GraMMi: Using a Standard Repository Management System to Build a Generic Graphical Modeling Tool

Carsten Sapia, Markus Blaschka, Gabriele Höfling
FORWISS (Bavarian Research Center for Knowledge-Based Systems)
Orleansstr. 34, D-81667 Munich, Germany
Email: {sapia, blaschka, hoefling}@forwiss.tu-muenchen.de

Abstract

Commercial off-the-shelf repository management systems claim to provide a solid basis for building CASE tools by offering special meta-data management facilities (like versioning, configuration management, long transactions support). Another important area of application for these systems is the management of meta-data in data warehouse environments. The ISO-IRDS standard defines a framework for the modeling of repository data. This paper describes a case study that implements a generic graphical modeling tool (GraMMi) on top of such a repository managment system using the IRDS standard to guide the design process. To show the feasibility of our approach, we present a case study where GraMMi is used for the conceptual design of data warehouse models based on a specific modeling technique. The paper describes the overall system architecture, the object-oriented design (including the repository schema) and the user interface design. Furthermore, we present experiences from the implementation of the system using an industry-standard platform (Softlab Enabler as the repository system and Visual C++ for implementing the modeling tool).

1 Introduction

Data warehouse systems integrate data from heterogeneous sources throughout an enterprise and restructure them according to the needs of analysts ([3]). The design of such a system and the involved data transformation process is a complex and cost intensive task. Specialized methodologies supporting the design of this process are currently emerging in the research area (e.g. [15]). Consequently, graphical design tools supporting these methodologies have to be built.

The Meta-CASE approach (e.g. [1], [7], [13]) is a popular way of building design tools by configuring generic components. One of the core components of such a CASE shells is a generic graphical modeling tool. It can be configured with a formal description of the modeling method (including the graphical representation and the syntactical constraints). Though following different approaches concerning architecture and meta-modeling formalisms (see section 3) all of these tools use a mostly custom-built repository component to store the models created by the CASE tool user and mostly the information about the method itself.

This is another interesting link to the application area of data warehouse (DW) design. A lot of process oriented and business oriented meta-data is managed in DW environments. Advanced systems use commercial off-the-shelf (COTS) repository management systems to manage the meta-data that drives the DW process. In recent years, such systems have gained commercial impact (e.g. Microsoft Repository, Softlab Enabler, unisys UREP). They offer database capabilities (transactions, backup and recovery) together with specialized meta-data functionality (like support for versioning, configuration management, active mechanisms etc.).

Therefore, it seems feasible to base a generic graphical design tool on a standard repository system. From our pragmatic viewpoint as designers of data warehoue notations this ensures a seemless integration into these environments. Besides, this allows us to benefit from the advanced functionalities offered by the repository system.

Following these considerations, we performed the GraMMi (**gr**aphical **meta-data**-driven **m**odel**i**ng tool) case study. The pragmatic objective of this experiment was to develop a Meta-Editor based on a standard off-the-shelf repository management system. Nevertheless, the results are interesting to a broader audience because we used the ISO-IRDS standard to guide the design and implementation process. Our case study demonstrates how the IRDS framework for modeling repository systems can be used to drive the design of Meta-CASE tools and other meta-data driven applications. As GraMMi is a typical example for such a system, it allows study of the important issues of the development methodology for such a tool (design of the meta-model layers, choice of a repository system, implementation of the data model using the repository and building a self configuring user interface).

The following section sketches our project environment in more detail. Section 3 gives a short overview of the related work we found useful in building the GraMMi system. The remainder of the paper from there is structured according to the classical software development process: we start by compiling the requirements (section 4)

Authorized licensed use limited to: Bibl Univ Moretsu Plantin. Downloaded on March 29,2022 at 14:36:31 UTC from IEEE Xplore. Restrictions apply.

that drove the design of GraMMi and show how the repository system helps to fulfill them. Section 5, the core of this paper, covers the GraMMi design phase focusing on the IRDS driven meta-modeling (section 5.1), the design of the repository schema (section 5.2) and the client application (section 5.3). An overview of the implementation using the Softlab Enabler Repository ([17]) and the Visual C++ development environment is given in section 6. Section 7 demonstrates the configuration process for the GraMMi tool using a datawarehouse modeling notation as an example. We conclude by describing some experiences we had during design and implementation.

2 Project background

In our research project 'System42', we are developing a method to specify and maintain data warehousing (http://www.forwiss.de/~system42/, [3]) environments. The core of the method is formed by a set of graphical notations ([14], [15]) that consider the specific semantics of the data warehouse design and maintenance cycle. This is further reflected in the overall vision that the warehouse designer works with a conceptual model (e.g. representing the static multidimensional data model using a specialized notation, the ME/R notation [15]) in a graphical design tool. The used notation was developed as part of our research work and we currently evaluate its feasibility in several research and industrial projects. To this end, we use an iterative process to involve end users in the language design. This implies frequent changes to the syntax and semantics of the modeling language. Therefore, we needed a very flexible modeling tool that allows the quick adaptation of the underlying modeling notation, without programming and recompilation.

An open repository management systems is used in advanced data warehouse environment to manage the metadata of the warehouse process ([3]). This means that such a system is already installed and running. Ensuring the best possible integration of the modeling tool into this environment provides a further motivation to build the modeling tool on top of such a system. If all relevant information objects are stored and linked by the same repository management system, consistency can be enfored by this system. This is especially important in environments where a lot of different tools are sharing the same information.

3 Related Work

Many of the topics occurring during the development of GraMMi have been researched in the context of visual languages (e.g. [12]), graphical editors and Meta-CASE (e.g. [4], [6], [8], [13], [16] , [18]) tools. Nevertheless, the emphasis for the GraMMi experiment was on using standard components for the repository management and the user interface and researching the impact of this decision on the development process of a meta-data-driven tool.

