The Model Manager: A Multidimensional Conceptual Modeling Tool in the CIM Framework

by

Dibesh Shakya

B. Eng. in Computer Engineering, Tribhuvan University, 2006

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES
(Computer Science)

The University Of British Columbia
(Vancouver)

February 2011

© Dibesh Shakya, 2011
Abstract

Organizations of all sizes collect a vast quantity of data every year. A data warehouse facilitates strategic multidimensional analysis on these data, by providing a single integrated queryable interface for multiple heterogeneous data sources. However, this interface is at the logical level schema, which fails to take advantage of the fundamental concepts of multidimensional data analysis like facts, dimensions, hierarchies, etc. In this thesis, we discuss a conceptual modeling language from the Conceptual Integration Modeling framework that serves multidimensional data to the users at a much higher level of abstraction. We not only provide the formal semantics for the language, but we also supply conceptual models depicting real world data analysis scenarios. These models are backed up with rigorous mathematical definitions to dispel any ambiguity in their interpretation. We developed a fully functional graphical editor, called the Model manager, to enable users to draw conceptual models visually.
Preface

This thesis is a result of a collaborative research between five people: Dibesh Shakya, Dr. Rachel Pottinger, Dr. Iluju Kiringa, Dr. Flavio Rizzolo and Kwok Wong. This work is a step forward in fulfilling the overall objective of the Conceptual Integration Modeling (CIM) framework, a joint work under supervision of Dr. Pottinger from the University of British Columbia and Dr. Kiringa from the University of Ottawa. The big picture of the CIM framework can be found in the paper [24]. This thesis focuses on only one component, the Model Manager, in the architecture of the CIM framework. I used plural nouns when referring to authors throughout the thesis to emphasize team work. Details of contribution made by each co-author is as follows:

Dr. Rachel Pottinger was my supervisor for my master’s program. She assisted me in developing novel ideas through weekly meetings. The design of the research was done by Dr. Kiringa, Dr. Pottinger and Dr. Rizzolo. They helped me both in finding and choosing relevant references. Moreover, they reviewed my writings and provided key insights into possible improvements. They also provided input in finding proper formal semantics language for the conceptual modeling language and the GMF framework for implementation part of my thesis. Kwok Wong was involved in preliminary phase of designing the graphical notations for the conceptual visual language.

I finalized the formal semantics of the Conceptual Visual Language (CVL) and the mathematical formulations of the hierarchy and dimension in the CVL. I also came up with the real world examples that aptly back up those rigorous descriptions. I implemented the graphical editor for the Model manager using the Eclipse GMF framework.
I wrote the entire thesis by myself except the Architecture section (Section 2.2.2) in the Background chapter. The Architecture section in this thesis is directly taken from the Architecture section of [24].
# Table of Contents

Abstract ................................................................. ii  
Preface ................................................................. iii  
Table of Contents .................................................... v  
List of Figures ....................................................... viii  
Acknowledgments ..................................................... x  
Dedication ............................................................. xi  
1 Introduction ........................................................ 1  
1.1 Motivation ....................................................... 2  
1.2 Proposed Solution .............................................. 3  
1.3 Contributions ................................................... 4  
1.4 Organization of the Thesis .................................... 5  
2 Background ......................................................... 6  
2.1 Data Warehouse ............................................... 6  
   2.1.1 The Multidimensional Model ............................. 7  
   2.1.2 Traditional Process for a Data Warehouse Construction ............................. 8  
2.2 Conceptual Integration Modeling ............................ 10  
   2.2.1 CIM Visual Model ......................................... 12  
   2.2.2 Architecture ............................................... 17  
   2.2.3 Current Status and Future Goal ........................... 18  
2.3 User Interaction with the Model Manager .................. 20
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A data cube example ([2])</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>The star schema corresponding to the data cube from Figure 2.1</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>The landscape of models of the CIM framework [24]</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>Node symbols in CVL</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>Cardinality symbols in CVL</td>
<td>14</td>
</tr>
<tr>
<td>2.6</td>
<td>Product Purchase example</td>
<td>16</td>
</tr>
<tr>
<td>2.7</td>
<td>CIM framework functional architecture [24]</td>
<td>17</td>
</tr>
<tr>
<td>2.8</td>
<td>The current functional diagram of CIM [24]</td>
<td>18</td>
</tr>
<tr>
<td>2.9</td>
<td>The ultimate functional diagram of CIM [24]</td>
<td>19</td>
</tr>
<tr>
<td>2.10</td>
<td>User interaction with the Model manager [24]</td>
<td>20</td>
</tr>
<tr>
<td>2.11</td>
<td>A block diagram of the Model manager</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Calendar hierarchy schema</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>Calendar hierarchy instance</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>A Geography hierarchy</td>
<td>33</td>
</tr>
<tr>
<td>4.4</td>
<td>A DAG equivalent of Geographical location hierarchy</td>
<td>36</td>
</tr>
<tr>
<td>4.5</td>
<td>Different combinations of Disjoint constraints</td>
<td>42</td>
</tr>
<tr>
<td>4.6</td>
<td>An instance of Geography hierarchy (Figure 4.3)</td>
<td>45</td>
</tr>
<tr>
<td>4.7</td>
<td>A mapping derived from Figure 4.6 between levels Branch and Country</td>
<td>49</td>
</tr>
<tr>
<td>4.8</td>
<td>Different types of Aggregation Traversals</td>
<td>51</td>
</tr>
<tr>
<td>4.9</td>
<td>Equivalent multiple bottom levels and single bottom level hierarchies</td>
<td>55</td>
</tr>
<tr>
<td>4.10</td>
<td>Single level and single hierarchy dimensions</td>
<td>57</td>
</tr>
<tr>
<td>4.11</td>
<td>An example of Staff dimension with multiple hierarchies</td>
<td>58</td>
</tr>
</tbody>
</table>
Acknowledgments

I am forever indebted to Dr. Rachel Pottinger, my research supervisor, for her invaluable guidance and enthusiastic support throughout the duration of this research. Her timely advices and regular funding helped me to produce this work on time. I thank her for allowing me a freedom to explore and learn, which made this entire experience highly enjoyable and memorable.

I would like to extend my gratitude to Dr. Iluju Kiringa, Dr. Flavio Rizzolo, and Kwok Wong from the University of Ottawa, for their invaluable feedback and for helping me shape this work so well. Many thanks to Dr. Ed Knorr for reviewing my thesis and providing insightful comments.

My special thanks to Uncle Jiwan and Tim for their continual encouragement from inception to conclusion of this work, and being above just family and friends. Further, I am grateful to Tanis Gieselman for making this journey very enjoyable. Finally, I thank my friends from the Data Management and Mining lab for being supportive and providing a friendly environment to work in.

Dibesh Shakya
Vancouver
February 2011
Dedication

To my beloved parents and family,
for their endless love and untiring support.
Chapter 1

Introduction

From government to non-government organizations, private to public sectors, or local companies to multinational corporations, all of them generate or amass a huge quantity of data every year. This data can be used for tactical and strategic decision-making. As a result, making sense of the data at its disposal is a priority for an organization; it can provide deeper insights into the effectiveness of the existing operations and viability of potential future directions. Subsequently, this can determine a success or failure of the organization.

A decision-support system (DSS) provides an end-to-end solution to the problem of mining the vast quantity of data possessed by an organization. At the input end is the massive passive data at large; while, at the output end is the summarized information reporting the trends and patterns derived from those data. This data is often scattered in various heterogeneous data sources in structured (relational database, XML files, spreadsheets, etc.) or unstructured format (e.g., Web pages and documents), or both. Consolidating this data together into a central data store is important to analyzing it. This is where a data warehouse comes in. A data warehouse \[11\] \[17\] is a central physical repository of data that stores the data integrated from multiple sources. A data warehouse is constructed after a careful study of the local schemas of data sources to meet the needs of data analyses. A data warehouse consists of a set of data marts, each of which contains a subset of data from the data warehouse.

A data warehouse is traditionally built by gathering user requirements, design-
ing a logical level schema, identifying the data sources and populating the schema by the data transformed from the sources employing extraction and transformation techniques. Once established, the logical schema forms a queryable interface for accessing the data stored in the data warehouse. The most popular logical level for a data warehouse in the industry is the Star schema [17] which is based on a relational model. A DSS provides analytical tools, called OLAP tools, that query the logical schema and summarize the data thus obtained. There is a set of core queries that runs frequently and in batches. However, a user might have ad-hoc queries as well. Thus a logical schema has to cater to both OLAP tools and business users.

1.1 Motivation

The traditional approach for developing a data warehouse puts too much emphasis on the data in the data sources to be integrated and thus degenerates into a data-driven method. While designing a schema, a lot of focus is put on the implementation details and this distracts the designer from fully attending to the user-requirements. Additionally, one fully functioning data mart is developed at a time, then they are successively integrated. This leads to a bottom-up approach where conflicts of descriptions may arise across schemas of data marts for the same entity. Additionally, the traditional approach to design and develop a data warehouse is labor intensive. It takes an immense effort, data expertise and time to build a data warehouse. As a result, deploying a data warehouse in an organization is a highly expensive venture. We need an approach that allows a user with a knowledge of business requirements to articulate the requirements. Those requirements will semi-automatically pilot the data warehouse development process. This way implementation details will have fewer chances of interfering with the schema design process.

Equally pressing is the issue of presenting the data to a user in an easily consumable format. Business users are the target audience of a data warehouse so they should be able to build reports or write direct queries on top of the data warehouse easily. To that end, the data needs to be modeled in a way that is simple to understand and intuitive to use for even an average business user without any special database knowledge. A logical schema does not fit this description because
it is fraught with implementation details. This requires a conceptual schema that only talks the language at the level of business requirements without any contaminations of specific-implementation details underneath. Further, a data warehouse inherently demands a multidimensional model for typical business analysis. So, there is a need for a conceptual modeling language that caters to the multidimensional data and handles the gap between the business requirement specifications and the data storage details. Also, the lack of a conceptual schema creates a gap between an abstract representation used by an OLAP tool or visualized by a user and a logical schema of the data warehouse. This gap is called an “impedance mismatch” in data management parlance and it can be bridged by introducing a conceptual schema in the data warehouse. Moreover, the conceptual model can be used to drive a data warehouse development process as mentioned earlier.

There is no widely accepted conceptual modeling language for a data warehouse [26], let alone a prospect of a multidimensional conceptual model that can actually drive the data warehouse building process. The current popular methodologies of developing data warehouses, such as Kimball methodology [19], presents data only at a logical level to users.

1.2 Proposed Solution

In this thesis, we discuss a framework for constructing a requirement-driven data warehouse which is a collaborative work between the University of British Columbia and the University of Ottawa. This framework is called Conceptual Integration Modeling (CIM) and it provides a top-down approach to design a data warehouse where user requirements are captured in a graphical conceptual schema [24]. This conceptual schema significantly raises the level of abstraction and offers both design time and run time support for a data warehouse. The design time support enables construction of a data warehouse, and the run time support facilitates the populating and querying of the data warehouse. The multidimensional conceptual modeling language for CIM is proposed in [24] which is an extension of the MultiDim [21].

In this thesis, we work on the Model manager [24] component of the CIM framework. The Model manager deals with the conceptual modeling language
used in CIM. More specifically, we provide precise definitions of artifacts from the conceptual modeling language and clearly describe their usage scenarios. We discuss different conceptual models built using the artifacts covering the entire spectrum of simple models to complex ones. These models give users a clear idea of the capability of our conceptual modeling language. The formal semantics of the conceptual model is presented in denotational semantics. We also developed a fully functional prototype graphical editor for the Model manager [24] that enables us to draw such conceptual models using a drag-and-drop methodology. Further, we document in detail the implementation of the graphical editor in the Model manager. The editor was built using an Eclipse Graphical Modeling Framework (GMF) [8].

1.3 Contributions

The contributions of our work is summarized below:

- We provide precise definitions of the conceptual modeling language proposed for the CIM framework to dispel ambiguities about functionality of each concept.

- We present a gamut of conceptual models extending from simple examples to complicated ones to elucidate the capabilities and ease of use of the conceptual modeling language.

- We provide a formal semantics of the conceptual modeling language in denotational semantics to setup a strong mathematical basis for evaluating the language and future manipulations.

- We develop a fully functioning graphical editor for the Model manager for drawing a conceptual model and mapping it to the schema of data sources.

- We provide detailed documentation of the implementation and usage of the graphical editor in the Model manager.
1.4 Organization of the Thesis

We start by providing a background of data warehouses and the CIM project in Chapter 2. Then, we give a description of related work in Chapter 3 to familiarize readers with existing works on the multidimensional conceptual modeling problem. The precise descriptions of the artifacts from the conceptual model and the instances that exemplify their usages are described in Chapter 4. The formal definition of the basic artifacts from the conceptual language is provided in Chapter 5. Next, a documentation of the graphical editor in the Model manager for designing conceptual models is presented in Chapter 6. We conclude by summarizing our work and suggesting future work in Chapter 7.
Chapter 2

Background

In this chapter, we give a short introduction of a data warehouse and its role in a Decision Support System (DSS). A DSS provides a means to analyze enterprise data and derive information with high value of knowledge. In addition, a brief description of a conventional data warehouse design process is presented. We also describe the CIM project and its architecture. The role of the Model manager in the big picture of CIM is well described. We also provide the current status and the future goal of CIM.

2.1 Data Warehouse

A data warehouse is a specialized database that is designed to store a very large quantity of data, typically in terabytes. The term “data warehouse” was coined by Inmon in [17]. The interest in data warehouses began in 80s in the industry to develop a common repository for storing enterprise-wide data. This centralized data store is meant to be used for facilitating multidimensional analysis and reporting. As a result, by design it is expected to face complex queries that span a significant subset of data in it and accordingly, it is tuned for fast query answering.

A data warehouse is a central repository of data for an enterprise constructed by integrating the data scattered among multiple data sources by reconciling the conflicts, if any, in the data. The data sources are called operational or transactional databases because they store day-to-day transaction or operational data. But the
data in operational data source is not fit for analysis as this source is designed to support efficient insertions, deletions, and updates rather than retrievals. So, a data warehouse is modeled specifically to support multidimensional analysis queries. Stated differently, a data warehouse stores data used for multidimensional analysis.

2.1.1 The Multidimensional Model

A multidimensional data model views data as consisting of numerical data that can be analyzed on the basis of different business perspectives. Intuitively, this visualization takes a form of a hypercube (an n-dimensional cube) where the numerical values are the data points in the space and the business perspectives are the dimensions. This hypercube is called a data cube and the data points are called the facts. The business perspectives are called dimensions. Each dimension consists of a set of hierarchies and each hierarchy consists of a set of levels, each with a distinct granularity of detail. For instance, if we analyze sales in a store using business perspectives customer, time and product, then this forms a three-dimensional cube with numerical attributes of the sales like quantity or sales amount as facts; and dimensions customer, time and products provide context to the analysis of facts. An instance of this data cube is shown in Figure 2.1. As is evident from the figure, each dimension consists of a set of values. But this set of values can be classified into different granularities of detail. Now, let us consider the dimension Time in it. Here, the original values along the Time dimension are at the granularity of Days (not shown in the figure). But we can easily categorize days to months and perform an analysis at the granularity of months by grouping values of facts corresponding to days into months using summation as displayed in Figure 2.1. We can take this one step further to the granularity of Years as well. We call each granularity a “level”. Each level only stores the values having a single granularity of detail. In the parlance of multidimensional data analysis, when we propagate an analysis from a detailed level to a summarized level, we call it a “roll up”. On the other hand, a propagation of analysis from summarized level to a detailed level is referred to as a “drill down”. The Roll ups and Drill downs are crucial operations in multidimensional data analysis.

The facts are chosen as a focus of analysis to give some non-trivial insights into
the operation of the organizations. Various statistical operations can be applied to facts to measure the efficacy of the operations. These metrics being measured are called Key Performance Indicators (KPI) or Success metrics [17] because their results quantify the success factors of the organizations. As an example, a yearly revenue (sum of sales amount) shows how much business was done through sales every year by the organization under consideration. So, yearly revenue is a KPI.

2.1.2 Traditional Process for a Data Warehouse Construction

A widely adopted method of constructing data warehouses in industry is a bottom-up approach; it involves a lot of human expertise and involvement along the entire process, from the gathering of user requirements to the deployment of the data warehouse. A data warehouse can be integrated from a set of data marts [17]. Each data mart stores data for an analysis of a specific business process. The traditional method creates one data mart at a time. This process is briefly listed below:

1. Gather user requirements of data analysis for a department of an organization
and document them.

2. Design a logical level schema for a data mart that addresses the gathered requirements.

3. Identify the necessary data sources and map them to the logical schema.

4. Create data extraction and transformation scripts (ETL scripts) for migrating data from the sources to the logical schema.

5. Execute the scripts. This populates the data mart.

6. Regularly update the data mart with new data from the sources with a set of ETL scripts.

7. Repeat steps 1–6 for another department to create another data mart resolving any conflicts with schemas of the data marts developed in previous iterations.

So, a data warehouse is built gradually bottom-up from data marts growing with each iteration. An integrated logical level schema emerges as the highest level of abstraction against which queries will be directly formulated. As emphasized earlier, a data warehouse is modeling a multidimensional data model, so it borrows the same concepts of facts, dimensions, levels, etc., for creating a logical schema design. The most popular choice for the logical schema is the Star schema. The Star schema is a relational schema which has a table for fact data surrounded by other tables for dimensions. Each tuple in a fact table references corresponding tuples from the dimension tables. A star schema model for a data warehouse corresponding to the data cube from Figure 2.1 is shown in Figure 2.2. If a dimension table is denormalized into levels then the resulting schema is called a Snowflake schema. The lack of a conceptual schema brings about many disadvantages to a data warehouse. These disadvantages are explained in detail in the following section. We also discuss how Conceptual Integration Modeling mitigates those shortcomings.
Figure 2.2: The star schema corresponding to the data cube from Figure 2.1

2.2 Conceptual Integration Modeling

Conceptual Integration Modeling (CIM) [24] is a data warehouse research project that is a joint work of the University of British Columbia and the University of Ottawa. This project offers a framework for constructing a data warehouse with design time as well as runtime support. We call this framework: CIM framework [24]. The design time support enables construction of the logical schema of the warehouse, and the runtime support facilitates data population and querying of the warehouse. The CIM project proposes a semantic layer on top of the widely used logical schema. This semantic layer, also called a business layer, provides a representation of the data model that closely matches the data requirements of the business users in terms they understand. In other words, the semantic layer recognizes concepts of the multidimensional model like facts, dimensions, levels, hierarchies, etc. as the first-class constructs and allows expression of the user requirements using these constructs. This semantic layer is fully capable of articulating the data needs of the business users in the form that they desire and they comprehend. This drastically boosts user satisfaction when the final product is used. In the database world, we deploy a semantic layer using a conceptual schema. A domain specific
Figure 2.3: The landscape of models of the CIM framework [24]

language used to express the conceptual schema is called the conceptual modeling language. The conceptual modeling language in the CIM project enables us to conceptualize a multidimensional data model.

Since a conceptual schema captures a complete semantics of the user requirements from the functional point of view, it has the potential to specify an exactly complementing logical schema. CIM uses this motivation to automate the generation of a logical schema from the given conceptual schema. CIM also allows an expression of mappings between the conceptual schema and the schemas of the data sources. This mapping is leveraged to populate the logical schema. This focus on the conceptual schema makes the data warehouse construction requirements-driven. The automatic construction and population of the data warehouse translates to very rapid deployment of the data warehouse. So, unlike the conventional approach of labor intensive data warehouse deployment, this new approach enables constructing the entire data warehouse at once instead of one data mart at a time. This approach lets us design an entire data warehouse rather than by integrating individual schemas of data marts. A set of data marts can be generated by selecting subsets of the data warehouse. This approach is called a top-down approach. In previous approaches, logical schemas were handcrafted and the data warehouse was populated manually, which made the top-down approach very slow; and thus, it could not achieve a wide acceptance in the industry. In conclusion, the CIM framework provides a top-down approach of constructing a requirements-driven data warehouse leveraging a semantic layer.

The conceptual modeling language in CIM has two facets: (1) CIM Visual Model (CVM) and (2) CIM Data Model (CDM). Each of the two models consists of three distinct languages as shown in Figure 2.3. The descriptions of the languages in the CVM are described below:

<table>
<thead>
<tr>
<th>CIM Visual Model (CVM)</th>
<th>CIM Data Model (CDM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVL    : Conceptual Visual Language</td>
<td>CDL    : Conceptual Data Language</td>
</tr>
<tr>
<td>MVL    : Mapping Visual Language</td>
<td>MDL    : Mapping Data Language</td>
</tr>
<tr>
<td>SVL    : Store Visual Language</td>
<td>SDL    : Store Data Language</td>
</tr>
</tbody>
</table>
• **Conceptual Visual Language (CVL)**: The conceptual visual language is a visual representation of the conceptual language. CVL is inspired from MultiDim [22] and StarER [27]. The latter two models are, in turn, derived from the ubiquitous ER model [12].

• **Store Visual Language (SVL)**: SVL is a visual representation of the (relational) multidimensional language.

• **Mapping Visual Language (MVL)**: This language represents the visual mapping for translating the conceptual model into the multidimensional model.

