APPLYING SOFTWARE ENGINEERING TECHNIQUES IN THE DEVELOPMENT AND MANAGEMENT OF LINEAR AND INTEGER PROGRAMMING APPLICATIONS

Fernando Costa, Leonardo Murta, and Celso C. Ribeiro

Computing Institute (IC), Universidade Federal Fluminense, Niterói, RJ 24210-240, Brazil

Emails: flpcosta@ic.uff.br [Costa]; leomurta@ic.uff.br [Murta]; celso@ic.uff.br [Ribeiro]

Original version: August 20, 2013

Revised:

Accepted:

Abstract

This work addresses characteristics of software environments for mathematical modeling and proposes a system for developing and managing models of linear and integer programming problems. The main features of this modeling environment are: version control of models and data; client-server architecture, which allows the interaction among modelers and decision makers; the use of a database to store information about the models and data scenarios; and the use of remote servers of optimization, which allows to solve the optimization problems on different machines. The modeling environment proposed in this work was validated using mathematical programming models that exploit different characteristics, such as the treatment of conditions for generating variables and constraints, the use of calculated parameters derived from other parameters, and the use of integer and continuous variables in mixed integer programming models, among others. This validation showed that the proposed environment is able to treat models found in various application areas of Operations Research and to solve problems with tens of thousands of variables and constraints.

Keywords: Mathematical Modeling, Linear Programming, Integer Programming, Software Engineering, Version Control.

1 INTRODUCTION

In a company, the Operations Research team develops mathematical models for solving optimization problems of various business areas. The professionals who make the decisions in these business areas use such models through a software application, where they can analyze different scenarios to support the decisions that have to be made. These professionals do not need to know the methods of mathematical modeling. They interact with a software application that encapsulates the techniques of Operations Research and allows the processing and the analysis of the data involved in decision-making. In this way, the Operations Research professionals must know the techniques for solving optimization problems to provide applications to decision makers that generate appropriate responses to problems in the company.

The quality of a decision support system is measured not only by the results it provides, but also by the ease of interaction among users, application, and data, and by the agility of its development, because time is an important variable in decision-making. Decision makers expect that Operations Research applications provide consistent results on a timely basis, which brings

profit to the business areas. Thus, some challenges in developing this kind of application can be identified, such as: the management of data needed for modeling a problem, the concurrency control over the mathematical models development, their associated data, and the obtained results, and the development of human-computer interface (HCI), which enables the interaction of business professionals with the decision support system.

Therefore, the process of developing a mathematical model, which is used in the decision making chain of an organization, can be associated with the process of software development. The area of Software Engineering has concepts that can also be applied in the construction and maintenance of mathematical models and decision support systems. According to PRESSMAN (2006), there are five phases in software development that must be performed to create a software: communication, when the project initiates and requirements are identified; planning, responsible for estimating the cost and preparing the project schedule; modeling, when the software and its architecture are designed; construction, when the software is coded and tested; and deployment, responsible for delivering the software to its users and to plan the maintenance.

The development of a mathematical model shares some similarities with the aforementioned Software Engineering phases. Activities such as gathering requirements from users (in this case, the decision makers), designing the solution, implementing, testing, deploying, and maintaining can also be found in the creation of decision support tools. Thus, the motivation of this work is to support the development of Operations Research applications, mainly involving mathematical programming modeling, dealing with information of the business areas, and generating results to assist in decision making.

Configuration Management techniques can be adopted to support some of these activities. According to ESTUBLIER (2000), Configuration Management is a field of Software Engineering that controls the evolution of complex systems. Systems evolve over time, since their requirements change and defects are found. Important contributions of Configuration Management are version control, managing repositories of software artifacts, and change control, which supports the software development process by tracking issues from their request up to their implementation in the software.