GraMMi takes up the basic idea of Meta-CASE tools that a CASE system can be built from method-independent components which are configured to meet the needs of the notation that is to be supported. Rhus, GraMMi approach could be used to build configurable graphical editor components for these environments. The main difference between GraMMi and the other approaches is the repository system (IRDS) driven design and its impact on the description of the notation. Therefore, we compare the GraMMi approach to existing Meta-CASE approaches mainly according to data storage and the graphical description formalism. Summarizing, in contrast to e.g. MetaView ([7]) , MetaEdit+ ([13]) and ToolBuilder ([1]), GraMMi does not use a scripting language to specify the graphical notation and its syntax. According to the repository centered philiosophy, the notation is edited by adding and linking objects in the repository. This approach allows to use the standard components of the repository system to be used for notation design and does not demand the notation designer to learn a high level language.

MetaView (e.g.[7]) is one of the most prominent Meta-CASE approaches. It uses three levels (user level, environment level and meta level) to structure the processes and components used in the CASE tool building process. This structure is somewhat similar to the layers two, three and four of the ISO-IRDS structure. But this structure is not represented in the schema of the software repository. A proprietary component is used for data storage. The tool that is functionally comparable to GraMMi is the Metaview Graphical Editor (MGED) which uses the EARA/GE (Entity, Aggregate, Relationship, Attribute with Graphical Extensions) Model to specify the notations.

MetaEdit+ (e.g. [13]) is a commercial tool developed by MetaCASE Consulting. MetaEdit+ uses a proprietary ObjectRepository as the core of its architecture. Similar to our approach all meta information is stored there only once ensuring consistency. Our approach differs in using an open repository system to implement this component. This makes the necessary integration into an existing environment easier (see section 2).

According to the philosophy of Meta-CASE tools, *ToolBuilder* [1] provides a set of generic tools and function libraries. The component that is equivalent to GraMMi is the Design Editor. It supports the creation of directed graphs with labeled nodes and links. The definition of the syntax is given using a Graph Definition Language (GDL).

The lately proposed *MetaMOOSE* system ([8]) is an object oriented framework approach using an interpreted OO language. Its novelty is the use of an object-oriented approach to build the meta-model. From all the Meta-CASE tools presented here, MetaMOOSE is most closely related to the concepts of GraMMi. We also use an object model to describe the entities of our notation. Another similarity is that MetaMOOSE uses a persistent object database for data and meta-data storage which offers some

Authorized licensed use limited to: Bibl Univ Moretsu Plantin. Downloaded on March 29,2022 at 14:36:31 UTC from IEEE Xplore. Restrictions apply.

of the functionality of a repository management system. Albeit, using a standard repository management system, we gain additional functionality besides the persistence (e.g. versioning capabilities).

4 Analysis of Requirements and Objectives

To better understand the design of the GraMMi system let us first take a look at the requirements that drove the design and implementation. The following functionality should be available in GraMMi:

- **Standard graphical modeling features:** From the end users point of view, GraMMi is a standard graphical modeling tool which can be used to build and revise models, e.g. ME/R models for warehouse design. As the frontend is not our primary focus we imitate commercial tools like Rational Rose [11] or E/RWin.
- **Integration into CASE environment:** The CASE process usually involves a set of tools from different vendors with different functionality. If a standard repository system is used as basis for all components, the integration on this level is enabled [2].
- **Full extensibility of graphical notations:** For an administrator, changing and extending the graphical modeling notation should be possible without changing (e.g. recompiling) the GraMMi system. This involves changes to the graphical representation of elements, adding new modeling elements to the notation, introducing or relaxing syntactical constraints. All these task should be possible with the repository administration tools (adding and linking objects).
- **Model Checking facilities:** The tool should be able to check a model for its syntactical correctness, before checking it back to the repository. For this task, it needs a formal description of the notational syntax.
- **Single point of control.** In large companies several modelers use the same notations (maybe even concurrently editing the same models). Changes to the notation should only be necessary once and should then be immediately visible in all modeling environments. As the definition of the notation is stored in the central repository, active mechanisms of the repository system can be used to implement this.
- **Versioning and Configuration mechanisms.** Models and modeling notations have to be versioned. When a new version of a modeling notation is created, models that have already been built do not correspond to the latest specification. Therefore, older versions of the modeling notations have to be maintained by the system automatically. A modeling methodology (like UML) includes several notations (e.g. class diagram, interaction diagram) that can itself have different versions. Thus, the system must be able to automatically maintain configurations (a set of objects across version space). A repository management system offers facilities for version and configuration management.

Starting from these requirements and rationales about good software design, the following technical objectives can be defined:

- **Use of 'COTS' components:** The main objective is to use a COTS (commercial off-the-shelf) repository system and use as much of the offered functionality as possible. But standard components can also be used for the user interface (graph layout and editing components) and the syntax checker (a graph grammar parser). The overall architecture must support such a modular approach.
- **Fully Meta-data-driven tool:** Changes to the modeling notation should be possible without changing the GraMMi program code. This means that the description of the modeling notation has to be modeled as meta-data and the tool must read this meta-data at runtime and configure the user interface and the consistency checks accordingly. Section 5.1 describes the design of the meta-data in more detail.
- **Independence of the specific repository product:** Two rationales imply that is a good idea to keep large portions of the system independent from the actual repository system used:
  - First, a repository system usually is the common data storage for a wide range of tools and the actual system used in a company is usually a strategic decision. Thus, to be able to integrate seamlessly into an existing environment it is necessary to support as many repository systems as possible. This means that the effort needed to port the system to a new repository system should be kept to a minimum.
  - Second, the repository technology is still evolving. New product versions contain significant new features. This means that the choice of the repository product should be revisable after a new release.

5 GraMMi Design

This section gives an overview of the GraMMi design process. The most important issues of the design were the modeling of the meta-data, the design of a repository schema and the design of the GraMMi application. We used the ISO-IRDS framework to start the meta-data modeling (section 5.1). The mapping process and refinement process that transforms the IRDS framework to an object-oriented repository schema is described in section 5.2. The object-oriented design of the GraMMi application itself is detailed in section 5.3.