Similarly, the languages in the CDM are described as follows:

• **Conceptual Data Language (CDL)**: CDL is a counterpart of CVL. It is the internal representation of CVL in the XML format.

• **Store Data Language (SDL)**: SDL is a counterpart of the SVL and forms the internal representation of SVL in the XML format.

• **Mapping Data Language (MDL)**: MDL is a counterpart of MVL. It is the internal representation of MVL in the XML format.

These internal representation languages allow easy manipulation and a loose coupling from the visual representation. The CDM is also more suited for Business Intelligence applications. We will not further delve into the CDM; interested users are encouraged to reference [24]. In this thesis, we only focus on the CVM. The semantics of the CDM will be derived from the that of the CVM. But a model in CDM should be derivable from a model in CVM and vice-versa.

### 2.2.1 CIM Visual Model

The section describes different symbols used in languages belonging to the CIM Visual Model to represent CVL, SVL, and MVL. A meaningful composition of these symbols generates a conceptual model, a store model and the mapping between them.

We start off by describing the symbols used in CVL. Figure 2.4 displays the constructs of the multidimensional data model that are represented as nodes. A fact
relationship (Figure 2.4(a)) and a dimension (Figure 2.4(b)) are represented by a shadowed diamond symbol and a shadowed rectangle symbol respectively. Similarly, a level (Figure 2.4(c)), a hierarchy (Figure 2.4(d)), and a property/measure (Figure 2.4(e)) are depicted using a plain rectangle, a rounded corner rectangle, and an ellipse, respectively. All of these aforementioned symbols have names associated with them. A property has other attributes such as its data type, two boolean attributes showing if it is a primary key and whether it is nullable or not. CIM uses Figure 2.4(f) symbol to represent a disjoint integrity constraint. More on this symbol will be said in Section 4.4. A parent-child relationship is represented using a line with cardinalities as shown in Figure 2.5.

SVL uses a UML-like representation used for the relational model with a big rectangle for a table and a stack of rectangular columns inside the comprising table symbol. Each table has a name, and column has a name and data type. An arrow is used to show a foreign key dependency.

A dashed line is used to represent a mapping between a property from a CVL model to a column in SVL model. Any condition for a mapping is shown with text attached to the dotted line for the mapping.
**Example 2.2.1** Let us consider a very simple conceptual schema consisting of one fact, one dimension and one hierarchy. Usually, a multidimensional model consists of numerous dimensions connected to a fact table. But, here, to simplify the diagram, we only show one dimension. The CVL diagram is shown inside a CVL box (a box with header CVL) and the SVL diagram inside an SVL box. A fact *Purchase* consisting of a fact property *TransactionID* and a measure *Qty* is analyzed using a dimension *Location*. This dimension *Location* consists of a hierarchy *Geography*. This hierarchy *Geography* consists of level *Branch*, *State*, *Province* and *Country* connected to each other. Each level consists of properties shown with ellipses. The underline under the name of property shows that it is a primary key of the comprising level. The cardinality in the parent-child relation shows the participation count constraint. In the diagram, level *Province* can have one or more branches whereas one branch can only belong to at most one province. Similarly, the SVL box contains the SVL artifacts. The tables *Country*, *StateProvince*, *Branch* and *Purchase* are shown along with the columns in these tables. The arrows between columns show foreign key dependencies. The blue dotted-line connecting a property from a CVL model to a column in a SVL model shows the mapping and the condition of mapping included as text near the mapping.
line. For instance, property CountryName from the level Country is mapped to a column Name in the table Country. This mapping has a predicate “Continent=“N. America”” as shown in the figure.
Figure 2.6: Product Purchase example
2.2.2 Architecture

We can now describe the architecture of the CIM Framework as is currently implemented in CIM. The main components of this architecture are given in Figure 2.7.

The functionality of the components is described as follows:

- **Storage Manager.** This component creates the SVL models automatically from the metadata exported from the underlying data warehouse. Currently, CIM uses Mondrian [1], an open source ROLAP engine, for this purpose. The component also maintains the cube views created by the View Compiler component.

- **Model Manager.** This component takes a CVM model as input (i.e., the SVL model generated by the Store Manager, and the CVL and MVL models provided by the user). The component produces a CDM model (i.e., CDL, MDL, and SDL specifications) as output. Any changes in the CVM model by the user will be incrementally propagated to the CDM.

- **View Compiler.** This component takes the CDL, MDL, and SDL models generated by the Model Manager, and produces the multidimensional views over the SDL model. The component first creates relational views over the SDL model, based on the mappings (MDL) for each fact relationship, level and parent-child relationship in the CDL model. Second, it restructures the hierarchies into summarizable ones [16]. Finally, it creates multidimensional SDL views based on the relational views from the first step and the summarizable hierarchies from the second one. These views allow queries posed against a CDL schema to be rewritten and posed against them.
• **Query Processor.** The query processor rewrites user queries in terms of the multidimensional views created by the View Compiler and sends the rewritten query to the Storage Manager. This component treats the processing of queries as an instance of the classical problem of processing queries using views.

Users interact with the Model Manager and the Query Processor components via a Graphical User Interface (GUI). All modules communicate with each other via a service bus. For the first prototype under implementation, this bus is simply the operating system’s file system where the models are stored and read in the form of XML files. The future plan is to replace the file system by Web services in a service-oriented architecture. Finally, the Query Processor component communicates directly with the Storage Manager component via a well-defined application programming interface.

### 2.2.3 Current Status and Future Goal

A current development of the CIM framework works as shown in Figure 2.8. On the left side (CVM) is the visual model emphasizing interaction with users. A user designs a CVL model and maps the model with the store model of the data source. The user can directly query the CVL model to analyze the data and the result will
Figure 2.9: The ultimate functional diagram of CIM [24]

be returned in the form described in the CVL model. On the right side (CDM) is the internal representation of the conceptual data model along with store and mapping data models. The XML languages are used for data model languages to enable easy interaction with Business Intelligence (BI) tools. At the center is the CIM framework that performs all the heavy-duty work of leveraging the mapping and store models to answer user queries. In the current implementation of CIM, it assumes a pre-existing relational data warehouse at the backend as a data store. This means that building and populating part of the data warehouse is already taken care of. So, this implementation focuses on expression of the user requirements in the conceptual models and efficient query answering leveraging the mappings. This solution can be deployed in circumstances where an enterprise already has a data warehouse but with an overwhelming logical schema. The semantic layer gives a simplified interface to use the data warehouse for a query formulation.

The future goal of CIM is to get rid of the assumption that demands pre-existence of the data warehouse as a data source. This scenario is shown in Figure 2.9. Here, the mapping of the conceptual model is done against the heterogeneous data sources. The presence of multiple sources creates the opportunity for a lot of hairy problems during data integration. To name a few problems: schema het-
ergogeneity, data inconsistency, and redundancy. A data warehouse needs to be designed and built (populated) as automatically as possible.

### 2.3 User Interaction with the Model Manager

Now, let us focus on the usage scenario of the Model manager. The Model manager, being a graphical editor, is a facade of the CIM framework and hence, allows a user to interact with the CIM framework. A user (possibly a business user) draws a graphical CVL diagram utilizing the drag-and-drop feature of the GUI interface. The Model manager is provided with an SDL model of the underlying multidimensional data source by the framework. This SDL model is transformed into the SVL model and visualized side-by-side with the CVL model designed by the user. This enables the user to perform mapping between the CVL model and SVL model by drawing lines between them. This visual mapping is stored in the MVL model. The input visual model, namely, the CVL model and the MVL model are transformed into their data model counterparts: the CDL and MDL models. Now, we have a complete set of three CVM models (CVL, SVL, and MVL) and three CDM models (CDL, SDL, and MDL). The CDM models are used by the framework for its internal functioning. The discussion of the internal functioning of the framework is beyond the scope of this thesis. Interested readers are encouraged to reference [24].

From the block diagram of the Model manager depicted in Figure 2.11, it is clearly evident that there are two functional components of the Model manager: the Graphical editor and the Transformation engine. The responsibility of the Graphical editor is to allow users to create conceptual visual models using the graphical tools. The editor also visualizes the Store visual model derived from the Store data
model. Now, a mapping can be performed between the conceptual visual model and the store visual model. So, the Graphical editor produces the CVL and MVL model files. These model files are transformed into data models, the CDL and MDL models, by the Transformation engine. The Transformation engine also allows the conversion of the SDL model into the SVL model. These two components constitute the Model manager. In this thesis, we only implement the Graphical editor component and leave the Transformation engine as future work.
Chapter 3

Related Work

The chapter reviews conceptual modeling languages for multidimensional data models from other researchers.

3.1 Conceptual Modeling Language

The need for a conceptual modeling language for information systems has been recognized for a long time in both academia and industry. For operational databases, the design process that progresses from conceptual model to logical model to physical model has achieved a “textbook” status. The Entity-Relationship (ER) model introduced by Chen in his seminal paper [12] is a ubiquitous language for designing conceptual models for operational databases. The ER model captures data from the real world in the form of entities and the relationships among them. Non-technical users understand this representation without much effort. This gives the conceptual model its “self-documenting” property. The basic knowledge of the syntax of the language and the nature of the problem domain suffices understanding what data is stored and in what form. This conceptual model translates in a straightforward fashion to a relational logical schema in third normal form (3NF). Adding the implementation tool dependent constructs to the logical schema gives us a physical schema. Almost all the major database vendors such as Oracle, Microsoft, Sybase, etc. provide the design tools to automate this three-step process. But, the data warehouse community has not had similar success. There has been a lot of
effort in the past two decades to establish a conceptual modeling language for a multidimensional data model. But a consensus on formalism or even a common terminology is yet to emerge [26].

Many mathematical models have been proposed for the multidimensional data model, both graphical [14, 22, 25–27] and non-graphical [13]. All these data models have tried to take advantage of fundamental concepts of multidimensionality such as facts, dimensions, hierarchies, levels, etc. This makes these mathematical models distinct from the ER model. Hence, we need a special multidimensional conceptual model that treats those fundamental concepts as first-class constructs. Since we work with a graphical conceptual language, CVL, in this thesis, only graphical models are reviewed in this chapter. The most noteworthy modeling languages from our point of view are discussed in [14, 22, 25–27]. Out of these graphical models, MultiDim [22], M E/R [25] and starER [27] are extensions of the ER model [12]. They extend the ER model to derive their own model to take advantage of the proven track record of the ER model and to simplify the learning process for users already familiar with the highly popular ER model. The multidimensional model MD was proposed by Torlone [26].

Below, we point out the similarities and differences between different aspects of the five graphical models: MultiDim, M E/R, MD [26], starER and CVL.

- **Cardinality in a parent-child relationship.** Only the MultiDim, starER and CVL models allow us to graphically express a participation cardinality in a parent-child relationship. A cardinality is a constraint on the schema, and adding it to the graphical notation increases the expressibility of the model. The increased expressibility of the model can be used to assess feasibility of an aggregation from the schema.

- **Structure of a hierarchy.** Hierarchy is a very central concept in multidimensional data modeling because it dictates the kinds of analyses that can be performed on the data model. MD describes a hierarchy as a linear chain of parent-child relationships. This means a child level can not have two parent levels in one hierarchy in the MD. On the other hand, the MultiDim, M E/R, starER and CVL models allow a lattice of levels to form a hierarchy. A lattice allows any directed acyclic graph including the linear paths.
• **Multiple hierarchies in a dimension.** To enable different ways to aggregate the bottom level of a dimension, all of the models allow multiple hierarchies within a dimension. They also allow the sharing of levels between hierarchies through overlap to prevent redundant descriptions of the same set of entities in the schema.

• **Non-strict hierarchy.** A hierarchy is said to possess the property of non-strictness if there is at least one parent-child relationship in it that allows many-to-many relationships between members of the parent and the child levels. Some real world scenarios demand this hierarchy but this property of non-strictness significantly complicates an aggregation process. MD does not support non-strict hierarchies; whereas, rest of the models do. But, none of them provide a generic solution for an aggregation within the non-strict hierarchy that applies in all circumstances. MultiDim attempts to solve an aggregation problem by introducing a distribution factor symbol in a many-to-many parent-child relationship allowing us to state the percentage by which a child level member distributes value of a measure to all its parent members. But, the sum of the distribution percentage must always be 100%.

• **Differentiating between a property and a measure in a fact relationship.** A fact relationship can contain some properties which can not be numerically analyzed, such as TransactionID for the Sales fact table. In such cases, knowing which attributes of a fact relationship are just descriptive and which are numerically analyzed prevents errors from a user while formulating a query. The specialized properties that can be numerically manipulated are called measures. MD and CVL models differentiate between a property and a measure. On the other hand, MultiDim, M E/R and starER only allow measures in a fact relationship.

• **Support for heterogeneous hierarchies.** The homogeneous hierarchies have an assumption that all parent-child relationships are total functions (i.e., each of the members of a child level has a mandatory participation in a parent-child relationship). Although, this assumption greatly simplifies an aggregation process, it restricts the real world scenarios that can be cap-
tured. Any hierarchy that is not homogeneous is heterogeneous. None of the models claim a restriction to allow only homogeneous hierarchies. But only MultiDim, MD and CVL explicitly claim to support heterogeneous hierarchies for aggregation.

- **Assign a role to an involvement of a dimension in a fact relationship.** One dimension can participate more than once, portraying different roles in a fact relationship. For example, the Time dimension can participate in the Sales fact relationship twice: once as an order time and once as a shipping time. Assigning a role to each participation helps to identify what part a dimension is playing for that particular involvement. MultiDim, MD and CVL explicitly allow a role of dimension to be specified in the graphical diagram whereas starER does not.

- **Hierarchy symbol.** Adding a hierarchy symbol to a graphical model enables a user to promptly identify hierarchies in a dimension. Only CVL and MultiDim use a hierarchy symbol explicitly in the graphical model. Although, MultiDim calls a hierarchy an “analysis criteria”, it is semantically similar.

- **Separation between declarations of dimension and its bottom level.** As will be explained in Section 5.6 separate identities of a dimension and its bottom level help to increase the expressibility of the model. All of MultiDim, MD, M E/R and starER implicitly assume the bottom level to represent a dimension. CVL adds a separate notation for a dimension.

- **Formal Semantics.** To the best of our knowledge, only MultiDim and CVL define their formal semantics. M E/R borrows its formal semantics from the ER model because M E/R is a specialization of the ER model. starER and MD have not defined their semantics formally.

- **Mathematical treatment of hierarchies and dimensions.** MD provides a mathematical model for a dimension but not for a hierarchy. It just defines hierarchy as a lattice of levels in a plain language. MultiDim, M E/R and starER do not provide any rigorous mathematical definitions for either hierarchy or dimension. Only, CVL provides formal mathematical defini-
tions for both hierarchy and dimension preventing any ambiguities in their descriptions.

None of the aforementioned models except CVL provides runtime support for the conceptual model. All of them use the conceptual models for designing data warehouses. Further, the graphical notations for the CVL model are inspired from the MultiDim and starER models.

The Entity Data Model (EDM) framework [10] developed by Microsoft provides a conceptual layer for operational databases. This conceptual layer is an extension of the ER model. Once this conceptual layer is mapped to the underlying data source, queries can be directly formulated against the conceptual layer. The EDM uses the Conceptual Schema Definition Language (CSDL) for describing a conceptual schema, the Storage Schema Definition Language (SSDL) for a logical level schema and the Mapping Definition Language (MDL) for a mapping between a conceptual and a logical level schema. CIM follows the similar path by using CDL, SDL and MDL as the data languages for conceptual, logical and mapping schemas, respectively.

To sum it up, our conceptual modeling language in CVL combines the best features from previous researches. This makes our CVL capable of representing any real world scenario supported by those previous models making our CVL at least as expressible as any of them. Additionally, the formal semantics and the rigorous mathematical treatment of the model make it robust and clear.
Chapter 4

Conceptual Visual Language

4.1 Definition
In this chapter, we describe the syntax of the constructs of the conceptual visual language (CVL) informally and then follow them by their mathematical definitions. Based on those descriptions, we also create examples presenting real world examples manifesting their capabilities.

4.2 Constructs of CVL
The constructs of CVL are the building blocks of the CVL models. These constructs consist of level, parent-child relationship, hierarchy, dimension, fact relationship, properties of levels and fact relationships, and disjoint constructs.

A level represents a set of real-world entities having the same granularity of detail and set of characteristics. For instance, if a level Product represents a set of items then all the items in the Product level provide the same fineness of detail. Instances of a level are called its members. Each characteristic of a level is described using a property. Two levels can be related through an association where the more granular level forms a child and the other one forms a parent.

In CIM, we describe a hierarchy schema as a lattice of levels (a subset) belonging to a dimension that consists of one bottom level and one or more top levels. The adjacent levels in the lattice are connected through parent-child relationships.
These parent-child relationships may form an aggregation or a specialization relationship. If a parent-child relationship forms an aggregation then out of the two involved levels, the level with finer granularity forms the child and the level with coarser granularity forms the parent level. For example, if levels *Product* and *Category* have a parent-child relationship between them, then *Product*, possessing a finer granularity of detail, forms a child level and consequently, the *Category* assumes a role of a parent. In case of a specialization relationship, a specialized level is a subset of a more general level. We assign a parent role to the specialized level and a child role is assumed by the generalized level. A level *Manager* is specialized from level *Employee*, so a parent-child relationship between them identifies *Manager* as a parent and *Employee* as a child. A hierarchy schema is formed by connecting a group of levels through parent-child relationships. Precisely, this lattice forms a directed acyclic graph (DAG) where levels are the vertices and parent-child relationships are edges such that each edge points from a child level to a parent level. Each of these edges in the DAG shows the direction in which an aggregation can be performed in the hierarchy schema. A hierarchy schema is not sufficient to capture integrity constraints applied on the parent-child relationships, such as a disjointness between two or more relationships. Consequently, these integrity constraints, called hierarchy constraints, are imposed on a hierarchy schema. Hierarchy is an accumulation of a hierarchy schema and the hierarchy constraints applied on the schema. Our description of a hierarchy aligns itself with the definitions of hierarchies as a lattice of levels as given in [16, 18, 20, 21].

**Example 4.2.1** Figure 4.1 depicts a typical example of a hierarchy named *Calendar* consisting of levels *Day*, *Week*, *Month*, *Quarter* and *Year*. Some of these levels form pairings (shown by edges) resulting in parent-child relationships between them. For instance, *Week-Day*, *Month-Day*, *Quarter-Month* etc. form parent-child relationships. Figure 4.1(a) shows the CIM representation of a hierarchy. Figure 4.1(b) shows the same hierarchy from Figure 4.1(a) in a DAG representation by replacing parent-child relationships by arrows directed from a child level to its parent level.

In addition to showing a direction of aggregation between two levels, a parent-
child relationship also enables the storage of a cardinality constraint (like in the ER model [12]) for the relationship (i.e., the upper and lower bounds on participation count of members of the involved levels). The bottom level of the hierarchy stores data of the finest granularity and the top levels store data aggregated to the coarsest (most summarized) granularity. Only the bottom level participates in a fact relationship. As we trace the levels sequentially from the bottom level to a top level, the coarseness of granularity of data increases monotonically along the intermediate levels.

**Example 4.2.2** In hierarchy Calendar (Figure 4.1), Day is the bottom level and Year is a top level. The coarseness of granularity of data increases from Day to Week and from Week to Year. Also, a level can have one or more parents such that the child level can roll up to all of its parent levels. The level Day in Figure 4.1 rolls up to levels Week and Month both. Similarly a level can have one or more child levels as shown in Figure 4.1 through Year-Week and Year-Quarter relationships.
Now, let’s talk about an instance of a hierarchy. In any instance of the hierarchy, a member of a child level rolls up to a member of the parent level along a parent-child relationship. These rollups between members of any two levels are defined by a rollup relation [16] between the domains of members of those two participating levels. A rollup relation is a (possibly partial) binary relation from a child level to a parent level. The rollup relation is an artifact of a hierarchy instance which indicates how elements of a child level clusters onto the elements of its parent level. For each rollup relation there must be a corresponding parent-child relationship in the hierarchy schema.

**Example 4.2.3** Consider a hierarchy instance (Figure 4.2) of the hierarchy schema *Calendar* from Figure 4.1. The members of the same levels are shown within the dotted line boxes, and edges between nodes display rollup relationships between the members of levels. For example, the level *Day* consists of members 2010Nov23, 2010Nov29, 2010Dec2 and 2011Jan1; and level *Week* consists of members 2010W49, 2010W52 and so on. The members of *Day* roll up to members of *Week* constituting a rollup relation and the rollup between members is shown by edges between those members. Note that in this example, member 2010W52 of level *Week* rolls up to two members of a parent level *Year* (week 2010W52 spans the end of the year 2010 and the start of year 2011), and also a parent member 2010 in *Year* has two child members 2010W49 and 2010W52 in level *Week*.

A hierarchy schema is a DAG with one bottom level so any node in the schema can be reached from the bottom level by following a chain of parent-child relationships transitively. Each of such chains is referred to as an aggregation path because the aggregated measure for an element of any level can be calculated by following the levels in the chains (containing the desired level) sequentially from the bottom level. Further, a level might be reached from the bottom level following numerous paths. A rollup path describes a chain of rollups that can be used to establish the transitive relationships from the bottom level to any desirable level in a hierarchy instance. A rollup path must have a corresponding aggregation path in the hierarchy.