The aim of this paper is to present a software environment for modeling linear and integer programming problems, dealing not only with the modeling activity, but also with the life cycle of mathematical models and associated data. In addition, the environment must support information sharing among its users. This environment, called GeMM – Manager of Mathematical Models (*Gerenciador de Modelos Matemáticos*, in Portuguese) – must meet the needs of both modelers and decision-makers. The main features of GeMM are: modeling of Linear Programming (LP), Integer Programming (IP), and Mixed-Integer Programming (MIP) problems; versioning of mathematical models and associated data; application generation from a mathematical model for data entry and analysis of results; use of an integrated database to store the models and data; and generation of the mathematical model documentation. In the remainder of this text, we shall refer to a MIP problem as a generalization that also encompasses both LP and IP problems.

This paper is organized in five sections. The second section presents some background concepts and related work. The third section introduces our approach for managing the life cycle of mathematical programming models. The fourth section presents some implementation details of the GeMM environment and a case study of how the modeling environment is used to solve optimization problems. Finally, the fifth section summarizes the contributions and limitations of this work.

2 BACKGROUND AND RELATED WORK

We performed a survey and a systematic literature review (KITCHENHAM, 2004) to identify the main research challenges and solutions related to the conception of mathematical programming models and the management of their life cycle. This survey involved Operations Research graduate students and professionals. Its main goal was to identify how decision support systems are currently being conceived and maintained, giving us a picture of the state-of-the-practice. The literature review was intended to complement the survey with a picture of the state-of-the-art, providing an overview of the existing works on the conception and management of mathematical programming models. All in all, the survey and the literature review focused on answering the same question: what are the most important requirements for conceiving and managing mathematical programming models, considering both the model developer and the decision maker perspectives?

FOURER (2011) contributes to define the nomenclature and the software elements involved in solving Linear Programming (LP) and Mixed Integer Programming (MIP) problems. His work focused on both industry and academia to identify and classify the major tools involved in solving this type of problem. The tools can be classified as solvers and modeling environments. Solvers are tools that search for a solution of a given problem. They receive an instance of a MIP problem as input and provide the optimal solution as output. Modeling environments are tools that interface between the Operations Research professionals and the solvers, providing general and intuitive ways to express symbolic models. They usually offer features that allow importing and processing data, generating problem instances for solvers, analyzing result, and interfacing with other applications, such as Database Management Systems (DBMS) and spreadsheets.

GEOFFRION (1989) describes five characteristics desirable for computer-based modeling environments to support Operations Research applications: to deal with the entire modeling life cycle; to consider the decision makers necessities; to support the model evolution throughout its existence; to adopt a model definition language independent from the languages used to solve the problem; and to allow easy resource management. These characteristics act as general guidelines of what is expected for a modeling environment.

Additionally, MURPHY et al. (1992) present a series of representations for MIP problems, such as matrix generators, algebraic representations, structured modeling, and database schemas. Matrix generators, such as OMNI (HAVERLY, 2001), were the first adopted representation. These generators provide a procedural language that allows the creation of MPS files (IBM CORPORATION, 1975), which represent an instance of a MIP problem and are interpreted by most solvers. While the MPS format allows efficient representation of sparse matrices, it makes modeling and debugging hard (BROOK et al., 1988). Algebraic representations, such as GAMS (BISSCHOP and MEERAUS, 1982) and AMPL (FOURER et al., 1990), describe models as mathematical expressions, being quite general and concise. Structured modeling, introduced by GEOFFRION (1987), aims at developing a general specification to represent, in an unambiguous way, all essential elements of a variety of models. Finally, database schemas consider two distinct but mingled requirements: the need to register information about the mathematical model structure and the need to register the problem data and its results.

As one can note, a prominent feature is the separation between data and model, primarily using database applications and spreadsheets. LEE (1991) stores mathematical models apart from the data of problem instances. Models should be general to deal with a range of common problems. FOURER (1997) also exploits the use of databases to handle mathematical models,

presenting database structures for mathematical programming models. Today, DBMSs are widely used in organizations, managing and centralizing key information. Therefore, the decision support systems must somehow be integrated with these information repositories.