5.1 Modeling meta-data using the IRDS framework

The IRDS standard ([10]) proposes a reference model for the design of repositories using four different layers of abstraction, where each layer x defines the schema for the layer x+1 (see Figure 1). We show how the IRDS reference model can be used to structure the problem of designing a generic modeling tool (as described in the previ-

Authorized licensed use limited to: Bibl Univ Moretsu Plantin. Downloaded on March 29,2022 at 14:36:31 UTC from IEEE Xplore. Restrictions apply.

ous section). When transferring this generic schema to an actual problem domain it is important to define what contents are represented on which level and which formalism should be used to model the different layers. When using a COTS repository system, the top layer (Layer 1) is determined by the modeling elements offered by the repository system (Softlab Enabler in our case).

We used a bottom-up approach starting with layer 4, which contains the graphical model the user builds with GraMMi.

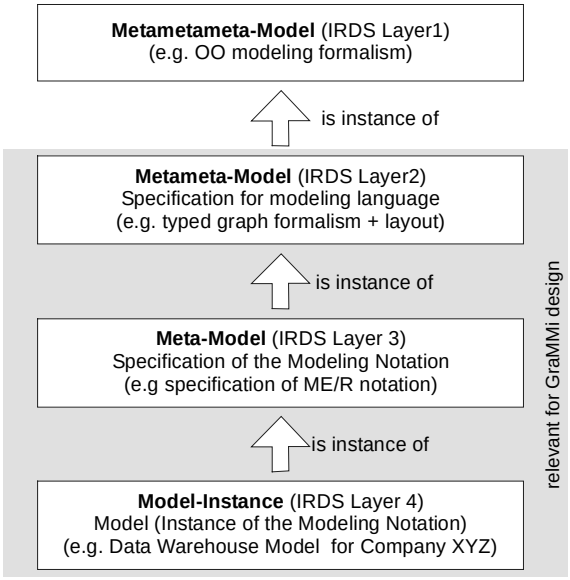

Figure 1: the IRDS layers revisited in the context of a generic graphical modeling tool

5.1.1 Model Layer (IRDS Layer 4)

This layer contains the actual model instance that is interactively manipulated by the GraMMi user during a design session (e.g. a class model of the accounting system modeled using the UML class model notation). Mathematically speaking a GraMMi model is represented as the instance of a typed graph.

Definition (typed graph): A typed graph over a set of edge types Σ_E and a set of node types Σ_N is defined as a tuple (N, E, t_N, t_E, s, t) with

- N is a finite set of nodes,
- E is a finite set of edges,
- t_N: N→Σ_N assigns each node to its node type
- t_E: E→Σ_E assigns each edge to its edge type
- s, t: E→N assigns each edge to its source and target ♦

The contents of the model-instance layer contains the values for N, E, t_N, t_E,s and t (e.g. the objects account and customer). Nevertheless, the definition of the edge and node types Σ_E and Σ_N (e.g. objects, relationships, packages) are part of the **modeling notation** which is described on the next layer**.**

5.1.2 Meta-Model Layer (IRDS Layer 3)

The meta-model layer contains the schema definition for the model layer. Therefore, it must contain all the information needed by GraMMi to automatically configure itself for a notation. It contains objects describing:

- the definition of the elements of a modeling method (Σ_E and Σ_N) (e.g. objects, relationships, packages),
- the syntactical constrains that have to be fulfilled by the model (e.g. that some node types may only be connected by a special edge type) and
- the corresponding graphical representation information (e.g. that the node type 'class' is being visualized using a rectangle shape). Alternative graphical representations may exist for the same meta-model.

As models are represented in GraMMi as typed graphs (see section 3.1), we use a graph grammar ([5]) to describe the syntactical constraints of the modeling notation.

Definition (graph grammar): A graph grammar over a set of edge types Σ_E and a set of node types Σ_N is defined as a tuple (d_0, P) with

- d_0 is a nonempty initial typed graph over (Σ_E, Σ_N) called the axiom.
- P is a finite set of productions. Each production p is of the form L →R where L and R are typed graphs over (Σ_E, Σ_N) with L being the left hand side and R being the right hand side. ♦

The replacement of nonterminals in graphs is far more complicated than in linear texts. Therefore, different embedding strategies have been proposed to solve this problem ([5]). For GraMMi, we use the concept of contexts. This means, that both sides of the production contain a common context graph that allows for defining which part of the existing graph the new elements should be connected to.

The parsing problem for context sensitive graph grammars is in general intractable. Therefore, our implementation is restricted to layered graph grammars as presented in [12] to allow parsing without restricting the expressiveness to context-free grammars.

The description of the modeling notation (i.e. the edge and node types with their graphical representation and the corresponding graph grammar) is defined by the architect of the notation resp. the GraMMi administrator. It is read by GraMMi and used to configure the interface and the syntax check. The structure for this definition is the contents of the next layer.

5.1.3 Metameta-Model Layer (IRDS Layer 2)

The metameta-model layer constitutes the schema for the definitions of the modeling notation. Thus, it contains the notation independent part of the modeling tool. It contains the structural definition for the edge and node types, the graph grammar (e.g. graph grammar contains an axiom and a set of productions) and the generic graphical repre-

Authorized licensed use limited to: Bibl Univ Moretsu Plantin. Downloaded on March 29,2022 at 14:36:31 UTC from IEEE Xplore. Restrictions apply.