In general, a rollup relation in CIM does not put any restriction on a number of parent members to which a child member can associate, and the same holds true.
for a drill down from a parent to its child level(s) as well. But allowing multiple parents in one level for a child member leads to a double counting and hence an inconsistency during an aggregation of the parent level [23]. When multiple rollup paths link a bottom level element with different elements in the same ancestor level leading to repeated counting of the bottom level element at the ancestor level, we say a double counting of the bottom level member has occurred. If a hierarchy allows a member from a bottom level to reach more than one member in an ancestor level, then such hierarchy is called a non-strict hierarchy [9]. A non-strict hierarchy is said to possess a property of non-strictness. The non-strictness can occur at either a parent-child relationship or at an ancestor-descendant relationship. A one-to-many or many-to-many cardinality relationship from a child to a parent level enables us to attain a non-strict hierarchy. This allows non-strictness at a parent-child relationship and it is trivial to detect. It can be prevented by enforcing one-to-one or many-to-one cardinality relationships for all parent-child relation-

Figure 4.2: Calendar hierarchy instance
ships in a hierarchy. However, detecting non-strictness at an ancestor-descendant relationship is hard and consequently difficult to prevent. This occurs when a member from a finer level rolls up to different members in its ancestor level through the same or different rollup path(s). Authors in [23] investigates in detail the consequences of allowing a non-strict hierarchy and advises against using it. On the other hand, there exists a subset of hierarchies in which regardless of paths followed to roll up an element from the bottom to the desired coarser level, the rollup always ends at the same element at a higher level. This is a very important property of a hierarchy (called strictness [9] of a hierarchy) which says an element from the bottom level always roll up to at most one element at any other level in the hierarchy. This strictness condition must be satisfied to prevent the inconsistency during aggregation that might result from choosing between inconsistent rollup paths [9].

**Example 4.2.4** Usually, the last week in a calendar year spans partially across two years. Let us consider parent-child relationship Year-Week from Example 4.2.3 (Figure 4.2) where level Week rolls up to level Year. The last week of year 2010, 2010W52, starts on Monday December 27, 2010 and ends on Sunday January 2, 2011. So, the member of week rolls up to two members (2010 and 2011) in level Year. This results in a non-strict hierarchy because week 2010W52 counts twice: once in year 2010 and again in year 2011. So any measure associated with the week 2010W52 is counted twice in the level Year inducing double counting. Further, when we aggregate all the members in the entire level Year, we get a different value of the summarized measure than what we would get by aggregating all the members in the Week level, which is an inconsistent result.

**Example 4.2.5** In the Geography hierarchy (Figure 4.3) all the parent-child relationships are many-to-one from a child level to its parent level. Thus, no members from a child level can roll up to more than one parent member making the hierarchy strict. In each rollup relation, a child member has at most one parent member. As a result, Geography is a strict hierarchy.

In the following sections, we give formal definitions of the key components of hierarchy, dimension and their instances in CIM.
Assume the existence of a Hierarchy schema $H$ in dimension $D$ consisting of a set of Levels $L$. Let us suppose $L \subseteq L$ and $\text{dom}(l)$ denotes a set of members of level $l \in L$. We maintain a logical and syntactic coherence with the definition of a hierarchy schema in [16], differing by exclusion of implicit single Top level All and shortcut-free restriction, and by inclusion of cardinalities in the parent-child relationships.

**Definition 4.3.1 (Hierarchy Schema).** A Hierarchy schema $H \in D$ is a DAG defined as a tuple $(L, \nearrow)$ where $L$ is a set of levels and $\nearrow$ is a partial order\(^2\) on $L$. Each tuple, consisting of a pair of levels from $L$, in the $\nearrow$ denotes a parent-child

\(^2\)A partial order on a set $S$ is a set of binary tuples $(s_i, s_j)$ such that $s_i, s_j \in S$ and $s_i$ precedes $s_j$ in $S$. 

---

Figure 4.3: A Geography hierarchy

\[ 
\text{Country} \\
\text{State} \quad \text{Province} \quad \text{SalesRegion} \\
\text{City} \\
\text{Branch} 
\]
relationship from the first element to the second one within the pair. If \( l_i, l_j \in L \) and \( l_i \supseteq l_j \) then there is a link from \( l_i \) to \( l_j \) in the DAG. The granularity of \( l_j \) is at least as coarse as that of \( l_i \). \(^3\) \[16\]

A hierarchy schema for hierarchy named \( X \) is represented as \( H_X \).

Since \( \supseteq \) is a partial order on \( L \), this renders \( L \) a partially ordered set. That means \( \forall l_i, l_j \in L : l_i \supseteq l_j \) then \( l_i \) must precede \( l_j \) in \( L \). \( H \) has a distinguished level \( l_{BOTTOM} \in L \), a bottom level, such that no other level rolls up to it (i.e., \( \neg \exists l \in L : l \supseteq l_{BOTTOM} \)). Also, \( H \) has a set of distinguished levels \( L_{TOP} \subseteq L \) called top levels such that these levels do not roll up to any other level in \( H \), \( L_{TOP} = \{ l_i \in L | \neg \exists l \in L : l_i \supseteq l \} \). \( \supseteq^a \), called an aggregation path, denotes a reflexive and transitive closure of \( \supseteq \). All levels of the hierarchy can be reached from \( l_{BOTTOM} \) (i.e., \( \forall l \in L : l_{BOTTOM} \supseteq^a l \)). Each parent-child relationship \( \supseteq \) also stores participation cardinalities for both child and parent levels. Hence, \( \supseteq \) is defined as an ordered tuple \( \langle l_i, l_j, (n_i, m_i), (n_j, m_j) \rangle \) where \( l_i, l_j \in L \) and \( l_i \supseteq l_j \) and \( n_i, m_i, n_j, m_j \in \mathbb{Z}^+ \cup \{\ast\} \) (\( \mathbb{Z}^+ \) is the set of non-negative integers and \( \ast \) is used for an unbounded upper limit) such that \( n_i \leq m_i \) and \( n_j \leq m_j \). \((n_i, m_i)\) is a participation constraint for \( l_i \) which means one \( l_i \) should relate to at least \( n_i \) and at most \( m_i \) number of \( l_j \) members. Similarly, \((n_j, m_j)\) is a participation constraint for \( l_j \). If the lower bound of cardinality \( n_i \) or \( n_j \) equals zero then it has optional participation. For a strict hierarchy, \( \forall \supseteq_i \text{ in } \supseteq: n_i = \{0,1\} \land m_i = 1 \text{ where } \supseteq_i = \langle l_i, l_j, (n_i, m_i), (n_j, m_j) \rangle \).

**Example 4.3.1** Reusing the DAG from Example 4.2.1, we show a hierarchy schema representation \( H_{Calendar} = (L, \supseteq) \) for it as follows:

\[
L = \{\text{Day}, \text{Week}, \text{Month}, \text{Quarter}, \text{Year}\}
\]
\[
\supseteq = \{\langle \text{Day}, \text{Week}, (1, 1), (1, \ast) \rangle, \\
\langle \text{Week}, \text{Year}, (1, \ast), (1, \ast) \rangle, \\
\langle \text{Day}, \text{Month}, (1, 1), (1, \ast) \rangle, \\
\langle \text{Month}, \text{Quarter}, (1, 1), (1, \ast) \rangle \\
\langle \text{Quarter}, \text{Year}, (1, 1), (1, \ast) \rangle \}
\]

\(^3\)If a child level splits/specializes into multiple parent levels preserving granularity of the child level then the parent levels have the same coarseness as the child level.
The parent-child relationship \( \langle \text{Day}, \text{Week}, (1,1), (1,*)) \) represents that a day can belong to one week but a week has one or more days. Further, \( \langle \text{Week}, \text{Year}, (1,1), (1,*) \rangle \) says a week belongs to at least one year and a year has one or more weeks; and so on for other parent-child relationships. Due to the presence of the parent-child relationship \( \langle \text{Week}, \text{Year}, (1,1), (1,*) \rangle \), a week can roll up to multiple years. As a result, this is a non-strict hierarchy.

**Example 4.3.2** A DAG representation of a Geography hierarchy in Figure 4.3 is shown in Figure 4.4. The Hierarchy schema \( H_{\text{Geography}} \) corresponding to Figure 4.3 consists of a set of Levels \( L \) and a set of parent-child relationships \( \nearrow \) as follows:

\[
L = \{ \text{Branch}, \text{City}, \text{State}, \text{Province}, \text{SalesRegion}, \text{Country} \}
\]

\[
\nearrow = \{ \langle \text{Branch}, \text{City}, (1,1), (1,*) \rangle, \\
\langle \text{City}, \text{State}, (0,1), (1,*) \rangle, \\
\langle \text{City}, \text{Province}, (0,1), (1,*) \rangle, \\
\langle \text{City}, \text{SalesRegion}, (0,1), (1,*) \rangle, \\
\langle \text{State}, \text{Country}, (1,1), (0,*) \rangle, \\
\langle \text{Province}, \text{Country}, (1,1), (0,*) \rangle, \\
\langle \text{SalesRegion}, \text{Country}, (1,1), (0,*) \rangle \}
\]

The parent-child relationship \( \langle \text{Branch}, \text{City}, (1,1), (1,*) \rangle \) says that a member of level \( \text{Branch} \) must relate to one member of level \( \text{City} \) whereas one member of \( \text{City} \) can have one or more branches. Similarly, \( \langle \text{City}, \text{State}, (0,1), (1,*) \rangle \) means that a city can belong to at most one state whereas a state must have at least one city; and so on for other parent-child relationships. A parent-child relationship has a rollup relation associated with it. In none of the parent-child relationships, a child can roll up to more than one parent member. This makes Geography a strict hierarchy.

**Definition 4.3.2 (Rollup Relation).** A rollup relation is an instantiation of a parent-child relationship \( l_i \nearrow l_j \) where \( l_i, l_j \in L \) and defined using the domains of the child and parent levels as a binary relation \( RUP_{l_i}^{l_j} : \text{dom}(l_i) \rightarrow \text{dom}(l_j) \). The instantiation of \( l_i \nearrow l_j \) as \( RUP_{l_i}^{l_j} \) is denoted as \( RUP_{l_i}^{l_j} \Rightarrow l_i \nearrow l_j \). Thus ‘\( \Rightarrow \)’ is referred to as an instance of symbol.
A rollup relation reduces to a function, when the corresponding parent-child relationship allows at most one parent for each child member. As a result, all the rollup relations in a strict hierarchy are mathematical functions.

**Example 4.3.3** In the Calendar hierarchy instance (Figure 4.2), the rollup relation $RUP_{Day}^{Week}$ corresponding to the parent-child relationship $Day \nearrow Week$ is expressed as follows:

$$RUP_{Day}^{Week} = \{(2010Nov23, 2010W49),$$
$$\quad (2010Nov29, 2010W52),$$
$$\quad (2010Dec30, 2010W52),$$
$$\quad (2011Jan01, 2010W52)\}$$

**Figure 4.4:** A DAG equivalent of Geographical location hierarchy
4.4 Hierarchy Constraints

In a DAG, an edge describes a relationship between the nodes it connects but there are scenarios where we may want to relate different edges through some dependencies. There is no such notion of dependencies between edges in a DAG and the same applies for a hierarchy schema as well. Thus, the definitions of parent-child relationships are not sufficient to capture all the business constraints that cross-cut multiple parent-child relationships in a hierarchy schema.

Example 4.4.1 Let us consider a group of parent-child relationships from the Geography hierarchy (Figure 4.3) such that a City level member can roll up through only one relationship among the group (City ↗️ State, City ↗️ Province) thus enforcing an exclusive rollup path for a member of the City level within the group of paths. This exclusivity (disjointness) of rollup within the group is not captured by the equivalent DAG (Figure 4.4).

The description of a hierarchy schema can be enhanced by incorporating the integrity constraints specifying the valid relationships between different parent-child relationships in a hierarchy schema [16]. All valid hierarchy instances over the hierarchy schema should follow these constraints to maintain consistency. The integrity constraints that ensure the consistency of a hierarchy instance by specifying legal rollup paths in it are called hierarchy constraints. The hierarchy constraints cross-cut the different parent-child relationships constituting the hierarchy schema so these constraints belong to the hierarchy schema as a whole rather than to any individual relationship. A hierarchy constraint checks whether a group of parent-child relationships are consistent with each other. The disjoint relation between parent-child relationships City ↗️ State and City ↗️ Province in Figure 4.3 can be represented as

\[(\text{City} \rightarrow \text{State}) \otimes (\text{City} \rightarrow \text{Province})\]

The symbol ‘⊗’ is used to denote disjointness. Each member of level City, a common level between two parent-child relationships, can roll up through only one of the parent-relationships. Those hierarchy instances over the hierarchy schema that violate this hierarchy constraint are invalid instances of the schema. This is an example of a split disjoint and the level City is called a splitting level. The
split disjoint has the same child level among all participating disjoint parent-child relationships.

Another kind of disjoint constraint exists in CIM: a join disjoint. A join disjoint has common parent among involved multiple parent-child relationships and this means a member of the parent level can drill down through only one of the parent-child relationships in the group. So, the member sets of the common parent level among parent-child relationships of the join disjoint group are disjoint. In Figure 4.3, parent-child relationships \( \text{State} \rightarrow \text{Country} \) and \( \text{Province} \rightarrow \text{Country} \) are join disjoint and this translates to the fact that a member of the level \( \text{Country} \) can drill down to either \( \text{State} \) or \( \text{Country} \) but not both. This join disjoint is represented as,

\[
(\text{State} \rightarrow \text{Country}) \otimes (\text{Province} \rightarrow \text{Country})
\]

### 4.4.1 The Language of Constraints

A constraint of disjointness imposed on a set of parent-child relationships \( \rightarrow = \{ \rightarrow_1, \rightarrow_2, \ldots, \rightarrow_n \} \) is denoted as \( \rightarrow_1 \otimes \rightarrow_2 \otimes \cdots \otimes \rightarrow_n \). It is a boolean expression and for the given rollup relations corresponding to the parent-child relationships they must satisfy the constraint to hold it true. The constraint asserts an exclusive participation of each member of the common level in at most one of the parent-child relationships in the set. The common level is shared as either a single parent or a single child level among all parent-child relationships. The symbol \( \otimes \) is called a disjoint symbol. A symbol \( \otimes \), called a big disjoint, is the condensed form of a disjoint applied on the set \( \rightarrow \), denoted as \( \otimes \rightarrow \) and is equivalent to \( \rightarrow_1 \otimes \rightarrow_2 \otimes \cdots \otimes \rightarrow_n \). \( \otimes \rightarrow \) represents a disjoint parent-child relationship set.

Let us consider a set of rollup relations \( RUP \) that are instantiations of a set of parent-child relationships \( \rightarrow \) such that \( \forall \rightarrow_i \in \rightarrow \exists RUP_i \in RUP : RUP_i \Rightarrow \rightarrow_i \). If the \( RUP \) satisfies the constraint of disjointness by maintaining exclusive participation of members of the common level then the boolean expression applied on the \( RUP \) returns TRUE otherwise FALSE. The TRUE and FALSE are true and false logical propositions respectively.

\( RUP \models \otimes \rightarrow \) shows \( RUP \) satisfies the constraint (i.e., \( \otimes \rightarrow \) evaluates to TRUE for the \( RUP \))

\( RUP \not\models \otimes \rightarrow \) shows \( RUP \) does not satisfy the constraint (i.e., \( \otimes \rightarrow \) evaluates
to FALSE for the \(RUP\)

The instantiations \(\forall \, \mathcal{P} \in \mathcal{X} \exists \text{RUP}_i \in \text{RUP} : \text{RUP}_i \Rightarrow \mathcal{P}_i\) is abbreviated as a mapping \(\text{RUP} \Rightarrow \mathcal{P}\). Mathematically speaking, the mapping of instantiations is a total\(^4\) injective function\(^5\) \(f_i : \mathcal{P} \rightarrow \text{RUP}\). Note that the \(\text{RUP}\) can contain a rollup relation that is not an instantiation of any parent-child relationship in the \(\mathcal{P}\) but not vice-versa. So if \(\text{RUP} \Rightarrow \mathcal{P}\) then any superset of \(\text{RUP}\), denoted as \(\text{superset}(\text{RUP})\), can also be mapped as instantiations of \(\mathcal{P}\) (i.e., \(\text{superset}(\text{RUP}) \Rightarrow \mathcal{P}\)). This definition comes handy for a definition of a Hierarchy instance in Definition 4.5.2.

Depending on whether a child or a parent level is shared in a \(\mathcal{P}\), the disjoint can be classified as:

1. **Definition 4.4.1** (Split disjoint). *If all the parent-child relationships in a set \(\mathcal{P}\) have the same child level i.e. \(\forall \, l^c_i \mathcal{P} \searrow l^p_i, l^c_j \mathcal{P} \searrow l^p_j \in \mathcal{P}: l^c_i = l^c_j\), then a disjoint constraint applied on \(\mathcal{P}\) is called a Split disjoint denoted by \(\bigotimes^S\). The members of the common child level can roll up exclusively through only one of the parent-child relationships in \(\mathcal{P}\). Mathematically,*

\[
\bigotimes^S \mathcal{P} \text{ where } \forall \, l^c_i \mathcal{P} \searrow l^p_i, l^c_j \mathcal{P} \searrow l^p_j \in \mathcal{P}: l^c_i = l^c_j \\
= \bigotimes_{i=1}^{\mathcal{P}} \bigotimes_{j=i}^{\mathcal{P}} \mathcal{P}_i \otimes \mathcal{P}_j \text{ where } \mathcal{P}_i \in \mathcal{P} \text{ for } 1 \leq i \leq n \\
\wedge \neg (\exists_{i \neq j} \mathcal{P}_i, \mathcal{P}_j \in \mathcal{P}) \exists m_1 \in \text{dom}(\mathcal{P}_i) \exists m_2 \in \text{dom}(\mathcal{P}_j) \exists m_1 = m_2)
\]

2. **Definition 4.4.2** (Join disjoint). *If all the parent-child relationships in a set \(\mathcal{P}\) have the same parent level i.e. \(\forall \, l^c_i \mathcal{P} \searrow l^p_i, l^c_j \mathcal{P} \searrow l^p_j \in \mathcal{P}: l^p_i = l^p_j\), then*

---

\(^4\)A function \(f : D \rightarrow R\) is total if \(\text{domain}(f) = D\).

\(^5\)A function \(f\) is injective if \(\forall x, y, z : f(x) = z \wedge f(y) = z\) then \(x = y\). Informally, an injective function maintains a distinct mapping.
a disjoint constraint applied on $\mathcal{P}$ is called a Join disjoint denoted by $\otimes^J$. The members of the common parent level can drill down exclusively through only one of the parent-child relationships in $\mathcal{P}$. Mathematically,

$$
\otimes^J \mathcal{P} \text{ where } \forall l^p_i, l^p_j, \mathcal{P}_i, \mathcal{P}_j \in \mathcal{P}: l^p_i = l^p_j
$$

$$
= \mathcal{P}_1 \otimes \mathcal{P}_2 \otimes \ldots \otimes \mathcal{P}_n \text{ where } \mathcal{P}_i \in \mathcal{P} \text{ for } 1 \leq i \leq n
$$

$$
\land \neg ( \exists \mathcal{P}_i, \mathcal{P}_j \in \mathcal{P}:
\exists m_1 \in \text{range}(\mathcal{P}_i)
\exists m_2 \in \text{range}(\mathcal{P}_j)
(i \neq j \land m_1 = m_2))
$$

By $\text{dom}(\mathcal{P}_k)$ and $\text{range}(\mathcal{P}_k)$ we mean the domain and range respectively of a rollup relation $RUP_k$ corresponding to a parent-child relationship $\mathcal{P}_k \in \mathcal{P}$ (i.e., $RUP_k \Rightarrow \mathcal{P}_k$). The negated boolean predicate in the definition of Split (or Join) disjoint checks that a member of the common level in the disjoint does not roll up (or drill down) through more than one participating parent-child relationship.

