.** This class represents the `gme::FCO` specific method called `instances`. The syntax of this method is:
  $instances()$.
  On evaluation, this method returns a collection containing all the type-instances of the invoking FCO. However, this method returns an empty collection if the invoking FCO is itself an instance.

- **Type.** This class represents the `gme::FCO` specific method called `type`. The syntax of this method is:
  $type()$.
  This method, when evaluated, returns the type of the invoking FCO.

- **Basetype.** This class represents the `gme::FCO` specific method called `baseType`. The syntax of this method is:
  $baseType()$.
  This method, when evaluated, returns the base type of the invoking FCO.

- **IsType.** This class represents the `gme::FCO` specific method called `isType`. The syntax of this method is:
  $isType()$.
  This method, when evaluated, returns true if the invoking FCO is a type.

- **IsInstance.** This class represents the `isInstance`, which is specific to`gme::FCO`. The syntax of this method is:
  $isInstance()$.
  This Method, when evaluated, returns true if the invoking FCO is an instance.

- **Folder.** This class represents the `gme::FCO` specific method called `folder`. The syntax of this method is:
  $folder()$.
  On evaluation, this method returns the closest folder that contains the invoking FCO recursively over models.

- **ReferencedBy.** This class represents the `gme::FCO` specific method called `referencedBy`. The syntax of this method is:
  $referencedBy(\{kind\})$.
  This method, when evaluated, returns a collection of references that refer to the invoking FCO. If `kind` is specified, then only the references of type `kind` will be collected. However, the metakind of `kind` should be `gme::Reference`, otherwise an exception is thrown.

- **ConnectionPoints.** This class represents the `gme::Connection` specific method called `connectionPoints`. The syntax of this method is:
  $connectionPoints(\{role\})$.
  On evaluation, this method returns the collection of connection points (*i.e.* association ends) of the connection. If `role` is specified then role of the points has to match that.

- **ConnectionPoint.** This class represents the `gme::Connection` specific method called `connectionPoint`. The syntax of this method is:

connectionPoint(*role*).

On evaluation, this method returns the connection point whose role matches the provided `role`.

- **UsedByConnPoints.** This class represents the `usedByConnPoints` method, which is a method specific to `gme::Reference`. The syntax of this method is:
  usedByConnPoints({*kind*}).
  On evaluation, this method returns a set of all the connection points of the reference in which the reference participates. If the `kind` is specified then only those points are collected whose connection kind is `kind`. However, the metakind of `kind` should be `gme::Reference`, otherwise an exception is thrown.

- **Refersto.** This class represents the `gme::Reference` specific method called `refersTo`. The syntax of this method is:
  refersTo().
  This method, when evaluated, returns the FCO to which the invoking reference refers.

- **Parts.** This class represents the `parts` method, which is a method specific to `gme::Model`. The syntax of this method is:
  parts({*role*}) or, *parts*(*kind*).
  On evaluation, this method returns a collection that contains all the immediate children of the invoking model. The user can specify a role name `role`, which is the containment role of the object as it is contained by the model. The user can also specify `kind`, which governs the metakind of model elements that can be included in the collection.

- **AtomParts.** This class represents the `atomParts` method, which is a method specific to `gme::Model`. The syntax of this method is:
  atomParts({*role*}) or, *atomParts*(*kind*).
  This method behaves similar to the `parts` method during evaluation. However, this method collects only the atom type children of the invoking model.

- **ModelParts.** This class represents the `modelParts` method, which is a method specific to `gme::Model`. The syntax of this method is:
  $modelParts(\{role\})$ or, $modelParts(kind)$.
  This method behaves similar to the `parts` method during evaluation. However, this method collects only the model type children of the invoking model.

- **ReferenceParts.** This class represents the `referenceParts` method, which is a method specific to `gme::Model`. The syntax of this method is:
  $referenceParts(\{role\})$ or, $referenceParts(kind)$.
  This method behaves similar to the `parts` method during evaluation. However, this method collects only the reference type children of the invoking model.

- **ConnectionParts.** This class represents the `connectionParts` method, which is a method specific to `gme::Model`. The syntax of this method is:
  $connectionParts(\{role\})$ or, $connectionParts(kind)$.
  This method behaves similar to the `parts` method during evaluation. However, this method collects only the connection type children of the invoking model.

- **Size.** This class represents the `size` method. The syntax of this method is:
  $size$.
  This method, on evaluation, returns the size of the collection that invoked it.