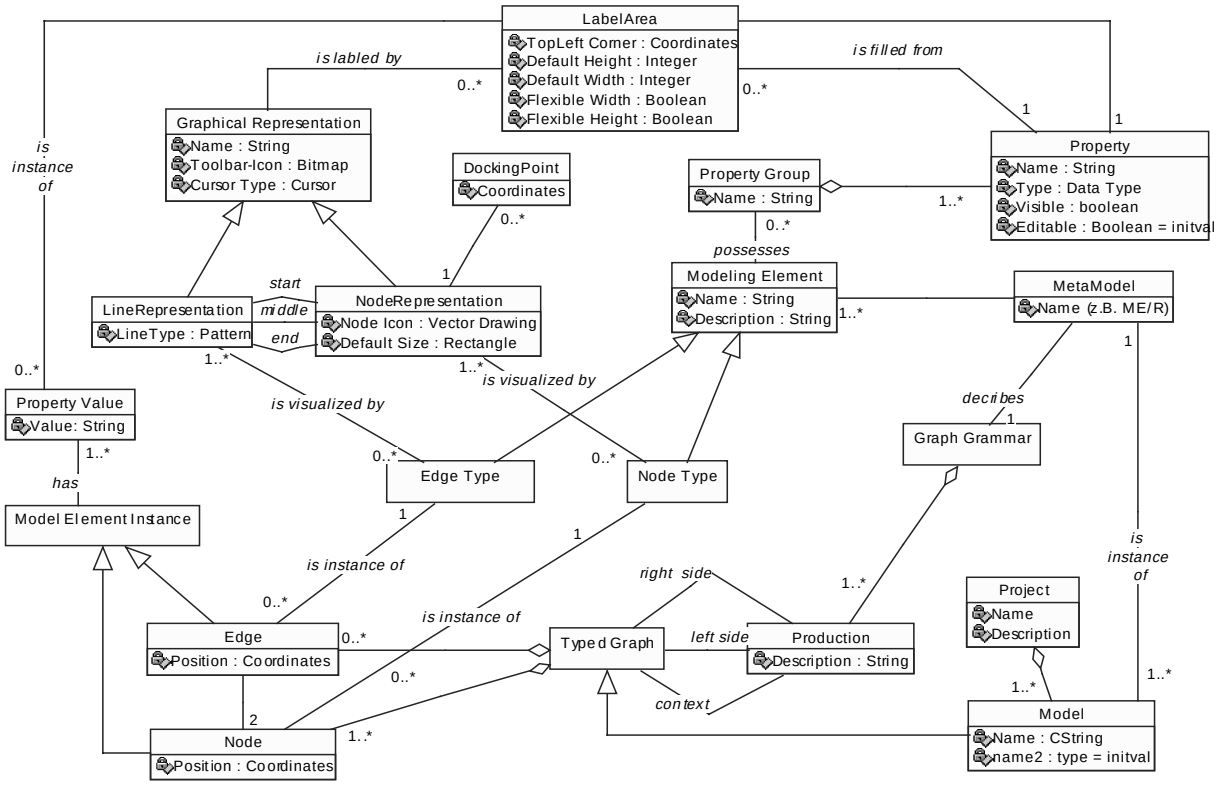

Figure 2: The object-oriented model of the repository schema

sentations. The metameta-model is hard-wired into the GraMMi application and will be further elaborated in the following sections.

Summing up, the following table gives an overview of the different meta-data layers, who designs them and how they are stored in GraMMi.

Layer	Designer	Storage
Metameta-Model Layer	GraMMi Designer	hard coded in Repository (schema) and in GraMMi Code (C++ class definitions)
Meta-Model Layer	Notation Designer	Stored in Repository as objects
Model Layer	(Business-) Modeler	Stored in Repository as objects

Table 1: Different layers and their handling

5.2 Designing the repository schema

The parts of the IRDS model that are relevant to GraMMi are distributed across three different layers of abstraction. That means that two instantiation steps are necessary (see Figure 1): one from the metameta-model to the meta-model and another one from the meta-model to the model. This concept (inherent to Meta-CASE tools) is not supported by common database systems and standard repositories. These systems support only one level of abstraction by allowing the user to define a schema (object-oriented or relational) and providing the facilities to create instance of this schema (i.e. create objects or tuples). Therefore, the logical design of the GraMMi applications involves the mapping of the three layered concepts (which is useful for understanding the meta-data-driven approach) to a two layered model (which is useful for the implementation).

An object-oriented model for the GraMMi Repository is shown in Figure 2. GraMMi stores several *Meta-models* (see shaded area, e.g. a specification of the ME/R model or the UML class diagram notation). Each of those models contains a set of valid *modeling elements* (e.g. classes, isa-relationships for the UML class diagram). A modeling element is either an *edge type* or a *node type.* In any case each modeling element can possess a set of *properties* which can be grouped to *property groups*. E.g. the modeling element class (of the meta-model UML) may own general properties like name and description and implementation-specific attributes like ‘transient/persistent’ or the modeling element relationship can possess properties like ‘multiplicity’ or ‘role name’.

To clarify this concept, Figure 3 shows an example of how this information is used by GraMMi to configure its interface at runtime. Let us assume that the administrator defined a modeling element (e.g. a dimension level, see section 7) that possess two property groups (“General” and “Code Generation”). Furthermore, the first group may

Authorized licensed use limited to: Bibl Univ Moretsu Plantin. Downloaded on March 29,2022 at 14:36:31 UTC from IEEE Xplore. Restrictions apply.

contain the three properties *"Name"*, *"Description"* and *"Unique"*. In this case, if a user inspects the properties of a modeling element instance, the dialog shown in Figure 3 is presented to the user.

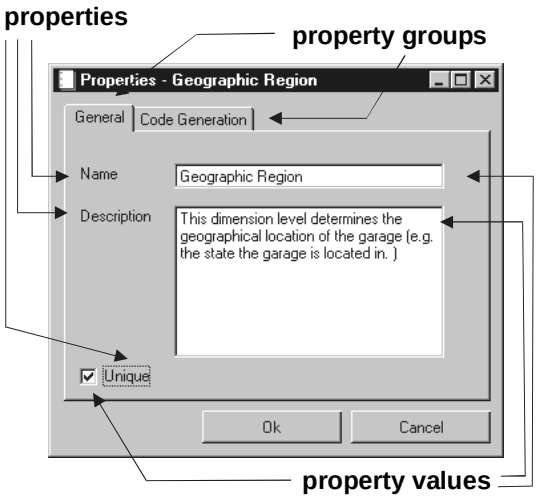

Figure 3: property groups and properties are used to define the structure of the user interface

The class *graphical representation* determines how a modeling element is represented in the diagram and how it is laid out on screen. The attribute *toolbar icon* determines how the modeling element is represented on the toolbar of the modeling tool (e.g. a small bitmap) while *cursor form* defines how the cursor looks when the user places the object. Depending on the type of the elements (node or edge), additional information is needed. This information is contained in the specialized classes *node representation* and *edge representation*.

The appearance of a node in a diagram is determined by the attribute *node icon*. It contains a vector drawing of the graphical representation (e.g. a rectangle or a cloud shape). Each node representation is further characterized by a set of *docking points* which contain the coordinates of the shape, where edges may start.