**Definition 4.4.3 (Hierarchy Constraint).** A hierarchy constraint $\phi$ is a boolean expression that specifies a disjointness among two or more parent-child relationships in a hierarchy schema. The constraint $\phi$ defined over a hierarchy schema $H = (L, \mathcal{P})$ is defined as $\phi = \mathcal{P}^\phi$ such that $\mathcal{P}^\phi \subseteq \mathcal{P}$. The $\phi$ is satisfied if $\otimes \mathcal{P}^\phi$ holds true. A hierarchy constraint can be either a Split disjoint ($\phi^S$) or a Join disjoint ($\phi^J$).

$$
\phi^S = \mathcal{P}^\phi \equiv \otimes^S \mathcal{P}^\phi
$$

$$
\phi^J = \mathcal{P}^\phi \equiv \otimes^J \mathcal{P}^\phi
$$

**Example 4.4.2** In *Geography* hierarchy (Figure 4.3), there are two hierarchy constraints: one Split disjoint ($\phi^S$) and one Join disjoint ($\phi^J$). They are defined as
follows:

\[ \phi^S = \{ \langle \text{City}, \text{State}, (0, 1), (1, *) \rangle, \langle \text{City}, \text{Province}, (0, 1), (1, *) \rangle \} \]
\[ \equiv \bigotimes \{ \langle \text{City} \nearrow \text{State} \rangle, \langle \text{City} \nearrow \text{Province} \rangle \} \]
\[ = (\text{City} \nearrow \text{State}) \otimes (\text{City} \nearrow \text{Province}) \]

\[ \phi^J = \{ \langle \text{State}, \text{Country}, (1, 1), (0, *) \rangle, \langle \text{Province}, \text{Country}, (1, 1), (0, *) \rangle \} \]
\[ \equiv \bigotimes \{ \langle \text{State} \nearrow \text{Country} \rangle, \langle \text{Province} \nearrow \text{Country} \rangle \} \]
\[ = (\text{State} \nearrow \text{Country}) \otimes (\text{Province} \nearrow \text{Country}) \]

**Example 4.4.3** In a real world scenario, a country can consist of either states or provinces or some other political divisions of their territories. This “some other” political division can be lumped together into a single category and represented by a direct rollup from cities to countries. The resulting hierarchy is shown in Figure 4.5(a) where a direct connection between a Split and a Join disjoint capture exactly this scenario. In this illustration, a country can drill down either to a group of states or to provinces or directly to cities (skipping categorization of these cities into either states or provinces first). This is represented by the Join disjoint. Also, those cities that roll up to states or provinces should not roll up directly to countries so as to prevent possible double counting of cities at the Country level. This is captured through the Split disjoint since the City \nearrow Country parent-child relationship is at a disjoint with City \nearrow State and City \nearrow Province. The Split and Join disjoint for this example is shown below:

\[ \phi^S = \{ \langle \text{City}, \text{State}, (0, 1), (1, *) \rangle, \langle \text{City}, \text{Country}, (0, 1), (0, *) \rangle, \langle \text{City}, \text{Province}, (0, 1), (1, *) \rangle \} \]
\[ \equiv \bigotimes \{ \langle \text{City} \nearrow \text{State} \rangle, \langle \text{City} \nearrow \text{Country} \rangle, \langle \text{City} \nearrow \text{Province} \rangle \} \]
\[ = (\text{City} \nearrow \text{State}) \otimes (\text{City} \nearrow \text{Country}) \otimes (\text{City} \nearrow \text{Province}) \]

41
Figure 4.5: Different combinations of Disjoint constraints

\[ \phi^I = \{ \langle \text{State}, \text{Country}, (1, 1), (0, *) \rangle, \]
\[ \langle \text{City}, \text{Country}, (0, 1), (0, *) \rangle, \]
\[ \langle \text{Province}, \text{Country}, (0, 1), (1, *) \rangle \} \]
\[ \equiv \Box\{ (\text{State} \not\leftrightarrow \text{Country}), (\text{City} \not\leftrightarrow \text{Country}), (\text{Province} \not\leftrightarrow \text{Country}) \} \]
\[ = (\text{State} \not\leftrightarrow \text{Country}) \otimes (\text{City} \not\leftrightarrow \text{Country}) \otimes (\text{Province} \not\leftrightarrow \text{Country}) \]

Example 4.4.4 To make the matter slightly more complicated, there are countries that consist of Provinces or States (e.g., the US) and in addition consist of cities (Washington D.C. in the US) that does not belong to any state/province. So these cities do not belong to any particular state/province but just the country. As a
result, the country consists of states/provinces plus one or more cities. Now, a Join disjoint from \( \text{Country} \) into \( \text{State}, \text{Province} \) and \( \text{City} \) can not be used as this enforces exclusive drill down to one and only one of those child levels. But the Split disjoint from \( \text{City} \) to \( \text{State}, \text{Province} \) and \( \text{Country} \) still holds. This updated hierarchy is shown in Figure 4.5(b). Now countries like the US can consist of states/provinces and cities. The Split and Join disjoint for this case is as follows:

\[
\phi^S = \{ \langle \text{City}, \text{State}, (0, 1), (1, \ast) \rangle, \\
\langle \text{City}, \text{Country}, (0, 1), (0, \ast) \rangle, \\
\langle \text{City}, \text{Province}, (0, 1), (1, \ast) \rangle \} \\
\equiv \boxtimes \{ (\text{City} \nearrow \text{State}), (\text{City} \nearrow \text{Country}), (\text{City} \nearrow \text{Province}) \} \\
= (\text{City} \nearrow \text{State}) \otimes (\text{City} \nearrow \text{Country}) \otimes (\text{City} \nearrow \text{Province})
\]

\[
\phi^J = \{ \langle \text{State}, \text{Country}, (1, 1), (0, \ast) \rangle, \langle \text{Province}, \text{Country}, (1, 1), (0, \ast) \rangle \} \\
\equiv \boxtimes \{ (\text{State} \nearrow \text{Country}), (\text{Province} \nearrow \text{Country}) \} \\
= (\text{State} \nearrow \text{Country}) \otimes (\text{Province} \nearrow \text{Country})
\]

### 4.5 Hierarchy and Hierarchy Instance

Now we define Hierarchy using the definitions of Hierarchy schema (Definition 4.3.1) and Hierarchy constraint (Definition 4.4.3). A Hierarchy instance is defined on the basis of Hierarchy (Definition 4.5.1).

**Definition 4.5.1 (Hierarchy).** A hierarchy is a combination of a hierarchy schema \( H = (L, \nearrow) \) and a set of hierarchy constraints \( \Phi \) defined over \( H \). Thus a hierarchy is defined as \( (H, \Phi) \).

A hierarchy named \( X \) is represented as \( \text{Hierarchy}_X \).

**Definition 4.5.2 (Hierarchy Instance).** A Hierarchy instance \( h \), corresponding to a hierarchy \( (H, \Phi) \), is a graph homomorphism \( f : (M, \text{RUP}) \rightarrow (L, \nearrow) \) where \( M = \bigcup_{l \in L} \text{dom}(l) \) and \( \text{RUP} \) is a set of rollup relations such that (a) \( \forall M_l \in M : f(M_l) = l \) where \( l \in L \) (b) \( \forall \text{RUP}_l^i \in \text{RUP} : f(\text{RUP}_l^i) = l_i \nearrow l_j \) where \( l_i, l_j \in L \) and \( l_i \nearrow l_j \in \nearrow \),
and (c) \( h \) satisfies \( \phi \), denoted as \( h \vDash \phi \), for all \( \phi \in \Phi \) in the hierarchy.

\[
h \vDash \phi \\
\equiv \text{RUP} \vDash \phi \text{ where } \phi \Rightarrow \text{RUP} \Rightarrow \phi
\]

Condition (a) implies that for each set of members in the hierarchy instance, there exists a corresponding level in \( L \) and (b) says that if a member of level \( l_i \) rolls up to level \( l_j \) then there must exist a parent-child relationship from \( l_i \) to \( l_j \). The last condition checks whether a hierarchy instance satisfies all the constraints laid out in the hierarchy so as to become a valid instance. This is equivalent to saying that the \text{RUP} corresponding to \( h \) satisfies the constraint. Only an instance that does not violate any constraints in \( \Phi \) qualifies as a valid instance.

A hierarchy instance defined over hierarchy \( X \) is represented as \( h_X \).

**Example 4.5.1** A hierarchy instance \( h_{\text{Calendar}} \) (in Figure 4.2) from Example 4.2.3 is a graph homomorphism of a DAG (Figure 4.1(b)) related to hierarchy Calendar in Example 4.2.1 since there are one-to-one correspondences between the entities in the two figures. Each set of members in the instance figure can be mapped to corresponding level in the DAG and similarly for the sets of rollup relation in the instance and parent-child relationships in the DAG. The hierarchy schema corresponding to the DAG does not possess any hierarchy constraint (i.e., \( \Phi = \emptyset \)); thus, the instance does not violate any constraint.

**Example 4.5.2** A more complicated example of a hierarchy instance is cited in Figure 4.6 as \( h_{\text{Geography}} \) that corresponds to a Geography hierarchy (HierarchyGeography) in Figure 4.3. The DAG related to the hierarchy schema is depicted in Figure 4.4 which captures parent-child relationships between the levels but excludes the hierarchy constraints. The hierarchy HierarchyGeography consists of the hierarchy schema \( H_{\text{Geography}} = (L, \Rightarrow) \) shown in Example 4.3.2 and the hierarchy constraints \( \Phi \) as follows:

\[
\Phi = \left\{ \{(\text{City}, \text{State}, (0, 1), (1, *))\}, \{(\text{City}, \text{Province}, (0, 1), (1, *))\}, \{(\text{State}, \text{Country}, (1, 1), (0, *))\}, \{(\text{Province}, \text{Country}, (1, 1), (0, *))\} \right\}
\]
A hierarchy instance corresponding to a hierarchy $Hierarchy_{Geography}$ is depicted in Figure 4.6 and it consists of following member sets $M_{level} \in M$ where $level \in L$ and Rollup relations $RUP_{level_{parent} \rightarrow level_{child}} \in RUP$ where $level_{parent}, level_{child} \in L$ and $level_{child} \rightarrow level_{parent}$:

$$M_{Branch} = \{Branch1, Branch2, Branch3, Branch5, Branch6\}$$

$$M_{City} = \{Vancouver, Richmond, Austin\}$$

$$M_{Province} = \{British\ Columbia\}$$

$$M_{State} = \{Texas\}$$

$$M_{SalesRegion} = \{Region1\}$$

$$M_{Country} = \{Canada, US\}$$

**Figure 4.6:** An instance of Geography hierarchy (Figure 4.3)
The graph homomorphism $f$ between $(L, \rightarrow)$ and $(M,RUP)$ is as follows:

$$f : L \rightarrow M = \{(Branch,M_{Branch}),
\ (City,M_{City}),
\ (Province,M_{Province}),
\ (State,M_{State}),
\ (SalesRegion,M_{SalesRegion}),
\ (Country,M_{Country})\}$$
\[f : \nearrow \rightarrow RUP = \{(\text{Branch} \nearrow \text{City}, RUP^{\text{City}}_{\text{Branch}}), \]
\[(\text{City} \nearrow \text{Province}, RUP^{\text{Province}}_{\text{City}}), \]
\[(\text{City} \nearrow \text{State}, RUP^{\text{State}}_{\text{City}}), \]
\[(\text{City} \nearrow \text{SalesRegion}, RUP^{\text{SalesRegion}}_{\text{City}}), \]
\[(\text{Province} \nearrow \text{Country}, RUP^{\text{Country}}_{\text{Province}}), \]
\[(\text{State} \nearrow \text{Country}, RUP^{\text{Country}}_{\text{State}}), \]
\[(\text{SalesRegion} \nearrow \text{Country}, RUP^{\text{Country}}_{\text{SalesRegion}})\}\]

The member sets and rollup relations shown above have corresponding levels and parent-child relationships respectively in the \text{HierarchyGeography}. Also, each city either rolls up to a State or a Province but not both. Similarly, each Country drills down to either a set of Provinces or a set of States but not both. This satisfies both hierarchy constraints in the \text{HierarchyGeography}. Since parent-child relationships \text{City} \nearrow \text{SalesRegion} and \text{SalesRegion} \nearrow \text{Country} do not participate in any hierarchy constraint, the rollup relations related to them do not violate any constraint.

\subsection*{4.6 Aggregation}

The primary motivation for defining a hierarchy is to establish a means of analysis of data at varying granularity through an aggregation. These data, called measures, are stored in a fact relationship that references the bottom level of the hierarchy. As a result, only the members of the bottom level have direct associations to the data in the fact relationship. All other non-bottom levels of the hierarchy are directly or indirectly related to the bottom level through parent-child relationships forming the hierarchy. This hierarchy structure is leveraged to compute aggregated measures for non-bottom levels. A hierarchy structure provides a mapping between members of the bottom level and its ancestor level.

In a non-strict hierarchy, there is a possibility of double counting during aggregation as elicited in Example \ref{ex:double-counting}. To the best of our knowledge, there is no general consensus on how to handle this double counting. Bertossi et al. in \cite{9} have tried to tackle this problem by giving approximate answers to those mem-
bers that account for double counting by “repairing” a non-strict hierarchy during aggregation. Other researchers [16, 18, 20] require hierarchies to be strict for aggregation to be applicable on them. An inconsistency in an aggregation invited by a non-strict hierarchy defeats the purpose of describing a hierarchy for aggregation. Not only does a non-strict hierarchy introduce a possibility of double counting but it also raises issues leveraging pre-computed aggregates of lower levels to calculate aggregation at higher levels. For this reason, in the current implementation of CIM, we only allow aggregation of strict hierarchies. All the reasoning on aggregation of hierarchies assume the hierarchies to be strict. Malinowski and Zimányi [22] discusses the possible methods to convert a non-strict hierarchy to a strict one, e.g., by adding a distribution factor, but none of the methods are generic for all circumstances.

All the discussion here onward till the end of this section assumes a hierarchy to be a strict hierarchy.

The bottom level has the finest granularity and one bottom level member can relate to only one member from a non-bottom level, so a mapping forms a mathematical function from the bottom level to the non-bottom level. This mapping combined with a summarization function gives an aggregation of data from the fact relationship. The summarization function can be SUM, COUNT, AVERAGE, MAX, MIN, etc., which are self-explanatory. The mapping between bottom and non-bottom level members is derived by traversing the hierarchy from the bottom level to the desired non-bottom level.

**Example 4.6.1** An example of this is illustrated in Figure 4.7. The mapping in this figure is derived by traversing the hierarchy in Figure 4.6.

This process of traversing a hierarchy from the bottom level to its ancestor level in order to calculate the summarized values of measures from the fact relationship is called an aggregation. The summarized values of measures from the fact relationship at the granularity of a desired level is referred to as aggregates for the level. The aggregation uses mappings between members of the desired level to those of the bottom level to calculate aggregates. Such a mapping gives us a group of bottom level members mapped to each member of the desired ancestor level. The two groups of bottom level members mapped to any two members of the desired level
Figure 4.7: A mapping derived from Figure 4.6 between levels Branch and Country

must be disjoint to prevent double-counting leading to an incorrect aggregation.

In order to compute a mapping between the bottom level and the desired level, one or more chains of parent-child relationships originating from the bottom level and terminating at the desired level need to be traversed. Each of these chains of parent-child relationships is known as an aggregation path.

**Definition 4.6.1** (Aggregation path). An aggregation path \( AGGPATH = L_{AGG} \) in a hierarchy schema \( H = (L, \nearrow) \) is an ordered set of levels such that (a) \( L_{AGG} \subseteq L \), and (b) every adjacent pair of levels \( l_i, l_j \in L_{AGG} \) forms a parent-child relationship \( l_i \nearrow l_j \) in \( H \).

If an aggregation path \( AGGPATH = \{l_1, l_2, \ldots, l_n\} \) exists in a hierarchy schema \( H \), then \( l_i \nearrow l_{i+1} \) for \( 1 \leq i \leq n - 1 \) in \( H \). This aggregation path may also be represented as \( l_1 \nearrow l_2 \nearrow \ldots \nearrow l_n \). An aggregation path must have at least one parent-child relationship; hence, at least two levels must exist in the path. It is easy to
note that every segment of an aggregation path is also an aggregation path. An aggregation path does not necessarily start at the bottom level of a hierarchy but a longest aggregation path in a hierarchy must have the bottom level as its first level.

**Example 4.6.2** In Figure 4.3, Branch-City-SalesRegion-Country, Branch-City-State-Country, Branch-City-Province-Country and their segments form aggregation paths.

**Example 4.6.3** A level in a hierarchy may be reached from the bottom level through one or more aggregation paths. For instance, in Figure 4.3, the level Province can be reached through only one aggregation path Branch ↗ City ↗ Province but level Country can be reached through three paths: Branch ↗ City ↗ State ↗ Country, Branch ↗ City ↗ Province ↗ Country and Branch ↗ City ↗ SalesRegion ↗ Country.

A group of aggregation paths reaching the desired level in a hierarchy forms a subhierarchy of the hierarchy. This subhierarchy is an induced subgraph\(^6\) of the desired level and its descendants in the hierarchy. The subhierarchy also includes the hierarchy constraints subsumed by its constituent levels. This subhierarchy is called an aggregation traversal for the desired level.

**Definition 4.6.2 (Aggregation Traversal).** An aggregation traversal \(\text{AGGT}_{l} = (L_l, \rightarrow_l, \Phi_l)\) of a level \(l\) defined over a hierarchy \((H, \Phi)\) is an induced subgraph of \(H = (L, \rightarrow)\) where \(L_l \subseteq L\) and \(l \in L_l\) such that (a) \(\forall l_i \in L : l_i \rightarrow l \in \leftrightarrow l \in L_l\), (b) \(\forall l_i, l_j \in L : l_i \rightarrow l_j \in \leftrightarrow l_i \rightarrow l_j \in L_l\), (c) \(\Phi_l \subseteq \Phi\), and (d) \(\forall \phi \in \Phi : \phi \subseteq L_l \iff \phi \in \Phi_l\).

The bottom level forms a source node of the graph corresponding to an aggregation traversal and the desired level forms the sink node. The sink node is called the root node of the aggregation traversal. Similarly, the bottom level forms the leaf node of the traversal. The structure of the aggregation traversal determines the complexity of computing an aggregation. An aggregation traversal may be decomposed into multiple aggregation paths that partially overlap. This partial overlap occurs at least at the leaf and the root nodes.

\(^6\)In graph theory, a graph \(H = (V_H, E_H)\) is an induced subgraph of a graph \(G = (V_G, E_G)\) if the set of vertices, \(V_H\), in \(H\) is a subset of vertices, \(V_G\), in \(G\); and for the set of all edges in \(G, E_G\), between pairs of nodes from \(V_H\), those edges must also exist in \(E_H\) and only those edges.
**Figure 4.8**: Different types of Aggregation Traversals

**Example 4.6.4** In Figure 4.3, the aggregation traversal for level *Country* is comprised of aggregation paths: (1) *Branch* ↗ *City* ↗ *State* ↗ *Country*, (2) *Branch* ↗ *City* ↗ *Province* ↗ *Country*, and (3) *Branch* ↗ *City* ↗ *SalesRegion* ↗ *Country*. All three constituent aggregation paths have *Branch*, *City* and *Country* levels in common. In addition, the traversal has two hierarchy constraints: (1) (*City* ↗ *State*) ⊗ (*City* ↗ *Province*), and (2) (*State* ↗ *Country*) ⊗ (*Province* ↗ *Country*).

Depending on the structure of the aggregation paths, they can be categorized into three groups: **Simple**, **Disjoint** and **Complex**.

- **Simple aggregation traversal**: A Simple aggregation traversal consists of only one aggregation path rendering the aggregation trivial. It also rules out the possibility of double counting the bottom level members during aggregation in a strict hierarchy. Obviously, no hierarchy constraint exists for Simple aggregation traversal.

**Example 4.6.5** An aggregation traversal for level *Province* in Figure 4.3 consists of only one aggregation path: *Branch* ↗ *City* ↗ *Province* as shown in Fig-
• **Disjoint aggregation traversal:** A Disjoint aggregation traversal consists of multiple aggregation paths bound together by one or more pairs — each pair composed of one Split and one Join — of hierarchy constraints. The hierarchy constraints rule out the possibility of double counting the bottom level members during aggregation by ensuring disjointness between the aggregation paths. Any two constituent aggregation paths are disjoint with each other. Each bottom level member can reach the root node of the traversal through only one aggregation path. Each member of the root node drills down through a single aggregation path to the bottom level. So a disjoint union of the mappings created through individual constituent aggregation paths gives the total mapping between the root level and the bottom level of the aggregation traversal. In a strict hierarchy, no deduplication is necessary because double-counting can not occur due to disjointness of the aggregation paths.

**Example 4.6.6** If we remove the level `SalesRegion` from Figure 4.3 and the parent-child relationships attached to that level for the sake of an example then the aggregation traversal for level `Country` reduces to the subhierarchies shown in Figure 4.8(b). In Figure 4.8(b), level `Country` is composed of two aggregation paths that are disjoint: (1) `Branch ↗ City ↗ State ↗ Country`, and (2) `Branch ↗ City ↗ Province ↗ Country`. They provide exclusive paths for the bottom level members so a disjoint union suffices to compute an aggregation for the level `Country`.

• **Complex aggregation traversal:** A Complex aggregation traversal consists of multiple aggregation paths such that all of them are not bound together by hierarchy constraints. It may consist of split and join disjoints but there are additional aggregation paths that are not necessarily part of the hierarchy constraints. So any two aggregation paths within the complex aggregation traversal are not necessarily disjoint with each other. This introduces the possibility of double-counting of the bottom level members when following two aggregation paths and thus we need to be careful during aggregation. If a bottom level member maps to the root node member through more than one aggregation path then we
need to reconcile these redundant mappings through deduplication. A unique
 drill-down path for a root node member no longer holds true and multiple paths
 may need to be considered for drill-down.

**Example 4.6.7** In Figure 4.8(c), the *SalesRegion* level is linked to the levels
*City* and *Country* and it is not bound by split and join constraints, so one or more
cities that map to *Country* through *State/Province* can roll up to level *Country*
through level *SalesRegion*, as well. So a simple disjoint union is not sufficient
between multiple aggregation paths and a deduplication operation is necessary
to remove redundant mappings between the levels *Branch* and *Country*.

The rollup relations corresponding to parent-child relationships can be combined
through composition to create a mapping between any two levels connected
through an aggregation path. The result of this composition is called a rollup path.
Rollup paths associated with the constituent aggregation paths in an aggregation
traversal can be merged through a union to get a rollup mapping [16] between two
levels. Only a complex aggregation path needs deduplication during a union.