- **Target.** This class represents the `target` method, which is a method specific to `gme::ConnectionPoint`. The syntax of this method is:
  $target()$.
  On evaluation, this method returns the FCO to which the invoking connection point is attached.

- **Owner.** This class represents the `owner` method, which is a method specific to `gme::ConnectionPoint`. The syntax of this method is:
  $owner()$.
  On evaluation, this method returns the connection which owns the invoking connection point.

C.5 Iterator Hierarchy

The iterator hierarchy is composed of classes that are responsible for evaluating the iterators contained in the iterator expressions. These iterators work on the collection that invokes them. The base class for this hierarchy is `Iterator`. All the classes that exist in this hierarchy return a value, which belongs to the value hierarchy. The major classes that are defined in this hierarchy are shown below [25]:

- **ForAll.** This class represents the `forAll` iterator. On evaluation, this iterator returns a boolean value true, if the sub-expression (contained in the iterator expression) evaluates to true for all the elements contained in the collection. However, this iterator also returns true if the invoking collection is empty.
- **IsUnique.** This class represents the `isUnique` iterator. On evaluation, this iterator returns a boolean value true, if the sub-expression (contained in the iterator expression) evaluates to different values for all the elements contained in the collection.
- **Exists.** This class represents the `exists` iterator. On evaluation, this iterator returns a boolean value true, if the sub-expression (contained in the iterator expression) evaluates to true for at least one element contained in the collection. However, this iterator returns false if the invoking collection is empty.
- **One.** This class represents the `one` iterator. On evaluation, this iterator returns a boolean value true, if the sub-expression (contained in the iterator expression) evaluates to true for only one element contained in the collection.

C.6 Value Hierarchy

The value hierarchy is composed of classes that represent the different data-types that can be evaluated or returned by the classes belonging to other hierarchies. The base class for this hierarchy is `Value`. The classes contained in this hierarchy not only represent a particular data-type but they also provide operations such as add,

subtract, equal, that can be performed on these data-types. The major classes that are defined in this hierarchy are shown as below:

- **IntValue.** This class represents the `int` data-type.
- **StringValue.** This class represents the `string` data-type.
- **LongValue.** This class represents the `long` data-type.
- **BooleanValue.** This class represents the `bool` data-type.
- **DoubleValue.** This class represents the `double` data-type.
- **ObjectValue.** This class represents the `gme:Object` data-type.
- **CollectionValue_T.** This template class represents the different types of collections that can be used during evaluation of expressions.

Appendix D: Boost Spirit Parser Framework

The Boost Spirit Parser Framework is an object oriented, back-tracking, recursive descent parser generator framework implemented using template meta-programming techniques [12]. The expression templates of this framework allow users to write grammars and format descriptions using a syntax similar to Extended Backus Naur Form (EBNF) [27] directly in C++. This is an added advantage over other parser generators that require an additional translation step from source EBNF code to C or C++ code. Spirit also allows seamless integration of the parsing and output generation process with other C++ code. The Boost Spirit library consists of 4 major parts shown as below (`boost-spirit.com`):

- **Spirit Classic.** This segment consists of the former Boost Spirit distribution and includes a special compatibility layer to ensure compatibility with the current distribution.
- **Spirit Qi.** This is the parser library that allows users to build recursive descent parsers. The domain-specific embedded language (DSEL) exposed by Spirit Qi allows the user to describe the grammars, and the rules for storing the parsed information.
- **Spirit Lex.** This is the library which is used for creating tokenizers or lexers. The DSEL exposed by Spirit Lex allows users to define regular expressions for token matching, associate code with the regular expressions that are to be executed for a successful match, and add the token definitions to the lexical analyzer.
- **Spirit Karma.** This is a generator library that allows users to create code for recursive descent, data type-driven output formatting.

The OCL parser component in Proactive Modeling Engine is build using the Spirit Qi parser library. For example, Figure D.1 shows a set of production rules that represent a calculator.

```
group = '(' >> expression >> ')';
factor = integer | group;
term = factor >> *(('*' >> factor) | ('/' >> factor));
expression = term >> *(('+' >> term) | ('-' >> term));
```

Figure D.1. Grammar specification for a calculator.

The production rule `expression`, shown in the listing can recognize inputs such as: `12345`, `1 + 2 * 4 / 6`, `1 + ((6 * 200) - 20) / 6` etc.