For an edge a *line type* is given which is a graphical pattern that is used to draw the line (e.g a dotted line). Furthermore, for each line the appearance of its beginning, end and middle is determined by a node representation.

Irrespective of the type, elements are labeled using the value of a property (e.g. classes are labeled with their name). The class *label area* determines the coordinates relatively to the graphical representation, where the label is placed. This area can be defined as flexible, i.e. that the graphical representation may be scaled if the label does not fit into the element. A graphical element may possess an arbitrary number of *labeling areas* (e.g. a class in UML notation contains a labeling areas for its name, the attributes and the methods).

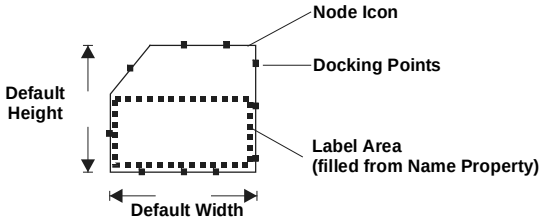

Figure 4: definition of a node representation

To illustrate the GraMMi concept of defining a graphical representation, Figure 4 contains a sample definition of a node representation (for a box with a cut edge that is described by its name). Figure 5 shows a sample line definition (for a line with different arrows at its end points and an oval in the middle).

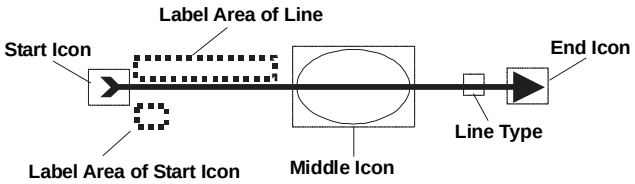

Figure 5: definition of a line representation

To describe the syntactical integrity constraints of a *meta-model*, a *graph grammar* is used. It consists of a set of productions that in turn consist of a right hand side, left hand side and a context. All of these are *typed graphs* over the type definition given by the modeling elements that describe the meta-model.

All the classes described so far are part of the metameta-model (IRDS layer 2). The classes of the meta-model are instances of these classes. Because of the limitation to two abstraction levels, we have to model this instantiation relationship explicitly.

A *model* (i.e. the data warehouse model of a specific company) is an instantiation of a meta-model (e.g. the ME/R model). A model is a typed graph consisting of edges and nodes. Each of these is assigned a type (edge type or node type). This type definition has to be associated with the meta-model of the model. For example a model for the ME/R notation may only contain edge and node types that are defined within the ME/R meta-model. Edges and nodes of the model are instances of a model element. An instance assigns a property value to each property of its corresponding model element.

To enable the use of several modeling notations in a project (e.g. the UML contains use cases, class diagrams, interaction diagrams etc.) several models can be grouped into a project.

5.3 Application Design

The first step in the design of GraMMi was to develop a layered architecture defining a functional decomposition of the system. This was followed by an object oriented design of each layer and the interface between the layers.

Authorized licensed use limited to: Bibl Univ Moretsu Plantin. Downloaded on March 29,2022 at 14:36:31 UTC from IEEE Xplore. Restrictions apply.

This section presents the different layers and explains the benefits of the decomposition.

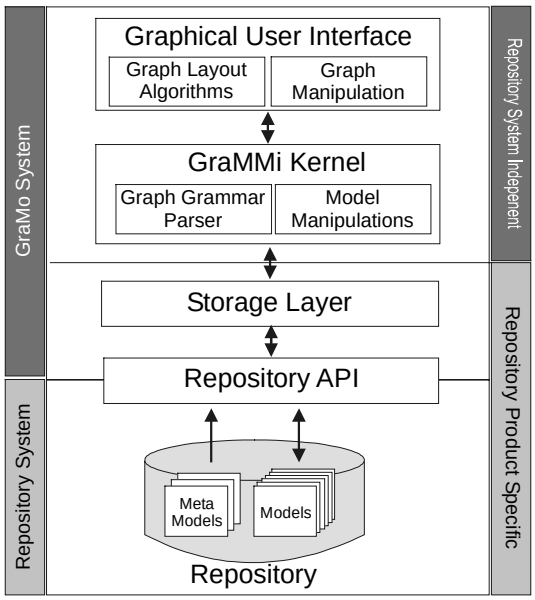

Figure 6: Architecture of the GraMMi System

5.3.1 Layered System Architecture

We functionally decompose the system into independent software layers that use the functionality of the underlying layers. Beside defining a clear starting point for the following OO design, the layered architecture enables a modular design and implementation. The layers are shown in Figure 6.

The **repository** persistently stores the meta-models (definitions of the notation) and the models (instances of the notation). It is a part of a specific repository system (e.g. Softlab Enabler). Access to the repository is performed via an **API interface** that is provided by the repository system. This interface is typically product specific. In order to keep the dependence of the GraMMi system on the actual details of this interface to a minimum, we introduce the **storage layer**. It provides a repository independent interface to the upper layers of the system. This interface contains methods like *LoadMeta-model* or *StoreModel.* None of the objects passed through this interface has methods that contain calls to the repository API. This means that the storage layer has to map these system independent methods to specific API calls of the repository product used. In case this product is changed, only this layer has to be ported.

The **GraMMi kernel** manages and manipulates the persistent objects that are needed during the session of the current user. It uses the interface of the Storage Manager to create (load) and store the objects. The kernel's task is to convert these objects (the models) into a form that can be understood by the graphical user interface (in our case typed graphs). It also has to implement the syntactical and semantic checks of the models (using the graph grammar defined in the meta-model). It uses a graph grammar parser component for this purpose.

The **graphical user interface** is a generic graph editor that knows nothing about the semantics attached to the graphs (e.g. the syntactical constraints). It contains functionality to visualize graphs, automatically layout graphs and allow graph manipulation functions (like insert edge, insert node). These manipulations are passed to the kernel for validation and execution. The strict separation of visualization and manipulations allows to use a generic component as the user interface (which can also be reused in other projects). Furthermore, it is easy to implement the interface between the GUI and the kernel layer as a remote interface (e.g. using RPC calls or Java RMI). This allows for a three tier architecture: A frontend tool e.g. implemented in Java to run on a thin client (e.g. a web browser), an application server (the kernel and the storage layer) and the repository system running on a database server machine.