**Definition 4.6.3 (Rollup path).** A rollup path $\text{RPATH}$ in a hierarchy instance $H$
is a mathematical binary relation between members of two levels formed by a
composition of rollup relations between those levels such that there exists a rollup
relation between every adjacent pair of levels. Mathematically, for a pair of levels
$l_s, l_e$ such that $l_s \uparrow^\ast l_e$, the rollup path between their members is defined as

$$\text{RPATH}_{le} = \text{RUP}_{li} \circ \text{RUP}_{l1} \circ \ldots \circ \text{RUP}_{lk-1} \circ \text{RUP}_{lk}.$$ 

Each member of a level should follow a rollup path to reach its ancestor at a higher level. A rollup path between
any two levels $l_s$ and $l_e$, defined as $\text{RPATH}_{le} = \text{RUP}_{li} \circ \text{RUP}_{l1} \circ \ldots \circ \text{RUP}_{lk-1} \circ \text{RUP}_{lk}$,
exists if there exists an ordered set of levels $l_s, l_1, \ldots, l_k, l_e$ such that $l_s \uparrow^\ast 1, l_i \uparrow^\ast 1_j$
where $1 \leq i < j \leq k$ and $l_k \uparrow^\ast l_e$. $\text{RPATH}_{le} : \text{dom}(l_s) \to \text{dom}(l_e)$.

Take every pair of levels $l_s, l_e$ in a strict hierarchy with $l_s \uparrow^\ast l_e$ through more
than one rollup path, then for each pair of rollup paths $\text{RPATH}_i, \text{RPATH}_j$ between
$l_s$ and $l_e$, the condition $\text{RPATH}_i(x) = \text{RPATH}_j(x)$ holds.

A rollup path accounts for a mapping generated through one aggregation path.
An aggregation traversal may consist of multiple aggregation paths. That leads to
multiple rollup paths for an aggregation traversal between two levels such that one
rollup path accounts for each individual aggregation path. The merging of multiple rollup paths gives a complete mapping between the two levels. This complete mapping between the two levels is called a rollup mapping.

**Definition 4.6.4** (Rollup mapping). A rollup mapping \( \Gamma \) from a level \( l_i \) to \( l_j \), denoted by \( \Gamma_{l_i}^{l_j} \), in a hierarchy \((H, \Phi)\) with \( H = (L, \rightarrow) \) is a union of a set of RPATH : \( \text{dom}(l_i) \rightarrow \text{dom}(l_j) \).

**Example 4.6.8** The example of a rollup mapping between levels Branch and Country, \( \Gamma_{\text{Country}}^{\text{Branch}} \) from Figure 4.6 is shown in Figure 4.7. As illustrated by the example, a rollup mapping gives a complete mapping between the members of two levels.

In the literature, there are proponents of hierarchies supporting multiple bottom levels [18]. We argue that despite forcing one bottom level in the definition of a hierarchy in CIM, we subsume the expressibility of the hierarchy model proposed by the authors in [18]. Despite the presence of multiple bottom levels in their model, these levels must be mutually disjoint among themselves creating non-overlapping partitions of their members and all the levels must have the same granularity. Consequently, each bottom level can have a distinct aggregation path of its own. Whereas in CIM, the hierarchy allows expressing generalization/specialization parent-child relationship using a disjoint symbol so even though only one bottom level participates in a fact relationship, a disjoint specialization through splitting parent-child relationships can create explicit partitions of bottom level members and, in turn, separate aggregation paths for each specialized parent.

**Example 4.6.9** In Figure 4.9(a) multiple bottom levels Men’s Wear, Women’s Wear and Kids’ Wear are shown. The equivalent hierarchy with single bottom level, as only permitted in CIM, is shown in Figure 4.9(b) where a new bottom level Wear is created and then a disjoint symbol partitions the bottom level into three levels.
4.7 Dimension

A dimension schema, or simply dimension\textsuperscript{7} consists of either a level or a collection of hierarchies. If a dimension has only one level then no hierarchy needs to be defined; otherwise, a hierarchy is mandatory. The presence of only one level in a dimension indicates that no aggregation is intended in the dimension. Like in [21], CIM supports multiple hierarchies in a dimension to depict multiple ways in which the bottom level can be aggregated to the same or different higher levels. The hierarchies belonging to a dimension can share levels and parent-child relationships between them, thus increasing intuitiveness, cleanliness and clarity of representation. They also prevent redundant mappings which would otherwise occur for repeated levels with the same semantics. This sharing is extended to the hierarchy instance as well. That means it is not just the schema constructs that are shared but also the levels domains and the rollup relations corresponding to the common levels and parent-child relationships. The multiple hierarchies subsumed by a dimension must share the bottom level.

\textsuperscript{7}Note that hierarchy schema and hierarchy are not the same.
Intuitively, in a multidimensional schema a dimension provides an axis along which a measure can be analyzed and the axis should contain data of only one type (with one granularity). Different hierarchies provide a variety of ways to analyze this most granular data in the bottom level with a desirable degree of summarization. So, a dimension consists of one bottom level that is shared among the hierarchies as their bottom level. A dimension is represented by a shaded rectangle symbol and attached with the hierarchy symbols (if any). A dimension participates in a fact relationship but at the logical level it is the bottom level of the dimension that is actually involved in the fact relationship.

**Example 4.7.1** In Figure 4.10 two cases are depicted, one with just one level in a dimension (Figure 4.10(a)) and other with a hierarchy (Figure 4.10(b)). Figure 4.10(a) shows the Competitor dimension which consists of only one level: Company. An aggregation is meaningless with just one level, so no hierarchy is mentioned for this case. In Figure 4.10(b), a lattice of levels is associated with a dimension Time, so a hierarchy Fiscal is created with Day as its bottom level. Note that the hierarchy symbol (a rounded corner rectangle) appears twice: one attached to the dimension to elicit all the hierarchies associated with the dimension, and another attached to the bottom level of the hierarchies to uniquely identify each lattice by tagging it. The hierarchy symbol for Fiscal appears once attached to the dimension Time and again with the bottom level to tag the lattice Fiscal along which the bottom level Day can be analyzed.

**Example 4.7.2** In Figure 4.11 there are two strict hierarchies associated with dimension Staff: Works and Lives. The hierarchy Lives consists of the aggregation path Employee-City-State, and hierarchy Works consists of the aggregation path Employee-Sales District-State. An employee can work in one state and live in another so the employee is associated with different states depending on which aggregation path we follow. Since, the definition of a strict hierarchy prevents a member from level Employee to roll up to more than a single member of level State, in order to accommodate the possibility of different rollups we have to use two hierarchies that share levels Employee and State.

So, if a member of a level can roll up to different members of a higher level through different aggregation paths, then it is advisable to make these aggregation
Figure 4.10: Single level and single hierarchy dimensions

paths belong to different hierarchies to enable aggregation.

**Definition 4.7.1** (Dimension). A dimension $D$ is defined as a tuple $(l, H)$ where $l$ is the bottom level of the dimension and $H$ is a set of hierarchies such that $\forall \text{Hier} \in H : l_{\text{Hier}_\text{BOTTOM}} = l$. In words, the bottom level $l$ is shared among the constituent hierarchies $H$ of the dimension $D$ as their common bottom level.

**Definition 4.7.2** (Dimension Instance). A dimension instance $d$ over dimension $D = (l, H)$ is defined as a tuple $h$ where $h$ is a set of hierarchy instances such that $\forall h \in h$ there exists $H \in H$ such that $h$ is a hierarchy instance of $H$. 

57
**Example 4.7.3** In Figure 4.11, the level **Employee** is shared between two strict hierarchies **Lives** and **Works** as their bottom levels in dimension **Staff**. So, the dimension **Staff** is defined as \( Staff = (Employee, \{Lives, Works\}) \). Figure 4.12 shows a dimension instance defined over a dimension from Figure 4.11. The instance consists of two hierarchy instances: one for **Lives** and the other for **Works**. The two instances share level domains for **Employee** and **State**. Since the two aggregation paths belong to different strict hierarchies, a member of level **Employee** called **Jack** can roll up to two different states, **Washington** and **Oregon**, by following two separate hierarchies, **Lives** and **Works**, respectively.

Very often, hierarchies from different dimensions can have the same levels or aggregation paths. In such situations, it makes sense to share those common structures for the same reason that we share levels or aggregation paths among
hierarchies within a dimension. To this end, we allow hierarchy overlap between multiple dimensions. An event where two or more hierarchies from the same or different dimensions share an aggregation path (simple or disjoint) resulting in an overlap is called a Hierarchy overlap. As mentioned above, the hierarchy overlap extends to an overlap in the hierarchy instance as well; meaning, the level domains and the rollup relations corresponding to the common levels and parent-child levels are also shared. The Parallel dependent hierarchy of [21] represents such an overlap.

**Example 4.7.4** Figure 4.13 shows a simple example where hierarchies belonging to two dimensions *Time* and *PayrollTime* share an aggregation path between them. The hierarchies *Fiscal* and *Monthly* both have *Month* and *Year* levels in common including the same parent-child relationship *Year-Month* (based on the Gregorian calendar) between them. This overlap allows us to combine the common structure. The hierarchy name is used to disambiguate which parts of a combined hierarchy belongs to the involved dimensions. The *Fiscal* hierarchy contains *Day-Month-Year-Decade*, and the *Monthly* hierarchy includes just *Month-Year* which

**Figure 4.12:** A dimension instance of the dimension from Figure 4.11
results in an overlap. This aggregation path is shared between the two hierarchies. An instance of this hierarchy overlap is shown in Figure 4.14 where it is clearly depicted as overlap between the level domains for Month and Year and the rollup relation $R_{\text{Month}}^{\text{Year}}$ between the Fiscal and Monthly hierarchies.

**Example 4.7.5** A slightly more complicated case is shown in Figure 4.15 where two dimensions Warehouse and Client share a subset of a disjoint aggregation path between them. The assumption behind the example is that these dimensions belong to a company that has storage facilities in Canada but does selling business in both the US and Canada. So the level City can only roll up to Province for the StorageDistribution hierarchy whereas the City can roll up to State, Province and
Figure 4.14: A simple instance of the hierarchy overlap from Figure 4.13

Country for CustomerDistribution hierarchy. Here, the hierarchy information is included in the parent-child relationship path to show those restrictions in a rollup path based on which hierarchy we are following. An instance of this hierarchy overlap is illustrated in Figure 4.16 where the hierarchy instance StorageDistribution in dimension Warehouse is a subset of the hierarchy instance CustomerDistribution in dimension Client as shown by the dot-dashed freeform loop circumscribing the hierarchy instance StorageDistribution.
Figure 4.15: An example of two dimensions with hierarchy overlap
Figure 4.16: A CustomerDistribution hierarchy instance of dimension Client with StorageDistribution hierarchy instance of dimension Warehouse sharing a subtree shown by a freeform loop.
Chapter 5

Formal Semantics

This chapter defines the formal semantics of the Conceptual Model in the CIM framework. CIM builds upon the conceptual model defined in the MultiDim model [22]. Since we are dealing particularly with the artifacts used for representing sets of data and relationships between them, we use denotational semantics to describe our model. Moreover, our method is inspired from the formalization in [22]. We achieve our goal by first translating our graphical conceptual model into an equivalent textual representation. The semantics of this “serialized” textual representation provides the semantics for our graphical model.

Many of the preliminary concepts and formalisms used in this chapter are taken from the same source [15] that MultiDim used in creating their formalism. In addition, the textual representation is modeled directly from MultiDim [22]. We use this as a basis for formalization to be in line with the existing practice for the conceptual modeling in data warehouse field.

We start out by describing the notation and definitions of the predefined data types supported by CIM, followed by the description of metavariables involved in abstract syntax of the conceptual model.

5.1 Notation

The following notation is used throughout the formalization:

- \( \text{SET} \) — the class of sets.
- \( FSET \) — the class of finite sets.
- \( TF \) — the class of total functions.
- \( REL \) — the class of relations.\(^1\)

Let us assume \( S_1, S_2, S_3, \ldots, S_n \in SET \), \( S_i \cup S_j \) denotes a disjoint union of sets, \( S_i \cup S_j \) indicates a union of sets and \( S_1 \times S_2 \times \ldots \times S_n \) represents the Cartesian product over the sets \( S_1, S_2, S_3, \ldots, S_n \). The disjoint union preserves duplicate elements from the participating sets by giving them unique identities.

Further, we let \( S^* \) denote the set of finite lists over \( S \) and \( S^+ \) represent the set of non-empty finite lists over \( S \), where \( S \) is a set.

We write finite sets as \( \{c_1, c_2, \ldots, c_n\} \), lists as \( \langle c_1, c_2, \ldots, c_n \rangle \) and elements of Cartesian products as \( (c_1, c_2, \ldots, c_n) \). For any set, we use \( \bot \) to denote an undefined value of the set.

### 5.2 Data Signature

In CIM, we are building a prototype model, so we are limiting the expressibility of the conceptual language to the fundamental data types, operations and predicates. These basics include integer, floating point and string data types, simple arithmetic operators and inequalities. A data signature describes these predefined data types, operations and predicates. We assume the data types \( \text{int}, \text{real} \) and \( \text{string} \); the inclusion of additional data types is straightforward. These data types and the operations and predicates on them have the usual semantics.

Each operator takes a list of input elements and produces an element as output. There are two operations involved during an execution of an operator, namely, provision of inputs and outputs. These operations can be defined as mathematical functions: \( \text{input} \) for an input function and \( \text{output} \) for an output function. Similarly, a predicate is a conditional operation that accepts a list of arguments (parameters) for which it holds true. A predicate is defined as a relation \( \text{arg} \). The \( \text{input} \) and \( \text{output} \) functions and the \( \text{arg} \) relation are collectively called a data signature. Let us assume that \( DATA \) denotes a set of data types, \( OPNS \) a set of operators and \( PRED \)

\(^1\)A relation is a set of tuples.
represents a set of predicates.

The syntax of the data signature is given below.

- the sets \(\text{DATA}, \text{OPNS}, \text{PRED} \in \text{FSET}\),
- a function \(\text{input} \in \text{TF}\) such that \(\text{input} : \text{OPNS} \to \text{DATA}^*\),
- a function \(\text{output} \in \text{TF}\) such that \(\text{output} : \text{OPNS} \to \text{DATA}\), and
- a function \(\text{args} \in \text{TF}\) such that \(\text{args} : \text{PRED} \to \text{DATA}^+\).

If \(\sigma \in \text{OPNS}\), \(\text{input}(\sigma) = \langle d_1, \ldots, d_n \rangle\), and \(\text{output}(\sigma) = d\), this is denoted as \(\sigma : \langle d_1, \ldots, d_n \rangle \to d\). If \(\pi \in \text{PRED}\), with \(\text{args}(\pi) = \langle d_1, \ldots, d_n \rangle\), this is denoted by \(\pi : \langle d_1, \ldots, d_n \rangle\).

The following are the predefined data types and some operators and predicates applied on those data types.

\[
\begin{align*}
\text{DATA} & \supseteq \{\text{int}, \text{real}, \text{string}\} \\
\text{OPNS} & \supseteq \{+i, -i, \times_i : \text{int} \times \text{int} \to \text{int}\} \\
 & \quad \{+r, -r, \times_r : \text{real} \times \text{real} \to \text{real}\} \\
 & \quad \{/i : \text{int} \times \text{int} \to \text{real}\} \\
 & \quad \{/r : \text{real} \times \text{real} \to \text{real}\} \\
 & \quad \{\text{cat} : \text{string} \times \text{string} \to \text{string}\} \\
 & \quad \ldots \}
\end{align*}
\]

\[
\begin{align*}
PRED & \supseteq \{<i,i>, \leq_i, \geq_i, \neq_i : \text{int} \times \text{int}\} \\
 & \quad \{<r,r>, \leq_r, \geq_r, \neq_r : \text{real} \times \text{real}\} \\
 & \quad \{<s,s>, \leq_s, \geq_s, \neq_s : \text{string} \times \text{string}\} \\
 & \quad \{\text{isNull}_i : \text{int}\} \\
 & \quad \{\text{isNull}_r : \text{real}\} \\
 & \quad \{\text{isNull}_s : \text{string}\} \\
 & \quad \ldots \}
\end{align*}
\]
5.3 Metavariabes

In this section we list a set of variables that will be used for semantics definitions.

$S_D \in \text{SchemaDecl}$ – CVL schema declarations
$D_D \in \text{DimDecl}$ – dimension declarations
$L_D \in \text{LevDecl}$ – level declarations
$PC_D \in \text{PCRelDecl}$ – parent-child relationship declarations
$F_D \in \text{FactRelDecl}$ – fact relationship declarations
$IC_D \in \text{ICDecl}$ – integrity constraint declarations
$H_D \in \text{HierDecl}$ – hierarchy declarations
$A_D \in \text{AttDecl}$ – attribute declarations
$M_D \in \text{MeasDecl}$ – measure declarations
$I_S \in \text{InvSpec}$ – the set of dimension involvement specifications
$D \in \text{Dimensions}$ – the set of dimension names
$role \in \text{Roles}$ – the set of role names
$F \in \text{FactRels}$ – the set of fact relationship names
$L \in \text{Levels}$ – the set of level names
$PC \in \text{PCrels}$ – the set of parent-child relationship names
$Q \in 2^{PCrels}$ – the set of subsets of parent-child relationship names
$H \in \text{Hier}$ – the set of hierarchy names
$E \in 2^{Hier}$ – the set of subsets of hierarchy names
$A \in \text{Attributes}$ – the set of attribute names
$K \in 2^{Attributes}$ – the set of subsets of attribute names
$M \in \text{Measures}$ – the set of measure names
$d \in \text{Data}$ – the set of basic data types supported by the CIM CVL model
$min, max \in \text{Integer constants}$ – the set of non-negative integer constants
5.4 Abstract Syntax

\[ S_D ::= D_D;L_D;PC_D;H_D;F_D;IC_D; \]
\[ D_D ::= D_D_1;D_D_2 \]
  \[ \text{Dimension } D \text{ includes level } L \]
  \[ \text{Dimension } D \text{ includes hierarchy } E \]
\[ L_D ::= L_D_1;L_D_2 \]
  \[ \text{Level } L \text{ has } A_D \]
\[ PC_D ::= PC_D_1;PC_D_2 \]
  \[ \text{P-C relationship } PC \text{ involves } L_1 \text{ as Parent, } L_2 \text{ as Child} \]
\[ H_D ::= H_D_1;H_D_2 \]
  \[ \text{Hierarchy } H \text{ with bottom level } L \text{ is composed of } Q \]
\[ F_D ::= F_D_1;F_D_2 \]
  \[ \text{Fact relationship } F \text{ involves } I_S \]
  \[ \text{Fact relationship } F \text{ involves } I_S \text{ has } P_D \]
\[ P_D ::= M_D \]
  \[ A_D;M_D \]
\[ IC_D ::= IC_D_1;IC_D_2 \]
  \[ K \text{ is primary key of } L \]
  \[ \text{Exclusive participation of } L \text{ in } Q \]
  \[ A \text{ in } L \text{ is non nullable} \]
  \[ \text{Participation of } L \text{ in } PC \text{ is } (\min, \max) \]
\[ A_D ::= A_D_1;A_D_2 \]
  \[ \text{Attribute } A \text{ of } A'_D \]
\[ A'_D ::= \text{type } d \]
\[ M_D ::= M_D_1;M_D_2 \]
  \[ \text{Measure } M \text{ of } A'_D \text{ is } add \]
\[ I_S ::= I_S_1;I_S_2 \]
  \[ D \]
  \[ D \text{ as role} \]
\[ d ::= \text{int} | \text{real} | \text{string} \]
\[ add ::= \text{additive} | \text{semiadditive} | \text{nonadditive} \]
5.5 Example

In this section, we take a simple representative example of a conceptual model in CIM and the textual representation thereof. Figure 5.1 presents a CVL model for a company’s product procurement department. Here, we are trying to analyze the quantity of procurement of different items based on when a product was ordered and delivered; and which branch of the company made the order and from which vendor that product was purchased. Each of these analysis qualifications becomes a separate dimension resulting in four dimensions – Product, Supplier, Date and Location. Each purchase has a unique transaction number, an identity of the product purchased, the date of order, the date of delivery, the branch where the product is delivered, and the vendor of the product. More than one type of product can be purchased in each transaction resulting in multiple records in the Purchase set with the same transaction number.

In the CVL model, there is one fact relationship, Purchase, that stores a transaction number (TransactionID) for an order along with the quantity (Qty) of the product bought during the transaction. Each dimension can have zero, one, or more hierarchies. The Supplier dimension consists of only one level, Vendor; therefore, no hierarchy is needed for this dimension. On the contrary, each of Product, Date and Location consists of a lattice of levels forming a hierarchy. The Product dimension consists of two hierarchies: Brand and Division. A bottom level Item is shared by the two hierarchies. For hierarchy Brand, the level Item rolls up to a top level Trademark; whereas, the level Item rolls up to a top level Category in hierarchy Division. Location has a more complex hierarchy structure (Geography) where the bottom level Branch rolls up to two disjoint levels: State and Province. These two levels join together to reach level Country, the coarsest granularity level of the Location dimension. The Date dimension consists simply of one hierarchy Calendar which allows two alternate aggregation paths to level Year: one path along Day-Month-Bimester-Year and another through Day-Month-Quarter-Year. So, an aggregation for the level Year demands a complex aggregation traversal. As is evident from the diagram, there is an overlap between the two paths at the levels Month and Year. Each roll up forms a parent-child relationship in the model.
Figure 5.1: Product Purchase CVL model
A snippet of the corresponding textual representation for the CVL model in Figure 5.1 is given below:

Level Definitions

Level Vendor has
   Attribute VendorID of type string,
   Attribute VendorName of type string;

Level Item has
   Attribute ItemID of type string,
   Attribute ItemName of type string;

... 

Parent-Child Relationship Definitions

P-C relationship TrademarkItem involves
   Trademark as Parent, Item as Child;

P-C relationship CategoryItem involves
   Category as Parent, Item as Child;

P-C relationship ProvinceBranch involves
   Province as Parent, Branch as Child;

P-C relationship StateBranch involves
   State as Parent, Branch as Child;

P-C relationship CountryProvince involves
   Country as Parent, Province as Child;

P-C relationship CountryState involves
   Country as Parent, State as Child;

P-C relationship MonthDay involves
   Month as Parent, Day as Child;

P-C relationship BimesterMonth involves
   Bimester as Parent, Month as Child;

P-C relationship QuarterMonth involves
Quarter as Parent, Month as Child;
P-C relationship YearBimester involves Year as Parent, Bimester as Child;
P-C relationship YearQuarter involves Year as Parent, Quarter as Child;

Hierarchy Definitions

Hierarchy Brand with bottom level Item
  is composed of TrademarkItem;
Hierarchy Division with bottom level Item
  is composed of CategoryItem;
Hierarchy Geography with bottom level Branch
  is composed of ProvinceBranch, StateBranch, CountryProvince, CountryState;
Hierarchy Calendar with bottom level Day
  is composed of MonthDay, BimesterMonth, QuarterMonth, YearBimester, YearQuarter;

Dimension Definitions

Dimension Supplier includes level Vendor;
Dimension Product includes hierarchy Brand, Division;
Dimension Location includes hierarchy Branch;
Dimension Date includes hierarchy Calendar;

Fact Relationship Definitions

Fact relationship Purchase
  involves Date as OrderDate, Date as DeliveryDate, Supplier, Product, Location
has
  Attribute TransactionID of type string,
Measure Qty of type int is additive;

Constraint Definitions

VendorID is primary key of Vendor;
ItemID is primary key of Item;

TransactionID in Purchase is non nullable;
VendorName in Vendor is non nullable;

Exclusive participation of Branch in ProvinceBranch, StateBranch;
Exclusive participation of Country in CountryProvince, CountryState;
Participation of Trademark in TrademarkItem is (1, n);
Participation of Item in TrademarkItem is (1, 1);
Participation of Category in CategoryItem is (1, n);
Participation of Item in CategoryItem is (1, n);
Participation of Province in ProvinceBranch is (1, n);
Participation of Branch in ProvinceBranch is (0, 1);
Participation of State in StateBranch is (1, n);
Participation of Branch in StateBranch is (0, 1);
Participation of Country in CountryProvince is (0, n);
Participation of Province in CountryProvince is (1, 1);
Participation of Country in CountryState is (0, n);
Participation of State in CountryState is (1, 1);
Participation of Month in MonthDay is (1, n);
Participation of Day in MonthDay is (1, 1);
Participation of Bimester in BimesterMonth is (1, n);
Participation of Month in BimesterMonth is (1, 1);
Participation of Quarter in QuarterMonth is (1, n);
Participation of Month in QuarterMonth is (1, 1);
Participation of Year in YearBimester is (1, n);
Participation of Bimester in YearBimester is (1, 1);
Participation of Year in YearQuarter is (1, n);
Participation of Quarter in YearQuarter is (1, 1);

5.6 Differences from MultiDim

The CIM conceptual model borrows heavily from the MultiDim model, but there are few key points where they contrast. Those key points are as follows:

- **Explicit separation between declarations of dimension and its bottom level.**
  In the MultiDim model, a level connected to a fact relationship implicitly represents the dimension declaration as well. This means the dimension name is forced to take the same name as the level. But a dimension is more than just the bottom level; the dimension represents the entire lattice of levels that can consist of one or more hierarchies. In CIM, a dimension has an separate identity independent from its bottom level allowing different names for a dimension and its bottom level. This makes up for a more expressive conceptual model than MultiDim.

- **Conceptually, a choice of a combination of dimensions is made first while analyzing a fact relationship; the exact granularity of detail for each dimension is decided only after that first step.** If we look at the multidimensional model, its core objective is to analyze fact data with different combinations of business perspectives. So, the natural cognitive tendency during query formulation over a conceptual schema is to start by choosing fact data to analyze and a combination of dimensions providing an analysis context. This is followed by a choice of a suitable granularity of detail — a level represents a distinct granularity of detail — for each dimension. In CVL, we enforce fact relationship to be linked with a dimension. A dimension, in turn, consists of lattice of levels. We argue that a direct connection of a fact relationship with a dimension is akin to this thought process. As a result, this approach is perceived as more natural to the modeling process by the users during data analysis. On the other hand, in MultiDim, a fact relationship is directly connected to the bottom level of a dimension. This approach fails to encourage the more natural thought process envisioned with multidimensional data model.

- **A clear disambiguation of a hierarchy-dimension relationship when a bot-
tom level is shared by multiple hierarchies/dimensions. In CIM, a hierarchy is attached with both the comprising dimension as well as the bottom level of the hierarchy. Although, this may look like a redundancy at a first glance, this notation helps to disambiguate situations where a bottom level is shared by two or more dimensions such that we need to disambiguate which hierarchies (originating from the bottom level) belong to which dimensions. To make the matter clearer, let us consider Figure 5.2(a) where the level Branch belongs to two hierarchies: Geography and Administration. The hierarchy Geography belongs to dimension Warehouse, whereas the hierarchy Administration is a part of dimension Employee. By attaching the hierarchy representations with the dimensions, we clearly disambiguate the relationship between the dimensions and the hierarchies and their exclusivity. So, a fact relationship $X$ can be analyzed using hierarchy Geography only and similarly, a fact relationship $Y$ using hierarchy Administration only. Generally, CIM allows us to represent exclusive hierarchy-fact relationship analysis scenarios. On the other hand, if we were to represent this same scenario in MultiDim, it would look like Figure 5.2(b). Note that, in the latter figure, an exclusive relationship is not clear between a hierarchy and a fact relationship.

- Precise differences between the abstract syntaxes of CVL and MultiDim.

Here, we list the exact points in the abstract syntaxes of CVL and MultiDim, where they differ. Unlike CVL, the abstract syntax for MultiDim also consists of specialized constructs for defining temporal and spatial data warehouses. As a result, the abstract syntax of CVL is a subset of that of MultiDim except that syntax of few constructs in CVL are supplemented with some extra information to make the syntax more expressive. These points of addendum in the syntax of constructs in CVL are shown underlined as follows:

1. Schema declaration:

   \[ S_D ::= D_D; L_D; PC_D; H_D; F_D; IC_D; \]

   In CVL, the hierarchy declarations $H_D$ are expressed in the schema definition to enable dimensions to reference the same hierarchy declaration. This sharing is convenient when an entire hierarchy is shared by more than one dimensions.
2. Parent-child relationship declaration:

\[ PC_D ::= PC_{D_1};PC_{D_2} \]

| P-C relationship \( PC \) involves \( L_1 \) as Parent, \( L_2 \) as Child |

In CVL, we explicitly state that out of the two levels \( L_1 \) and \( L_2 \) participating in a parent-child relationship \( PC \), which one is the parent level and which one is the child level. MultiDim relied on the order of levels in the syntax \( L_1,L_2 \) to imply that first level \( L_1 \) is the parent level and second level \( L_2 \) is the child one. This information is not apparent unless a user digs into the definition of the abstract syntax. We argue that explicitly pointing this information out in the syntax makes it easier for the user to figure out roles of the two levels in the parent-child relationship.

3. Hierarchy relationship declaration:

\[ H_D ::= H_{D_1};H_{D_2} \]

| Hierarchy \( H \) with bottom level \( L \) is composed of \( Q \) |
The bottom level in a hierarchy has a special importance because it is the level from which an aggregation stems up in the hierarchy. By starting from the bottom level, we can navigate to any other level in the hierarchy through a chain of parent-child relationships. For this reason, we explicitly specify the bottom level in a hierarchy declaration in the abstract syntax of CVL.

4. Fact relationship declaration:

\[ F_D ::= F_{D_1}; F_{D_2} \]
\[ | \text{Fact relationship } F \text{ involves } I_S \]
\[ | \text{Fact relationship } F \text{ involves } I_S \text{ has } P_D \]

We recognize that a fact relationship may store properties (e.g., IDs) which are not numerical in nature (i.e., these properties are not measures per se). In order to allow users to declare non-numerical properties in a fact relationship, we allow both properties and measures to be included in a fact-relationship declaration with \( P_D \). Unlike CVL, MultiDim only allows measures to be associated with a fact relationship. \( P_D \) is defined as follows in the abstract syntax of CVL:

\[ P_D ::= M_D \]
\[ | A_D; M_D \]

5. Integrity constraint declaration:

\[ IC_D ::= IC_{D_1}; IC_{D_2} \]
\[ | K \text{ is primary key of } L \]
\[ | \text{Exclusive participation of } L \text{ in } Q \]
\[ | A \text{ in } L \text{ is non nullable} \]
\[ | \text{Participation of } L \text{ in } PC \text{ is } (min, max) \]

We assert that nullability of a property is an integrity constraint widely used in the databases. We allow users to express this constraint in the CVL model.

5.7 Semantics

In this section, we delve into the semantics of the textual representation of the conceptual model for CIM introduced in previous sections. The semantics of the
textual representation describes how we interpret the entities and their relationships captured by the graphical conceptual model and also put limits on the kinds of analyses that can be performed on those data. The conceptual language in CIM is compositional in nature such that the semantics of the composite entities are derived from the semantics of their constituent entities. Put differently, the semantics of the simpler entities such as attributes, levels and parent-child relationships is used to derive the semantics of complex entities like hierarchies and dimensions.

To start with, we define the assumptions behind the conceptual language in CIM followed by the semantics of the data signature (defined in section 5.2) and then work our way to the semantics of the model after describing some auxiliary functions.

5.7.1 Assumptions

There are a few assumptions embraced by our semantics description of the conceptual language that only impact how semantics are defined. These assumptions simplify the semantics of the language. These assumptions are as follows.

- **At the schema level, each conceptual model entity of a type has a unique name within the type.** To make the semantics description simpler, we assume that all the entities of the same type of the conceptual model schema have a unique name. For example, no two attributes within the schema have the same name even though they belong to different levels. So, we directly reference an entity from its comprising entity by its name without ambiguity while describing the semantics. This can be extended to the non-unique scenarios by employing standard techniques like namespaces.

- **Each level has a surrogate key to uniquely identify each member of the level at the instance level.** Each member of the levels represents a distinct entity from the real world; and assigning a surrogate key to each entity becomes necessary to uniquely identify it and reason about it. A surrogate key is supplemented as an attribute of a level and it must have a non null value and be unique for all records within the level. Hence a surrogate key is attached to each entity. This constraint can be explicitly enforced in the schema itself through a primary key constraint.
in our conceptual language. We heavily use surrogate keys in the description of the semantics of our conceptual model in the CIM.

5.7.2 Semantics of the Data Signature

This section is directly copied from the Section A.6.1 from the book [22]; except that we added an example of semantics of the $<_i$ predicate at the end. The semantics of the data signature is given by three functions:

- A function $\mathcal{D}[\text{DATA}] \in TF$ such that $\mathcal{D}[\text{DATA}] : \text{DATA} \rightarrow \text{SET}$ and $\bot \in \mathcal{D}[\text{DATA}](d)$ for every $d \in \text{DATA}$. The membership of $\bot \in \mathcal{D}[\text{DATA}](d)$ is required because $\bot$ represents an undefined value to indicate a null value, an incorrect use of a function or an error.

- A function $\mathcal{D}[\text{OPNS}] \in TF$ such that $\mathcal{D}[\text{OPNS}] : \text{OPNS} \rightarrow \text{TF}$ and $\sigma : d_1 \times \ldots \times d_n \rightarrow d$ implies $\mathcal{D}[\text{OPNS}](\sigma) : \mathcal{D}[\text{DATA}](d_1) \times \ldots \times \mathcal{D}[\text{DATA}](d_n) \rightarrow \mathcal{D}[\text{DATA}](d)$ for every $d \in \text{DATA}$.

- A function $\mathcal{D}[\text{PRED}] \in TF$ such that $\mathcal{D}[\text{PRED}] : \text{PRED} \rightarrow \text{REL}$ and $\pi : d_1 \times \ldots \times d_n$ implies $\mathcal{D}[\text{PRED}](\pi) \subseteq \mathcal{D}[\text{DATA}](d_1) \times \ldots \times \mathcal{D}[\text{DATA}](d_n)$ for every $d \in \text{DATA}$.

The semantics of predefined data types, one of the operators ($+_i$) and a predicate ($<_i$) is defined as follows:

| $\mathcal{D}[\text{DATA}](\text{int})$ | $\mathbb{Z} \cup \{\bot\}$ |
| $\mathcal{D}[\text{DATA}](\text{real})$ | $\mathbb{R} \cup \{\bot\}$ |
| $\mathcal{D}[\text{DATA}](\text{string})$ | $\mathbb{A}^* \cup \{\bot\}$ |
| $\mathcal{D}[\text{OPNS}](+_i)$ | $\mathcal{D}[\text{DATA}](\text{int}) \times \mathcal{D}[\text{DATA}](\text{int}) \rightarrow \mathcal{D}[\text{DATA}](\text{int})$ |
| | $\begin{cases} i_1 \times i_2 \rightarrow i_1 + i_2 & \text{if } i_1, i_2 \in \mathbb{Z}, \\ \bot & \text{otherwise.} \end{cases}$ |
| $\mathcal{D}[\text{PRED}](<_i)$ | $\mathcal{D}[\text{DATA}](\text{int}) \times \mathcal{D}[\text{DATA}](\text{int})$ |
| | $\begin{cases} i_1 \times i_2 & \text{if } i_1, i_2 \in \mathbb{Z} \land i_1 \text{ is less than } i_2, \\ \bot & \text{otherwise.} \end{cases}$ |
5.7.3 Surrogate Domains

The CIM conceptual language includes the following value domains for surrogates:

- $D_S \cup \{\bot\}$ – the set of surrogates
- $D^L_S \subseteq D_S$ – the set of surrogates assigned to $L \in \text{Levels}$
- $D^{PC}_S \subseteq D_S$ – the set of surrogates assigned to $PC \in \text{PCrels}$

5.7.4 Auxiliary Functions

Auxiliary functions define a set of functions that help in defining semantics of the conceptual language artifacts.

$\text{attOf}$ The $\text{attOf}$ function takes as its argument a declaration of levels or attributes and returns a list of names of their attribute names.

$\text{attOf}(\text{Level } L \text{ has } A_D) = \text{attOf}(A_D)$
$\text{attOf}(A_D_1,A_D_2) = \text{attOf}(A_D_1) \cup \text{attOf}(A_D_2)$
$\text{attOf}(\text{Attribute } A \text{ of } A'_D) = A$

$\text{inSch}$ The predicate $\text{inSch}$ takes as its first argument the name of a level, a parent-child relationship or a fact relationship, and a schema declaration as its second argument. The predicate returns $\text{true}$ if the first argument is declared in the schema declaration; otherwise, returns $\text{false}$.

$\text{inSch}(L,S_D) = \text{inSch}(L,D_D;L_D;PC_D;H_D;F_D;IC_D;) = \text{inSch}(L,L_D)$

$$= \begin{cases} 
\text{true} & \text{if Level } L \text{ has } A_D \in L_D,
\text{false} & \text{otherwise}.
\end{cases}$$

$\text{inSch}(PC,S_D) = \text{inSch}(PC,D_D;L_D;PC_D;H_D;F_D;IC_D;) = \text{inSch}(PC,PC_D)$

$$= \begin{cases} 
\text{true} & \text{if P-C relationship } PC \text{ involves } \ldots \text{ as Child } \in PC_D,
\text{false} & \text{otherwise}.
\end{cases}$$

$\text{inSch}(F,S_D) = \text{inSch}(F,D_D;L_D;PC_D;H_D;F_D;IC_D;) = \text{inSch}(F,F_D)$
true  if Fact relationship $F$ involves $I_S \in F_D$,
true  if Fact relationship $F$ involves $I_S$ has $P_D \in F_D$,
false otherwise.

cnt The function $cnt$ takes a level member $l$ as its first argument, a level $L$ as the
second argument, and a set of instances of a parent-child relationship as its third
argument and returns the number of tuples in the relationship set that the level $l$
participates in.

$$cnt(l, L, \{t_1, \ldots, t_n\}) = \begin{cases} 0 & \text{if } n = 0, \\ cnt(l, L, \{t_1, \ldots, t_{n-1}\}) & \text{if } n \geq 1 \land t_n[s_L] \neq e, \\ cnt(l, L, \{t_1, \ldots, t_{n-1}\}) + 1 & \text{if } n \geq 1 \land t_n[s_L] = e. \end{cases}$$

bottomLevel The function $bottomLevel$ takes a dimension name as its argument
and returns the bottom level of the dimension.

$$bottomLevel(D) = \begin{cases} L & \text{Dimension } D \text{ includes level } L \in S_D, \\ L & \exists H \in E (\text{Dimension } D \text{ includes Hierarchy } E \in S_D) \land \\ & (\text{Hierarchy } H \text{ with bottom level } L \ldots \in S_D). \end{cases}$$

role Since a dimension can participate multiple times in a fact relationship, a role
is assigned to uniquely identify each participation. The function $role$ takes a de-claration of a dimension as an input and returns the name of the role.

$$role(D as role) = role$$
$$role(D) = D$$

dimension The function $dimension$ takes the involvement of a dimension in a fact
relationship and returns the dimension name as:

$$dimension(D) = dimension(D as role) = D$$
The function \( \text{parOf} \) takes the name of a parent-child relationship as its only argument and returns the levels involved in that relationship.

\[
\text{parOf}(PC) = \begin{cases} 
\text{parOf(P-C relationship } PC \ldots) & \text{P-C relationship } PC \ldots \in S_D, \\
\bot & \text{otherwise.}
\end{cases}
\]

\( \text{parOf}(\text{P-C relationship } PC \text{ involves } L_1 \text{ as Parent, } L_2 \text{ as Child}) = \{L_1, L_2\} \)

### 5.8 General Description of Semantics

In this section, we use the auxiliary functions defined in the previous section (Section 5.7.4) to describe the general semantics of the components of our conceptual language. We describe the semantic functions first and then follow them by their general descriptions. Utilizing the compositional nature of the conceptual language, we first describe simpler components such as levels, attributes and parent-child relationships; and then use them to describe other more complex structures like hierarchies and dimensions.

#### 5.8.1 Surrogate Key

The semantic function \( \mathcal{S} \) determines the set of surrogate keys of the levels involved in a parent-child relationship, or a dimension involved (using a role) in a fact relationship. Surrogate keys of a dimension are the surrogate keys of the bottom level of the dimension.

\[
\mathcal{S} : \text{Levels} \cup \text{Dimensions} \rightarrow D_S^L
\]

\[
\mathcal{S}[I_{S_1}:I_{S_2}] = \mathcal{S}[I_{S_1}] \times \mathcal{S}[I_{S_2}]
\]

\[
\mathcal{S}[I_S] = \mathcal{S}[\text{bottomLevel}(\text{dimension}(I_S))]
\]

\[
\mathcal{S}[L] = \begin{cases} 
D_S^L & \text{if } L \in \text{Levels}, \\
\bot & \text{otherwise.}
\end{cases}
\]
5.8.2 Attribute and Measure

The semantic function \( \mathcal{A} \) defines the value domains of attribute and measure declarations and their combination: \( A_D, M_D \) and \( P_D \) respectively.

\[ \mathcal{A} : \left( Attributes \cup Measures \right) \times DATA \rightarrow \mathcal{P}[DATA] \]

\[ \mathcal{A}[A_{D_1}, A_{D_2}] = \mathcal{A}[A_{D_1}] \times \mathcal{A}[A_{D_2}] \text{ where } A_{D_1}, A_{D_2} \in A_D | M_D | P_D \]

\[ \mathcal{A}[\text{Attribute } A \text{ of } A_D'] = \mathcal{A}[A_D'] \]

\[ \mathcal{A}[\text{Measure } M \text{ of } A_D' \text{ is add}] = \mathcal{A}[A_D'] \]

\[ \mathcal{A}[\text{type } d] = \begin{cases} \mathcal{P}[DATA](d) & \text{if } d \in DATA, \\ \bot & \text{otherwise.} \end{cases} \]

5.8.3 CVL Schema

The semantic function \( \mathcal{S} \) defines the semantics of the schema description of the entire conceptual model which consists of the declaration of dimensions, levels, parent-child relationships, hierarchies, fact relationships and integrity constraints. The semantic evaluation of the schema declaration is the accumulation of the semantics of the constituents of the schema.

\[ \mathcal{S} : S_D \rightarrow \mathcal{S}[S_D] \]

\[ \mathcal{S}[S_D] = \mathcal{S}[D_D; D_D; PC_D; H_D; F_D; IC_D] \]

\[ \mathcal{S}[D_D; L_D; PC_D; H_D; F_D; IC_D] = \mathcal{S}[D_D] \sqcup \mathcal{L}[L_D] \sqcup \mathcal{P}[PC_D] \sqcup \mathcal{H}[H_D] \sqcup \mathcal{F}[F_D] \sqcup \mathcal{I}[IC_D] \]

5.8.4 Level

The semantic function \( \mathcal{L} \) defines the semantics of a level as a set of tuples that associates the attributes of a level with the values from associated domains. Further, the function \( \mathcal{L} \) applied to a composition of level definitions returns the disjoint union of the functions applied individually to each component.

\[ \mathcal{L} : Levels \times AttDecl \rightarrow \mathcal{S}[L] \times \mathcal{A}[A_D] \]

\[ \mathcal{L}[L_{D_1}; L_{D_2}] = \mathcal{L}[L_{D_1}] \sqcup \mathcal{L}[L_{D_2}] \]
\( \mathcal{L}[\text{Level } L \text{ has } A_D] \)
\[
= \{ t \mid t \in TF \land \text{dom}(t) = \{ s, attO f(A_D) \} \land t[s] \in D^t_{L} \land \\
\forall A_i \in attO f(A_D)(t[A_i] \in \mathcal{A}[\text{Attribute } A_i \text{ of } A_D] \}
\]

5.8.5 Parent-child Relationship

The semantic function \( \mathcal{P}C \) defines the semantics of a parent-child relationship as a set of tuples relating surrogates of the participating parent and child levels. Since each parent-child relationship defines a unique set of tuples, the function \( \mathcal{P}C \) applied to a composition of parent-child relationships definitions returns the disjoint union of the function applied individually to each component.