The next section shows how the general idea of separating visualization and manipulation control is mirrored in the object-oriented design phase.

5.3.2 Designing the kernel and the graphical user interface

The user interface configures itself based on the meta-model read from the repository. We used the model-view-controller pattern (MVC, [9]) to design the user interface. This pattern functionally divides the user interface into three parts (see Figure 7). A model class implements the representation of the data that is to be visualized and manipulated by the tool. In our case the model encapsulates the internal representation of the typed graph data structure that is used to describe a model. This data structure is filled with data from the repository by the GraMMi kernel. The model knows nothing about the interpretation and semantics of the graph nor its visualization.

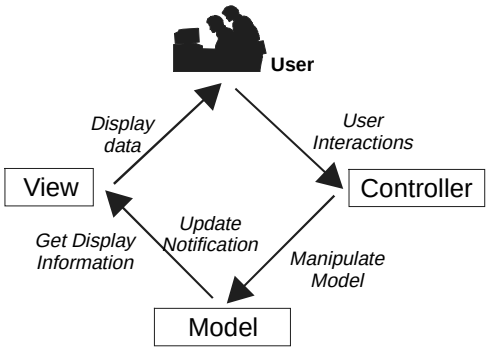

Figure 7: Model-View-Controller (MVC) pattern

Each model can be connected to one or more views that are responsible for the visualization of the data. Each view may offer a different visualization (e.g. showing different parts of the diagram). The view calls methods of the model

Authorized licensed use limited to: Bibl Univ Moretsu Plantin. Downloaded on March 29,2022 at 14:36:31 UTC from IEEE Xplore. Restrictions apply.

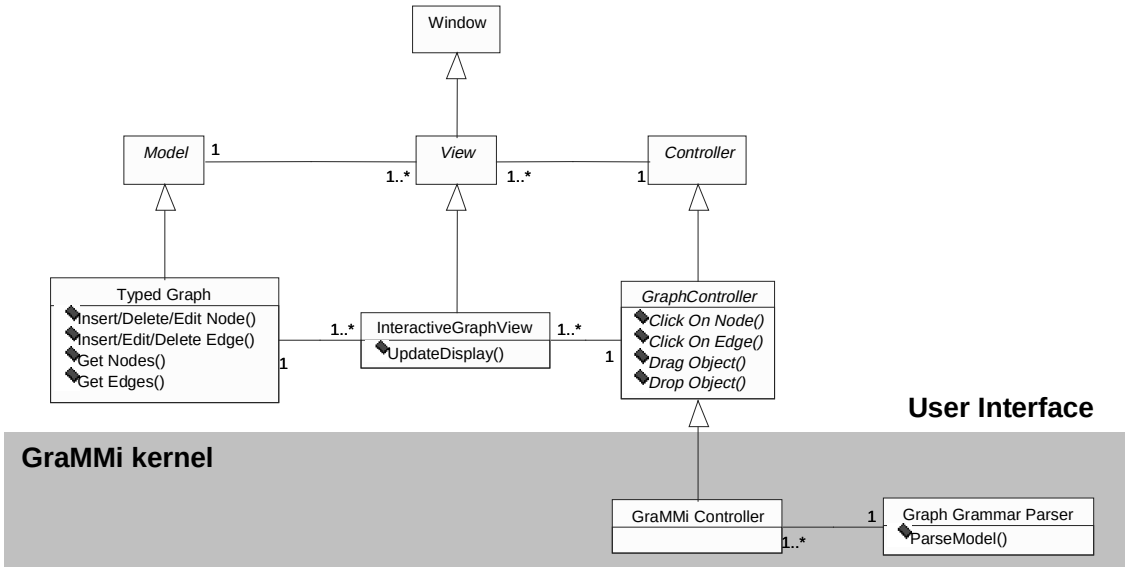

Figure 8: The OO design of the user interface and its interface to the kernel

to get the information needed for the visualization. The view does not know about the internal data structures of the model nor the semantics of the data being displayed.

Knowledge about the semantics is encapsulated in the control class that each view is connected to. This class translates the user interactions and manipulations on the graphic representation to operations on the model. It is also independent of the internal model representation as it uses methods of the model class to carry out manipulations. In our case the controller validates the syntactical correctness of the graph according to the graph grammar defined with the meta-model (specification of the notation). For this purpose it uses a graph grammar parser class.

Figure 8 shows a class diagram of the user interface and is interface to the GraMMi kernel. The classes *GraMMi Controller* and *Graph Grammar Parser* are part of the GraMMi Kernel (shaded gray) while the rest is part of the user interface.

6 GraMMi Implementation

We prototypically implemented GraMMi using the Softlab Enabler Repository System to manage the central repository. The client application was developed using MS Visual C++ (versions 5.0 and 6.0) under Windows NT.

The Enabler repository uses an object oriented paradigm for data modeling but it does not support the concept of inheritance. Apart from this, the model presented in Figure 2 could be easily implemented as the repository schema using the Administration tools. The Enabler Repository System is a full database system in the sense, that it does not make use of another (e.g. relational) database engine to store the meta-data but implements its own storage management. The Enabler functionality (including versioning and configuration management) can be accessed using a C++ class library encapsulating a C-API. According to our design, all the Enabler API calls were encapsulated in a storage layer (see Figure 6) which was implemented as a C++ repository class offering methods to persistently store models and meta-models. To port our system to another repository system only the methods of this class have to be rewritten.

The client implementation uses the Microsoft Foundation Classes (MFC); a framework that provides classes for elements of the graphical user interface (e.g. menus, windows, listboxes) and allows for handling user interaction. The MFC supports a variation of the MVC paradigm (called document-view concept) that was extended in our implementation to fully match the MVC pattern. This enabled the reuse of a generic class library for the drawing, layout and manipulation of graphs. This library contains the classes *InteractiveGraphView*, *GraphControl* and *TypedGraph* (see Figure 8).

7 System Demonstration and Case Study

In this chapter, we demonstrate the functionality of GraMMi and the steps to configure the tool for a given modeling notation using an example. The ME/R model ([15]) is a specialization of the E/R model that can be used to build multidimensional conceptual data models (e.g. need for OLAP and data warehouse applications). It contains three extensions to the classical E/R model representing the special semantics of the multidimensional paradigm: an n-ary fact relationship, a classification relationship and a dimension level entity set. The graphical icons used for these elements are shown in Figure 10.

The first step in configuring GraMMi for the ME/R notation is to define the necessary modeling elements and

Authorized licensed use limited to: Bibl Univ Moretsu Plantin. Downloaded on March 29,2022 at 14:36:31 UTC from IEEE Xplore. Restrictions apply.

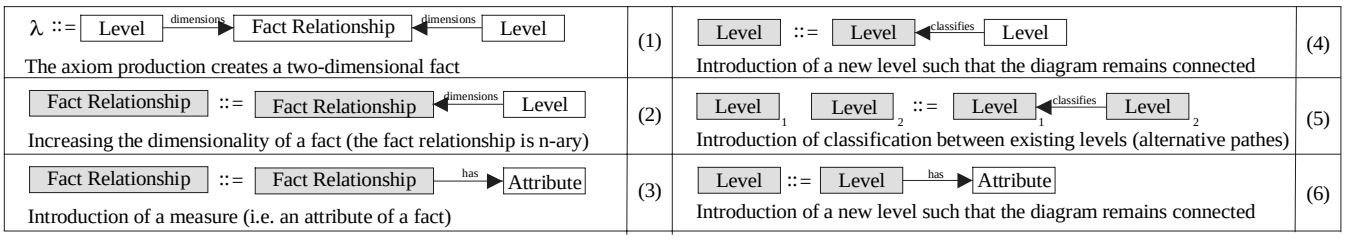

figure 9: A sample graph grammar for the ME/R model

their graphical representation in the repository. We use the standard graphical tool that ships with Enabler to instantiate the classes 'Node Element' and 'Edge Element' accordingly.

Our metameta-model only supports binary relationships between modeling elements. Therefore, the fact relationship has to be expressed by two modeling elements: a fact entity and a binary is-dimensioned-by relationship. Properties can be defined for each modeling element. In our case all the elements possess two properties: a name and a description.

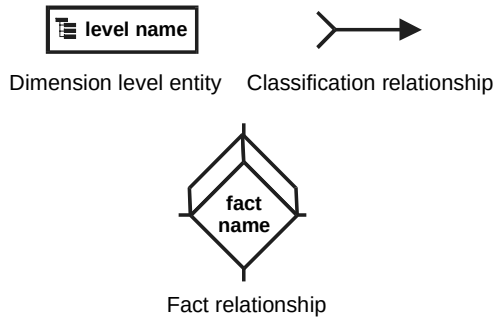

Figure 10: Graphical elements of the ME/R model

After having defined the elements, they have to be given a graphical representation. The toolbar icon, the cursor and the basic display graphic are stored as attributes to the class node graphical representation respectively its subclasses (using the Windows Meta File Format). The next step is the definition of the language syntax. The productions of the graph grammar for the ME/R model are shown in Figure 9.

After these configuration steps, the ME/R notation is available for all clients. When the GraMMi client tool is started, the user is prompted with all the available notations (e.g. UML class diagram, use case diagram, ME/R diagram). If the user chooses the ME/R notation, the toolbox (see Figure 11, left side) is filled accordingly with the toolbar icons of the graphical modeling elements. The main part of the window is filled by an InteractiveGraphView object which can be used to manipulate the modeling element instances and their properties. e.g. clicking on the dimension level geographic region reveals the dialog shown in Figure 3.

8 Conclusions and Future Work

The objective of our case study was to show that standard repository systems and the IRDS standard offer a good basis to build meta-data driven applications (e.g. a generic case tool). Therefore, we built a project specific instantiation of the IRDS reference model defining the contents of the different layers and deciding which layers are fixed or variable at system runtime and design time. The next step was to find useful representation mechanisms for the different layers. For example, we chose a typed graph as the formalism of our models and a graph grammar to describe the syntactical constraints. For the technical design this conceptual model must be mapped to a two layered model due to the restrictions of current storage systems (e.g. databases and repository systems). The next challenge was the design of the self-configuring core application and the user interface. We used a three layered architecture (storage, kernel, GUI) and used the model view controller pattern for the design of the user interface. This separation enabled the reuse of graphical editing components. During the implementation, we made extensive use of the functionality offered by the repository system (namely version and configuration management).

The IRDS reference model was helpful during the early phases of the design. But as already described, repository systems and computer languages currently do not support more than one flexible layer of abstraction at runtime. Therefore, a mapping to a two layered model (schema and instances) has to be performed. This means that instantiation relationships have to be modeled /implemented manually (e.g. performing dynamic type checks).

The GraMMi tool currently has the status of a research prototype that means that some of the functionality is still missing and some of the functionality is cumbersome to access (e.g. there is no assistance tool to comfortably define graph grammars and graphical representations).

Future research will be dedicated to the integration of our approach with existing Meta-CASE tools. The Warehouse instantiation of the GraMMi tool is used in several data warehouse projects to model the static schema ([15]) and the user behavior ([14]). Current work is dedicated to development of a component for the automatic generation of logical and physical datawarehouse schemas from our models.

Authorized licensed use limited to: Bibl Univ Moretsu Plantin. Downloaded on March 29,2022 at 14:36:31 UTC from IEEE Xplore. Restrictions apply.