\( \mathcal{P}C : PCrels \times Levels \times Levels \rightarrow \mathcal{I}[L] \times \mathcal{I}[L] \)
\[
\mathcal{P}C[PC_{D_1}, PC_{D_2}] = \mathcal{P}C[PC_{D_1}] \cup \mathcal{P}C[PC_{D_2}]
\]
\( \mathcal{P}C[\text{P-C relationship } PC \text{ involves } L_1 \text{ as Parent, } L_2 \text{ as Child}] \)
\[
= \{ t \mid t \in TF \land \text{dom}(t) = \{ s_{L_1}, s_{L_2} \} \land t[s_{L_1}] \in \mathcal{I}[L_1] \land t[s_{L_2}] \in \mathcal{I}[L_2] \}
\]

5.8.6 Fact Relationship

The semantic function \( \mathcal{F} \) defines the semantics of fact relationships as a set of tuples such that each tuple consists of a bottom level member of each of its dimensions including values for its attributes and measures. When the function \( \mathcal{F} \) is applied to a composition of fact relationship definitions, it returns the disjoint union of the functions applied individually to each component.

\( \mathcal{F} : FactRels \times Invspec \times AttDecl \times MeasDecl \)
\[
\rightarrow \mathcal{I}[I_S] \cup \mathcal{I}[I_S] \times \mathcal{I}[A_D \cup M_D]
\]
\( \mathcal{F}[F_{D_1}, F_{D_2}] = \mathcal{F}[F_{D_1}] \cup \mathcal{F}[F_{D_2}] \)
\( \mathcal{F}[\text{Fact relationship } F \text{ involves } I_S] \)
\[
= \{ t \mid t \in TF \land \text{dom}(t) = \{ \bigcup_{D_i \in I_S} s_{\text{role}(D_i)} \} \land \\
\forall D_i \in I_S(t[s_{\text{role}(D_i)}] \in \mathcal{I}[D_i]) \}
\]
\( \mathcal{F}[\text{Fact relationship } F \text{ involves } I_S \text{ has } P_D] \)
\[
= \{ t \mid t \in TF \land \text{dom}(t) = \{ \bigcup_{D_i \in I_S} s_{\text{role}(D_i)}, attO f(P_D) \} \land \\
\forall P_i \in \text{attO f}(P_D)(t[P_i] \in \mathcal{A}[\text{Attribute } P_i \text{ of } A_D] \}} \}
5.8.7 Integrity Constraints

The semantic function \( \mathcal{FC} \) defines the semantics of integrity constraints in the CIM. A valid instance of a conceptual model in the CIM should satisfy all the constraints altogether.

\[
\mathcal{FC} : IC_D \rightarrow PRED \\
\mathcal{FC}[IC_{D_1};IC_{D_2}] = \mathcal{FC}[IC_{D_1}] \land \mathcal{FC}[IC_{D_2}]
\]

Primary Key

The primary key constraint asserts that the values of the key attributes are unique for all members of the level.

\[
\mathcal{FC}[K \text{ is primary key of } L] = inSch(L,S_D) \land ((K \subseteq attOf(L \text{ has } A_D)) \land \\
\forall t_i,t_j \in L[\text{Level } L \text{ has } A_D](t_i[K] = t_j[K] \Rightarrow t_i[s] = t_j[s]))
\]

Nullability

The nullability constraint ensures a null value can be assigned to an attribute. By default, an attribute is nullable.

\[
\mathcal{FC}[A \text{ in } L \text{ is non nullable}] = inSch(L,S_D) \land A \in attOf(L) \land \\
\forall t \in L[\text{Level } L \text{ has } A_D](t[A] \neq \bot)
\]

Exclusive Participation

A member from a splitting or joining level participates exclusively in a set of parent-child relationships constrained by disjointness.

\[
\mathcal{FC}[\text{Exclusive participation of } L \text{ in } Q]
\]
A cardinality constraint is used to impose a restriction on the count of the counterpart level members’ participation with a member of a given level in a given parent-child relationship within a specified bracketing value-pair. The minimum of the bracket is $\min$ and the maximum is $\max$.

$\mathcal{P} \mathcal{C}[[\text{Participation of } L \text{ in } PC \text{ is } (\min, \max)]]$

$= inSch(L, S_D) \land inSch(PC, S_D) \land L \in \text{parOf}(PC) \land \forall l \in D^L_S$

$(\min \leq \text{cnt}(l, L, \mathcal{S} \mathcal{C}[\text{PC relationship PC}_i \text{ involves } \ldots]) \leq \max)$

### 5.8.8 Hierarchies and Dimensions

Hierarchies and Dimensions provide a structure to the conceptual schema by grouping together the levels that are related through parent-child relationships. The inclusion of the hierarchies and dimensions in the conceptual language in the CIM is strictly as a design artifacts to form meaningful groups of the levels. Hence no application data is associated with these entities.

A hierarchy consists of a set of levels such that its elements are arranged in monotonically decreasing order of fineness of granularity along with a set of parent-child relationships between those levels. A dimension consists of a level or a set of hierarchies sharing the bottom level between them.

Hierarchies and dimensions are essential for defining aggregations in OLAP. The mathematical descriptions of a hierarchy and a dimension are provided in Chapter 4.
5.9 Semantics of the Running Example

We use the the general semantics developed in Section 5.8 to describe the semantics of the CVL model from the running example from Section 5.5.

5.9.1 Surrogate Key

We begin by defining a semantic function $\mathcal{I}$ that determines the set of surrogate keys of the levels involved in a parent-child relationship or a fact relationship. Here, we show by example that the surrogate keys for level $\text{Vendor}$ is a set of tuples denoted as $D^\text{Vendor}_S$.

$$\mathcal{I}[\text{Vendor}] = D^\text{Vendor}_S$$

5.9.2 Level and Attribute

A level is an abstraction of a real-world entity described by its properties. In other words, a set of properties of the level characterizes it and gives it an identity. At the schema level, a Level $L$ defines a named entity type which consists of a set of attributes $A_D$. Similarly, an attribute is a named property that is assigned a data type at design time. At the instance level, a level $L$ consists of a set of entities of type $L$ each composed of a set of attribute values. The level $\text{Vendor}$ in the example schema description in Section 5.5 defines an entity type whose attributes are $\text{VendorID}$ and $\text{VendorName}$. We are only concerned with these two properties of Vendors for our analysis, so our abstraction only captures these two attributes.

Each attribute of a level has a value domain. A function $\text{dom} : A_D \rightarrow \mathbb{D}$ describes the association between the set of attributes $A_D = \{A_1, A_2, \ldots, A_n\}$ and the set of domains $\mathbb{D}$. A member of a level composed of a set of attributes can be treated as a tuple. A tuple $t$ over a set of attributes $A_D$ is actually a function that associates each attribute $A_i \in A_D$ with a value from the value domain $\text{dom}(A_i)$. Further, $t[A]$ denotes the value assigned to attribute $A$.

The semantic function $\mathcal{A}$ defines the value domains of attribute declarations $A_D$. Both attributes $\text{VendorID}$ and $\text{VendorName}$ from the level $\text{Vendor}$ have data type $\text{string}$ which translates to
\[ \mathcal{A}[\text{Attribute VendorID of type string}] = \mathcal{D}[\text{DATA}(\text{string})] = \mathbb{A}^* \cup \{\perp\} \]
\[ \mathcal{A}[\text{Attribute VendorName of type string}] = \mathcal{D}[\text{DATA}(\text{string})] = \mathbb{A}^* \cup \{\perp\} \]

The semantics of the attributes defined above are used to derive semantics of the comprising level Vendor. The semantics of the level type Vendor is a set of functions (tuples) termed Vendor. Each tuple associates every attribute with a value from its corresponding domain. So each tuple can be defined as a function whose domain is the set of attributes and its range is the set of values from the corresponding value domains of the attributes determined by the semantics of attribute declarations.

The semantic function \( \mathcal{L} \) defines a set of tuples that associates the attribute of a level with the values from associated domain. For level Vendor, the attribute VendorID is a primary key so it can be the surrogate key for the level.

\[ \mathcal{L}[[\text{Level Vendor has...}]] = \{ t \mid t \in \mathcal{T}_F \land \text{dom}(t) = \{\text{VendorID}, \text{VendorName}\} \land \\
\quad t[\text{VendorID}] \in \mathcal{A}[\text{Attribute VendorID of type string}] \land \\
\quad t[\text{VendorName}] \in \mathcal{A}[\text{Attribute VendorName of type string}] \} \]

\[ = \{ t \mid t \in \mathcal{T}_F \land \text{dom}(t) = \{\text{VendorID}, \text{VendorName}\} \land \\
\quad t[\text{VendorID}] \in \mathbb{A}^* \cup \{\perp\} \land \\
\quad t[\text{VendorName}] \in \mathbb{A}^* \cup \{\perp\} \} \]

### 5.9.3 Parent-child Relationship

A parent-child relationship type defines a set of parent-child relationships between members of parent and child levels. More formally, each parent-child relationship, tuple \( t \), relates a parent level and a child level through their associated surrogate keys. A semantic function \( \mathcal{PC} \) defines a set of tuples that associates each of the two participating levels with its value from associated surrogate key domain.

For example, the levels Category and Item participates in a parent-child relationship CategoryItem. The semantic function \( \mathcal{PC} \) defines CategoryItem as
follows.

\[ P-C \text{ relationship CategoryItem involves } \ldots ] \]
\[ = \{ t \mid t \in TF \land \text{dom}(t) = \{ \text{CategoryID}, \text{ItemID} \} \land \]
\[ t[\text{CategoryID}] \in D_{S}^{\text{Category}} \land \]
\[ t[\text{ItemID}] \in D_{S}^{\text{Item}} \} \]

5.9.4 Fact Relationship

At the schema level, a fact relationship type relates several dimensions and includes some attributes and measures. However, dimensions are just schema artifacts; so at the instance level, the bottom levels of those dimensions participate in the fact relationships. Hence, a fact relationship, at the instance level, is a set of tuples \( t \) such that each tuple consists of a bottom level member from each of the associated dimensions including values for its attributes and measures. Note that when we say a dimension is participating in a fact relationship, it is actually its bottom level that is representing the dimension. So a fact relationship always stores data at the finest granularity of the associated dimensions. A dimension can participate multiple times in a fact relationship, using different roles. In this case, the name of the role is used instead of the name of the dimension. If a dimension participates only once then its name can be considered the role of the dimension. So, the set of tuples \( t \) is a Cartesian product over surrogates of the roles of participating dimensions with the value domains of the attributes and the measures. Members of the participating dimensions are identified through their surrogates, with the value domain defined by \( \mathcal{S} \).

The semantic function \( \mathcal{F} \) defines the semantics of a fact relationship as a set of tuples as shown below.

\[ \mathcal{F}[\text{Fact relationship Purchase involves } I_{S} \text{ has } P_{D}] \]
\[ = \{ t \mid t \in TF \land \]
\[ \text{dom}(t) = \{ \text{OrderDate}, \text{DeliveryDate}, \text{Supplier}, \text{Product}, \text{Location}, \]
\[ \text{TransactionID}, \text{Qty} \} \land \]
\[ t[\text{OrderDate}] \in \mathcal{F}[\text{bottomLevel}(\text{Date})] \land \]
\[
\begin{align*}
t[\text{DeliveryDate}] & \in \mathcal{I}[\text{bottomLevel}(\text{Date})] \land \\
t[\text{Supplier}] & \in \mathcal{I}[\text{bottomLevel}(\text{Supplier})] \land \\
t[\text{Product}] & \in \mathcal{I}[\text{bottomLevel}(\text{Product})] \land \\
t[\text{Location}] & \in \mathcal{I}[\text{bottomLevel}(\text{Location})] \land \\
t[\text{TransactionID}] & \in \mathcal{A}[\text{Attribute TransactionID of type string}] \land \\
t[\text{Qty}] & \in \mathcal{A}[\text{Measure Qty of type int is additive}] \\
\end{align*}
\]

\[
\begin{align*}
= \{ t \mid t \in \mathcal{TF} \land \\
\text{dom}(t) & = \{ \text{OrderDate}, \text{DeliveryDate}, \text{Supplier}, \text{Product}, \text{Location}, \\
& \text{TransactionID}, \text{Qty} \} \land \\
t[\text{OrderDate}] & \in \mathcal{I}[\text{Level Day has . . .}] \land \\
t[\text{DeliveryDate}] & \in \mathcal{I}[\text{Level Day has . . .}] \land \\
t[\text{Supplier}] & \in \mathcal{I}[\text{Level Vendor has . . .}] \land \\
t[\text{Product}] & \in \mathcal{I}[\text{Level Item has . . .}] \land \\
t[\text{Location}] & \in \mathcal{I}[\text{Level Branch has . . .}] \land \\
t[\text{TransactionID}] & \in \mathcal{A}[\text{Attribute TransactionID of type string}] \land \\
t[\text{Qty}] & \in \mathcal{A}[\text{Measure Qty of type int is additive}] \\
\end{align*}
\]

5.9.5 Integrity Constraint

Integrity constraints are restrictions imposed on a conceptual model through a set of predicates. So a semantics of a constraint is a set of predicates that a conceptual
model must satisfy. As specified in the abstract syntax in Section 5.4, the integrity constraints in the textual representation are specified as separate constructs, so the predicates must first verify whether the constructs (e.g., levels, relationships and attributes) used in the predicates exist in the schema.

We use semantic function \( \mathcal{IC} \) to define the semantics of the integrity constraints. First we define the primary key constraint which carries its usual meaning. A primary key constraint ensures that the value of the key is unique for all the members of the level. Stated differently, if two tuples in a level have the same value for a primary key, then they are the same entity.

\[
\mathcal{IC}[[\text{VendorID is primary key of Vendor}]] = \text{inSch(Vendor, } S_D) \land \\
\text{VendorID } \in \text{attOf(Level Vendor has } A_D) \land \\
\forall t_i, t_j \in \mathcal{L}[[\text{Level Vendor has } A_D]](t_i[\text{VendorID}] = t_j[\text{VendorID}] \Rightarrow t_i[s] = t_j[s])
\]

A nullability constraint ensures whether an attribute is allowed to have a null value, which means either a value is missing or not applicable (i.e., it does not exist at all for this instance). A null value is denoted by \( \perp \) in our conceptual language.

\[
\mathcal{IC}[[\text{VendorName in Vendor is non null}] = \text{inSch(Vendor, } S_D) \land \text{VendorName } \in \text{attOf(Vendor)} \land \\
\forall t \in \mathcal{L}[[\text{Level Vendor has } \ldots ]](t[\text{VendorName}] \neq \perp)
\]

An exclusive parent-child relationship dictates that a member from a common splitting (or joining) level among these parent-child relationships rolls up (or drills down) exclusively to one of the parent (or child) levels. So, a member from the common level between the disjoint relationships is involved exclusively in only one parent-child relationship.

\[
\mathcal{IC}[[\text{Exclusive participation of Branch in ProvinceBranch, StateBranch}]] = \text{inSch(Branch, } S_D) \land \\
\text{inSch(ProvinceBranch, } S_D) \land \text{inSch(StateBranch, } S_D) \land
\]
\[
\neg \exists t_1 \in \mathcal{PC}[[\text{P-C relationship ProvinceBranch involves } \ldots]]
\]
\[
\exists t_2 \in \mathcal{PC}[[\text{P-C relationship StateBranch involves } \ldots]]
\]
\[
(t_1[s_{Branch}] = t_2[s_{Branch}])
\]

The cardinality constraint ensures that in a parent-child relationship, a member of a level can be related to a minimum of \( \min \) and a maximum of \( \max \) number of another level’s members. We use an auxiliary function \( \text{cnt}(l, L, \mathcal{PC}[[\text{P-C relationship CategoryItem}]]) \) to count the number of occurrences of member \( l \) of level \( L \) in the domain of a parent-child relationship \( \mathcal{PC} \).

\[
\mathcal{IC}[[\text{Participation of Category in CategoryItem is } (1,n)]]
\]
\[
= \text{inSch}(\text{Category}, S_D) \land \text{inSch}(\text{CategoryItem}, S_D) \land \\
\text{Category} \in \text{parOf}(\text{CategoryItem}) \land \\
\forall l \in D_{\text{Category}} \\
\quad (\min \leq \text{cnt}(l, \text{Category}, \mathcal{PC}[[\text{P-C relationship CategoryItem } \ldots]]) \leq \max)
\]

In this chapter, we gave the formal semantics of the constructs in CVL by starting with the general descriptions and concluding with the specific examples that exhibits the use of the semantics.
Chapter 6

Implementation

This chapter discusses the implementation of the graphical editor component of Model manager. First, we argue our choice of framework to build our graphical editor. Next, we dissect the framework to explain its components at an abstract level and then we discuss the implementation phases in more detail. We close the chapter with the instructions on how to use the graphical editor.

6.1 Graphical Modeling Framework

The Graphical Modeling Framework (GMF) [8] is a graphical editor building tool developed for the Eclipse Platform [3]. The framework provides the basic features expected in a generic graphical editor such as menus for commands, a canvas to visualize figures, a drawing tool palette etc. These features can be used either as is or be modified/extended depending on a domain-specific language we are trying to target and the desired sophistication of the features. It takes a few configuration files as input and generates a plug-in that runs on Eclipse as a professional-looking graphical editor [5]. In the editor, diagrams based on the particular domain-specific language can be designed and the result can be serialized to an XML representation. In our case, the domain-specific language is the Conceptual language in CIM which consists of semantic models — a software engineering jargon — such as levels, fact relationships, their properties, hierarchies, dimensions and disjoints. These semantic models dictate the graphical shapes that constitute the output dia-
gram in the graphical editor. So, these semantic models form the linchpin of the graphical editor development process in the GMF. Hence, the GMF is based on the Model-driven approach of software engineering.

In CIM, we created a graphical editor using the GMF, called the Model Manager, that allows users to graphically design the Conceptual Visual Language (CVL) model, visualize the Store Visual Language (SVL) model and enable mapping between them. These conceptual and store models and mapping are serialized in the form of an XML file into its CVL, SVL and MVL elements respectively.

6.2 Motivation

Building a graphical editor is a highly time-consuming and graphic-intensive task and developing a graphical editor with extensible features is even harder. The GMF mitigates this daunting undertaking by providing an application framework that provides both design time and runtime support for building and executing graphical editors. An application framework consists of a set of cooperating classes that, like a scaffolding, forms an architectural structure of the application. The framework also provides a flow of control in the application along with the minimal functionalities generally expected in any application of a particular domain. These functionalities are implemented in the form of generic functions (that users can override) or subclasses depending on the specific details of the problem at hand. Similarly, the GMF provides a reusable design for a graphical editor applicable to the Model manager in CIM. The GMF reduced the time and effort needed to build the Model manager by manyfold. Further, the involvement of a big community of developers working on continual improvement of the code makes the framework robust and reliable.

6.3 Implementation

In this section, we describe how the GMF works and how it was used in our project to implement the Model manager. The GMF project adopted the term “toolsmith” to refer to developers who use the framework to develop graphical editors, and the term “practitioner” to refer to users who utilize those editors [8]. So based on this terminology, we are the toolsmith who developed the Model manager, and the
users of the Model manager are its practitioners.

As stated earlier, the GMF provides two components:

- **Generative component** The generative component enables a toolsmith to automatically generate the Java source code for a fully functioning graphical editor using a set of input configuration files. These configuration files can be either handcrafted or built using a set of software wizards provided by the GMF dashboard \(^1\) (Figure 6.1). The wizard guides the toolsmith along the code generation process by asking a series of questions essential to the code being generated. In the GMF, the configuration files allow the toolsmith to fill in the details of the framework that are specific to the domain of the problem. Examples of these details are the metamodel of the domain specific language, shapes of the semantic models from the metamodel, their graphical attributes, etc. The GMF uses a generative approach on top of those configuration files to produce the Java code. The code thus generated follows popular software design patterns so that it is easier to understand and maintain. The generated code may not meet all the fine details of the user’s requirements so the code can be further modified or subclassed by the toolsmith to develop the final code for the editor.

- **Runtime component** The runtime component provides APIs used by the Java code generated by the generative component during its execution. The runtime provides other sophisticated features such as transaction processing, persistence of the diagrams etc.

### 6.3.1 Generative Component

Figure 6.2 shows a simplified diagram illustrating the generative component. As shown in the diagram, there are five configuration files: Domain model (*.ecore), Graphical definition (*.gmfgraph), Tooling definition (*.gmftool), Mapping model (*.gmfmap) and Generator model (*.gmfgen). We give the descriptions of the

\(^1\)The GMF dashboard is an Eclipse view provided by the GMF framework accompanying every GMF project.
configuration files in the Generative component based on clarifications provided in [6, 8].

- **Domain model**: The domain model defines the grammar of the domain specific language. This grammar defines the entities of the language and how they are associated with each other. This puts limits on what attributes are allowed on entities, their data types, association between entities, etc. As a result, the domain model captures the metamodel of the domain specific language. There is a graphical counterpart of the domain model that uses UML notation to allow us graphically define the metamodel.

- **Graphical definition**: The shapes of different graphical entities allowed in the diagram are described in the configuration files. These shapes act as a decoration for the entities from the metamodel. The graphical definition allows us to describe orientation of the shapes, their color, their types (for instance, nodes or connections), etc.

- **Tooling definition**: The tooling definition file enables us to define the palette and the tools in the palette that will allow a user to design a diagram on the canvas. Moreover, any menu or toolbars can be defined here.

- **Mapping model**: The domain model, graphical definition and tooling definition separately define different aspects of a graphical editor. The mapping
Generator model: The graphical definition, tooling definition, mapping model and domain model store data at a high level not suitable for direct generation of the code. So, a generator model is created that combines the data from all four model files and fills up generator-specific details like name of the classes/packages, etc. The generator model allows further fine-tuning of the code based on the toolsmith’s implementation-specific requirements.
Figure 6.3: The domain model for the Conceptual language in CIM
6.3.2 Steps of Implementation

The code building process for the Model manager is as follows:

1. The generation process begins by creating a GMF project.

2. A domain model (*.ecore file) is created to represent the metamodel of the conceptual language in CIM. The metamodel is built graphically as a *.ecore_diagram file based on the UML representation. Figure 6.3 shows the metamodel for the conceptual model in CIM as a UML class diagram. The entities of the conceptual model are defined as classes. In the diagram, the CIM class is the main class that represents the entire diagram (a canvas). Similarly, CVL, SVL and Mapping represent the conceptual visual model, store visual model and the mapping between them respectively within the CIM diagram; and so on for other classes.

One class can relate to another by using standard UML relationship concepts: an association, an inheritance or an aggregation. An association, depicted as a simple arrow, represents a relationship from one class to another. It also facilitates the participation cardinality. For instance, consider classes LevelProperty and Level and an association level from Level to LevelProperty. This association suggests that each LevelProperty is associated with a Level. Further, each LevelProperty must associate with one and only one Level. An inheritance is denoted by an arrow with a hollow triangular head; and it points from a subclass to its superclass. The inheritance carries its usual meaning from the UML class diagram. In the diagram, all three classes LevelProperty, FactProperty and Measure inherit from the class Property. So, all the attributes and relationships belonging to the class Property are inherited by its subclasses. There are a few classes: LevelOrJoin, LatticeEntity and HierarchyOrLevel which are created to make an exclusive association with one entity out of a group of different entities easier. To explain with an example, a parent-child relationship can have a Level or a Join class as its parent but only one of them at a time. We created a class LevelOrJoin that generalizes a Level and a Join and then associated it with a ParentChildRelationship class and puts the cardinality limit of 0…1 so that a parent-child
relationship can only refer to one of either a *Level* or a *Join* at a time. Similar explanations hold for other generalized classes, that is, *LatticeEntity* and *HierarchyOrLevel*. An aggregation exhibits composite-component relationship and it is shown by an arrow with solid diamond head pointing from a component to its composite. A composite is made up of its components. In the diagram, the class *CIM* is composed of the classes *CVL*, *SVL* and *Mapping*. A user-defined data type can also be defined in the domain model. Three custom data types *PropertyType*, *Operator* and *Cardinality* are shown in the top left corner of the diagram and their descriptions are self-explanatory.

The domain model not only serves to capture the metamodel of the conceptual language in CIM, but it also dictates the format of serialization of the diagram in an XML file. Since, the *CIM* class is composed of all other classes directly or indirectly, all its component classes will appear as nested elements in the output XML file. In other words, *CIM* forms a root node of the XML file; and *CVL*, *SVL* and *Mapping* nodes appear as its child nodes.

3. The graphical description of entities from the metamodel are defined in the Graphical definition file (*gmfgraph*). Figure 6.4 shows the contents of the graphical definition in the form of a tree. The Node elements in the tree represent the nodes in the conceptual model. For instance, Node Level represents the node Level in the diagram and refers to the graphical property of Level such as its rectangular shape, length, width, etc. Similarly, Connection elements depict the links drawn between the nodes in the conceptual model. An example of this is LevelProperties which is a connection between a Level and a LevelProperty. The Compartment elements are the boxes that contain within it other nodes from the conceptual model. For example, The CVL compartment contains other conceptual visual language elements like Level, LevelProperty, Factrelationship, etc. within it. The labels supplied to each Node, Connection and Compartment are described by Diagram Label elements and then associated with the related elements.

4. The tools used to create compartments, nodes and connections in the canvas in the Model manager are defined in the tooling definition file. At first, a palette is initialized which contains all the tools. The tools are categorized
Figure 6.4: The Graphical definition of the Conceptual model in CIM
Figure 6.5: The Tooling definition of the Conceptual model in CIM

into four distinct tool groups: Compartments, CVL Nodes, SVL Nodes and Links. The Compartments tool group stores the creation tool for the CVL and SVL compartments. The CVL and SVL nodes contain the node creation tools for CVL and SVL models, respectively. Finally, the Links tool group is comprised of three tools, namely, Link, PCRelationship and Mapping. The Link creation tool is a multipurpose connection tool which is used to connect many node pairs in both the CVL and SVL compartments. The PCRelationship and Mapping creation tools are used to make a connection between two levels, and between a property (LevelProperty, Measure and Factproperty) from a CVL model and a column from a SVL model, respectively.

5. Once the domain model, the graphical definition and the tooling definition files are defined, the next logical step is to bind them together by mapping the descriptions of an entity of the CIM conceptual language distributed among those three files. As shown in Figure 6.6, a Node Mapping maps an entity from the domain model to a graphical node in the graphical definition file
along with a creation tool (not visible in the Figure) from the tooling definition. Similarly, a Link mapping construct maps an entity from the domain model to a connection in the graphical definition and a creation tool from the tooling definition.

6. Next, the generator model was built from the mapping definition. Here, few properties of the graphical editor for the Model manager were changed. For instance, the layout for the CVL and SVL compartments were set to null so that entities can be placed randomly within the compartments. Also, the extension of the output XML files and the decision to keep the semantic model and diagram model in separate files was finalized.

7. A Rich Client Platform (RCP) application was generated from the generator definition file which contains the Java code. The application can be run as a plug-in in the Eclipse platform.

8. Although the input configuration files for the generative component allowed almost all the feature-adding and fine-tuning to the Model manager, there were still some details we had to get our hands dirty with for direct modification of the generated Java code [4 7]. To name a few: to add shadows to symbols for fact relationships and dimensions, to enable cardinality visuals on the parent-child relationship symbol, to allow underlining of the names of properties or columns that are primary keys, etc. To make sure that the direct changes done to the source code are not overwritten during next auto-generation, the Java annotations for the functions modified were changed from “@ generated” to “@ generated NOT”.

103
Figure 6.6: The Mapping definition of the Conceptual model in CIM
6.4 The Model Manager GUI

After running the final code that is auto-generated and hand-modified, a graphical editor for the Model manager is achieved. This code is executed as a plug-in in Eclipse. The resulting GUI has an appearance as shown in Figure 6.7 and it consists of 5 distinctly visible regions:

1. Menu and Tool bar
2. Canvas
3. Tool Palette
4. Outline pane
5. Property pane

These regions are described in detail as follows:

- **Menu and Toolbar**: The menu and toolbar appear at the top of the Model manager GUI. The menu consists of five menus: File, Edit, Diagram, Window and Help. The File menu allows a user to create new diagrams, open existing ones, close diagrams and exit the application. The Edit menu performs the usual editing function of copy, paste, select etc. The Diagram menu enables decorative changes to the diagram like changing the width of lines or borders, text alignment, fonts, etc. A new GUI window can be opened from the Window menu. A list of shortcuts can be accessed through the Help menu. The tool bar makes the frequently used menu items easier to access.

- **Canvas**: The canvas is the pane where a user designs the conceptual model using tools from the palette. It is the blank space on the left side of the palette. Right-clicking on the canvas produces a context menu which, among other features, allows a user to save the diagram in different image formats including PDF through the File button.

- **Tool palette**: The tool palette, located at the center of the GUI, is divided into four groups each containing specific kinds of tools. The compartment
group has two containers: CVL and SVL. CVL Nodes and SVL Nodes are nodes that can be drawn only within CVL and SVL containers respectively. The last tool group contains connection tools: Link, PCRelationship and Mapping connections. The Link connection is a generic connection tool; whereas, PCRelationship and Mapping connections allow connections between a particular pair of nodes. A further explanation of these connection tools are explained in Section 6.5.1 for the Link Tool, Section 6.5.2 for the PC Relationship tool and Section 6.5.3 for Mapping tool.

- **Outline pane:** The Outline pane, located in the top right corner, provides the zoomed-in version of the diagram on the Canvas such that a user can directly point to any portion of the diagram in the outline pane and work on it on the Canvas.

- **Property pane:** Almost all the node types in the diagram come equipped with a set of properties which can be accessed using the property pane. The property pane is located in the bottom right corner of the GUI. Some properties can only be changed from the property pane like cardinalities of the parent and child levels in the parent-child relationship and the Primary key attribute of a Property in a CVL model, and a Column in a SVL model. The changes are reflected visually on the canvas.
Figure 6.7: The GUI of the Model manager
6.5  Designing Conceptual Model in the Model Manager

The steps to draw or design conceptual model in the Model manager are as follows:

1. Launch the Model manager.
3. Draw a CVL compartment before you can draw any CVL entities like levels, properties, dimensions, etc. The CVL entities can only be drawn inside a CVL compartment.
4. Similarly, draw a SVL compartment before you draw any SVL entities, namely Tables and Columns. A Table must be drawn first and then columns are dropped inside the table.
5. Draw mappings from Properties in the CVL model to Columns in the SVL model.
6. Save the diagram.

6.5.1  Link Tool

Link is a very generic tool which can be used to connect various combinations of entities like a LevelProperty to a Level, a FactRelationship to a Dimension, a Dimension to a Hierarchy, etc. But for a particular combination of entity types, a link can be drawn in only one direction. For example, a link can only be drawn from a LevelProperty to a Level but not the other way around. The list below shows the direction in which entities can be connected:

- LevelProperty → Level
- FactProperty → Factrelationship
- Measure → Factrelationship
- Factrelationship → Dimension
• Dimension → Hierarchy
• Dimension → Level
• Hierarchy → Level
• Hierarchy → Hierarchy
• Level → Split
• Join → Level
• Split → Split
• Join → Join
• Column → Column (Foreign key to Primary key)

Basically the links are created in the direction of the intuitive flow of references.

6.5.2 PC Relationship Tool

We have separate tool to draw a parent-child relationship between two levels and between one level and one disjoint symbol. The parent-child relationship must be drawn from a parent level to a child level. A parent-child relationship has two cardinalities: a parent cardinality and a child cardinality.

• Parent cardinality (x) : 1 parent member can connect to x child members.
• Child cardinality (y) : 1 child member can connect to y child members.

The parent and child cardinalities are assigned using the Properties pane. The change in the Property pane is reflected in the diagram by a change of graphical representation of the parent-child relationship.

6.5.3 Mapping Tool

A mapping must be drawn from a LevelProperty in a CVL model to a Column in a SVL model. A predicate will be stored as a string.
6.6 Sample Diagram Drawn in the Model Manager

A subset of the Product purchase example from Section 5.5 is shown in Figure 6.8. Here, we draw only the fact relationship Purchase and the Location dimension. The CVL components are drawn within the CVL compartment. In order to show the mapping capability, we also draw the SVL compartment and the schema of fictitious relational database consisting of tables Country, StateProvince, Branch and Purchase. The tables contain columns within them; and the arrows between the columns show foreign key dependencies between the tables. The blue lines cross-cutting the CVL model and the SVL model are the mappings between the properties from the CVL model and the columns from the SVL model. Some of the mappings include predicates in them as shown by texts attached to the mappings.
Figure 6.8: Subset of the Purchase Product example
6.7 Output Files of the Model Manager

As stated earlier, two output XML files are generated for each conceptual model drawn in the Model manager. There two files are semantic model file (*.cim) and diagram file (*.cim_diagram). The semantic model file stores the entities from the conceptual language in CIM that were drawn on the canvas of the Model manager. Here, we show in Listing 6.1 the semantic model file for the diagram depicted in Figure 6.8.

In the listing, notice that names of the domain model relationships appear as tags/elements of the document and the aggregation relationship\(^2\) leads to nesting of tags. The attributes of the domain model entities form the attributes of the tags. The association relationship in the diagram results in the cross-referencing. The GMF uses an order-based path language to reference other nodes in the XML. For example, in Listing 6.1, the dimension Location references the hierarchy Geography, which is identified with path “//@cvl/@hierarchy.0”. This path translates to: start from the root(//) to the cvl tag (@cvl) and finally to hierarchy (@hierarchy) that is defined first in the hierarchy list (.0). The path language uses zero-indexing (i.e., counting starts with 0). Similarly, the column named SPID in the table Branch of the SVL model refers to column Name of table StateProvince with the path “//@svl//@table.3//@column.0” (i.e., first column of the fourth table inside tag svl).

The root of the *.cim document is equivalent to CIM class from the domain model. The CIM class contains three classes (CVL, SVL and Mapping) through three aggregation relationships: cvl, svl and mapping. These form three tags cvl, svl and mapping in the *.cim document. Similar arguments hold for other tags and their attributes.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<modelManager:CIM xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:modelManager="http://modelManager">
    <cvl>
        <level name="Branch"/>
        <level name="Province"/>
    </cvl>
</modelManager:CIM>
```

\(^2\)Here, aggregation means the software engineering aggregation (composition), NOT the OLAP aggregation.
<level name="State"/>
<level name="Country"/>
<dimension name="Location" hierarchyOrLevel="/@cvl/@hierarchy.0"/>
<factrelationship name="Purchase" dimension="/@cvl/@dimension.0"/>
<property xsi:type="modelManager:FactProperty"
    name="TransactionID"
    factrelationship="/@cvl/@factrelationship.0"/>
<property xsi:type="modelManager:Measure"
    name="Qty"
    factrelationship="/@cvl/@factrelationship.0"/>
<property xsi:type="modelManager:LevelProperty"
    name="BranchName"
    key="true"
    level="/@cvl/@level.0"/>
<property xsi:type="modelManager:LevelProperty"
    name="Manager"
    level="/@cvl/@level.0"/>
<property xsi:type="modelManager:LevelProperty"
    name="ProvinceName"
    key="true"
    level="/@cvl/@level.1"/>
<property xsi:type="modelManager:LevelProperty"
    name="StateName"
    key="true"
    level="/@cvl/@level.2"/>
<property xsi:type="modelManager:LevelProperty"
    name="CountryName"
    key="true"
    level="/@cvl/@level.3"/>
<disjoint xsi:type="modelManager:Split"/>
<disjoint xsi:type="modelManager:Join" joinTo="/@cvl/@level.3"/>
<hierarchy name="Geography"
    connectTo="/@cvl/@level.0"/>
<hierarchy splitfrom="/@cvl/@disjoint.0"
    name="Geography"
    connectTo="/@cvl/@level.0"/>
<pcrelationship ParentCardinality="1...*"
    ChildCardinality="0...1"
    parent="/@cvl/@level.1"
    child="/@cvl/@disjoint.0"/>

113
<pcrelationship ParentCardinality="1...*"
    ChildCardinality="0...1"
    parent="//@cvl/@level.2"
    child="//@cvl/@disjoint.0"/>
<pcrelationship ParentCardinality="0...*"
    ChildCardinality="1...1"
    parent="//@cvl/@disjoint.1"
    child="//@cvl/@level.1"/>
<pcrelationship ParentCardinality="0...*"
    ChildCardinality="1...1"
    parent="//@cvl/@disjoint.1"
    child="//@cvl/@level.1"/>
</cvl>
<svl>
<table name="Country">
    <column name="Name" key="true"/>
    <column name="Continent"/>
</table>
<table name="Purchase">
    <column name="TranID"/>
    <column name="Qty"/>
    <column name="BranchID" FKreference="//@svl/@table.2/@column.0"/>
</table>
<table name="Branch">
    <column name="name" key="true"/>
    <column name="Officer"/>
    <column name="SPID" FKreference="//@svl/@table.3/@column.0"/>
</table>
<table name="StateProvince">
    <column name="Name" key="true"/>
    <column name="IsProvince"/>
    <column name="CountryID" FKreference="//@svl/@table.0/@column.0"/>
</table>
</svl>
<mapping column="//@svl/@table.0/@column.0"
    property="//@cvl/@property.6"
    Predicate="Continent=&quot;N. America&quot;"/>
<mapping column="//@svl/@table.1/@column.0"
    property="//@cvl/@property.0"/>
<mapping column="//@svl/@table.1/@column.1"}
The diagram file (*.cim_diagram) stores the details about graphical decorations applied to the entities from the semantic model file (*.cim). The semantic model file contains the CVL, SVL and MVL models inside it, nested under the cvl, svl and mvl tags. So, this file becomes an input to the transformation engine of the Model manager. The transformation engine derives the CDL and MDL models from the CVL and the MVL models, respectively.

Listing 6.1: PurchaseProduct.cim
Chapter 7

Conclusion and Future Work

This thesis is a step forward to the bigger goal of what CIM endeavors. The conceptual modeling language introduced in CIM was studied in detail in this thesis. Rather than making an arbitrary observation and empirical claims, we provided a mathematical basis to the model to enable a scientific and rigorous study of them. This helped to precisely define what a conceptual model actually means rather than leaving it up to the user to interpret it arbitrarily. We provided such mathematical definitions of the hierarchies and dimensions, in particular, in Chapter 4 to dispel any ambiguities introduced by ad hoc interpretations of them. Lots of examples were used to show the real world scenarios that can be easily captured using our conceptual modeling language. In Chapter 5, the formal semantics of the language was provided which firmly places our CVL into the mathematical realm. This leaves no ambiguity toward the syntax and meaning of our language. Having a formally defined language opens the door to a well defined query language to be developed over it. A business user familiar with the language will never have to second guess what data is available in the conceptual model, and in what form, before formulating a query.

We not only defined the conceptual language but also developed a prototype graphical editor based on that language. This graphical editor is a significant part of the Model manager in CIM and enables even technically-challenged user to articulate their business requirements through visual diagrams. This graphical editor has a familiar interface and hence an easier learning curve than drawing diagrams.
by hand. The use of an open source framework to develop such a graphical editor implies a robust architecture and potentially a lot of support from the developer community for maintenance.

Specifically, we made the following contributions through our thesis:

- We provided formal semantics of the conceptual visual language in CIM, planting it firmly in the mathematical model in Chapter 5.

- We presented precise mathematical definitions of the more complicated structures such as hierarchy and dimensions in Chapter 4.

- A fully functional prototype of the graphical editor in the Model manager was developed that allows users to draw the conceptual models described in this thesis.

- Documentation that walks the users through the development process. The usage scenario for the graphical editor was supplied in Chapter 6.

As future work, we envision to add the model transformation capability to the Model manager to allow CVM to CDM transformation and vice-versa. This is the goal of the transformation engine in the Model manager. But before that, the schema of the CDM needs to be defined based on the semantics of CVL. Once this is achieved, the transformation process should not be very complicated. Also, the runtime API provided by the GMF can load the output XML file of the CVL model into memory as an XML DOM structure. This DOM model can be leveraged to drive the transformation process. This transformation engine completes the functional objective of the Model manager in CIM project.
Bibliography


http://gerardnico.com/wiki/_media/database/oracle/oracle_olap_aw_cube.jpg → pages viii, 8

http://www.eclipse.org/ → pages 93

http://www.eclipse.org/forums/index.php?i=thread&frm_id=16&S=2fa72f5b9a34b09ca4818a3bb27014b2 → pages 103


[7] GMF samples and tutorials overview. 