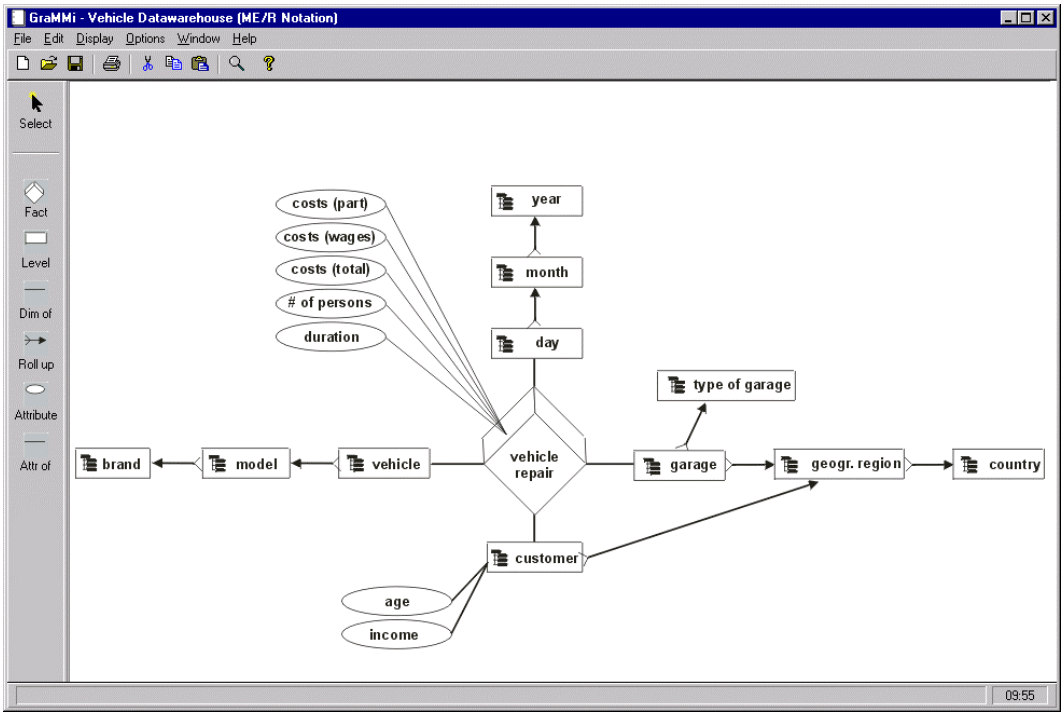

figure 10: A screenshot of GraMMi displaying a sample ME/R model

Acknowledgements

We thank our master student Stefan Haas for implementing the core of the GraMMi system and the Softlab Corp. for supporting our work in an unbureaucratic way. Furthermore, we want to express our gratitude to the anonymous reviewers of this paper. They provided very detailed and helpful comments that played an invaluable role in the enhancement of this paper.

References

[1] A. Alderson: *Meta-CASE Technology*. in A. Endres and H. Weber (eds.): *Software Development Environments and CASE Technology*, LNCS 509, Springer-Verlag, 1991.

[2] P.A. Bernstein, U. Dayal: *An Overview of Repository Technology*, in Proc. of the International Conference on Very Large Databases (VLDB), Chile, 1994.

[3] S. Chaudhuri, U. Dayal: *An Overview of Data Warehousing and OLAP Technology*, ACM SIGMOD Record 26(1), March 1997.

[4] J. Ebert, R. Süttenbach, I. Uhe: *Meta-CASE in Practice: a Case for KOGGE*. in A. Olive, J. A. Pastor: *Advanced Information Systems Engineering*, LNCS 1250, Berlin, 1997.

[5] H. Ehrig: *Introduction to the algebraic theory of graph grammars (a survey)*, in Proc. of the International Workshop on graph Grammars and Their Application to Computer Science and Biology, LNCS 73, Springer Verlag, 1979.

[6] Engels G., Lewerentz C., Nagl M., Schaefer W., Schurr A.: *Building Integrated Software Development Environments Part I: Tool Specification*. In ACM Transactions on Software Engineering and Methodology, vol. 1, no. 2, 1992.

[7] P. Findeisen (ed.): *The Metaview System*, http://www.cs.ualberta.ca/~softeng/Metaview/doc/, 1994.

[8] R.I. Ferguson, N.F. Parrington, P. Dunne, C. Hardy, J.M. Archibald, J.B. Thomnson: *MetaMOOSE – an Object-Oriented Framework for the construction of CASE tools*, in Proc. of The First International Symposium on Constructing Software Engineering Tools (CoSET'99), USA 1999.

[9] E. Gamma, R. Helm, R. Johnson and J. Vlissides: *Design Patterns Elements of Reusable Object-Orientated Software*, Addison-Wesley, 1995.

[10] *IRDS Framework* ISO/IEC IS 10027, 1990.

[11] Rational, Inc. *Rational Rose 98 Technical Reference*, 1998.

[12] J. Reekers, A. Schürr: *Defining and Parsing visual Languages with Layered Graph Grammers*, Journal of Visual Languages and Computing 8(1): 27-55, 1997.

[13] M. Rossi, S. Kelly: Construction of a CASE tool: the case for MetaEdit+ in Proc. of The 1st International Symposium on Constructing Software Engineering Tools (CoSET'99), USA, 1999.

[14] C. Sapia: *On Modeling and Predicting Query Behavior in OLAP Systems*, in Proceedings of the CaiSE'99 Workshop on Design and Management of Data Warehouses 99 (DMDW99), Heidelberg, June 1999.

[15] C. Sapia, M. Blaschka, G. Höfling, B. Dinter: *Extending the E/R Model for the Multidimensional Paradigm* in Y. Kambayashi et. al. (Eds.): *Advances in Database Technologies, LNCS Vol. 1552, Springer, 1999.*

[16] K. Smolander, K, Lyytinen, V.-P. Tahnavainen, O. Martiin: Meta-Edit – *A flexible graphical environment for methodology* modelling in R. Andersen, J. Bubenko, A. Sølvberg (eds.): *Advanced Information Systems Engineering*, Trondheim, Norway, 1991.

[17] Softlab Technologies, *Enabler 2.0 Technical Reference*, 1999.

[18] P.G. Sorenson, J.-P. Tremblay, and A.J. McAllister: *The Metaview System for Many Specification Environments*, IEEE Software, vol. 14, pp. 30-38, 1988.

Authorized licensed use limited to: Bibl Univ Moretsu Plantin. Downloaded on March 29,2022 at 14:36:31 UTC from IEEE Xplore. Restrictions apply.