Going in a different direction, MAKOWSKI (2005) discusses the necessary requirements for developing modeling environments. According to his work, the model development life cycle is composed of the following phases: requirements elicitation, design, construction, testing, use, review, maintenance, documentation, model analysis, results analysis, and model evolution. The author states that the existing modeling environments only meet one or two phases of the whole model development life cycle. Issues such as data processing, data sources documentation, change tracking, models integration, and access control are especially critical for complex or large-scale models. The following activities were identified as poorly supported by the existing modeling environments: version control over the model specification, preparation of data for the definition of parameters, generation of model instances binding the model specification with a specific data set, and the use of multiple views to analyze the results.

For instance, MATURANA et al. (2004) use the mathematical model as input to automatically generate the user interface and the database structure of a decision making system. This strategy supports agile development of Operation Research applications, increasing the productivity and the easiness for providing solutions to the decision makers of an organization. However, their approach lacks an underlying version control infrastructure, making it hard to evolve the model and the generated system.

Besides the use of databases as an integration tool for models, their respective instances and the problem data, FOURER (1998) adopts web technology as an access infrastructure to optimization systems. Moreover, FOURER et al. (2010) propose the use of an optimization server that receives jobs via XML (eXtensible Markup Language) files transported through web services. This kind of service is useful to isolate the modeling language and the solvers from the decision making system itself. These works show how Software Engineering can be used as an enabling technology to the construction of Operations Research applications.

The survey and the literature review showed that the following features are relevant for the development of modeling environments: models, instances, and data separation; integrated databases to store models and data; interface with external software, such as solvers; graphical user interface for data input and results analysis; and efficient communication with solvers. We could not identify a research or commercial modeling environment for MIP that fulfills the needs of both modelers and decision makers, allowing the interaction of such professionals throughout the mathematical model and its data life cycle. This scenario is especially difficult because models, instances, and data evolve over time, demanding a consistent versioning solution.

3 MANAGEMENT OF MATHEMATICAL MODELS

In this section, we introduce our approach for managing mathematical programming models, named GeMM. It is intended not only to provide a tool for writing mathematical models and equations, but also to control the life cycle of a model, since its establishment and until its use by the decision makers and evolution. GeMM adopted a client-server architecture, stores its information in a database, provides version control to both mathematical models and associated data, and enables information sharing among users of the system, observing concurrency control. The remaining of this section presents the architecture of GeMM and discusses model formulation, data management, and, finally, version control of both models and data.

3.1 ARCHITECTURE

Two distinct roles in the development of mathematical programming models for supporting decision making are considered: modeler and decision maker. The modeler is an Operations Research professional who knows in details the modeling techniques to handle optimization problems in a specific domain. The decision maker does not have specific knowledge of Operations Research, but knows the domain data and has the necessary skills for using optimization models to support decision-making processes in the organization.

Thus, the system described in this article was designed according to a client-server architecture. In this architecture, it is possible to have modules to meet different functions that are interconnected through a database. Furthermore, this architecture allows the system to support multiple users, assuming the aforementioned roles. Figure 1 shows the overall architecture of the system. Basically, modelers interact with the application by creating and maintaining mathematical programming models in order to solve optimization problems. On the other hand, decision makers populate the system with domain data, with the possibility of creating different scenarios for the same optimization problem, and visualize the results obtained by solvers. With this structure, each user has an appropriate environment to perform her work, either modeling or using a model, but these environments shared data through a single database.

Another important element in the architecture is the optimization servers. For large problems that require high processing power in order to find optimal solutions in appropriate time, machines with dedicated hardware and software are usually used for running the optimization processes. Furthermore, the sharing of such machines is highly desirable to solve different problems at different moments. Since a desktop computer may not have adequate capacity to perform the required processing, GeMM makes use of optimization servers. The application server is responsible for managing the execution of optimization algorithms by controlling the distribution of tasks among optimization servers that actually run the solvers.

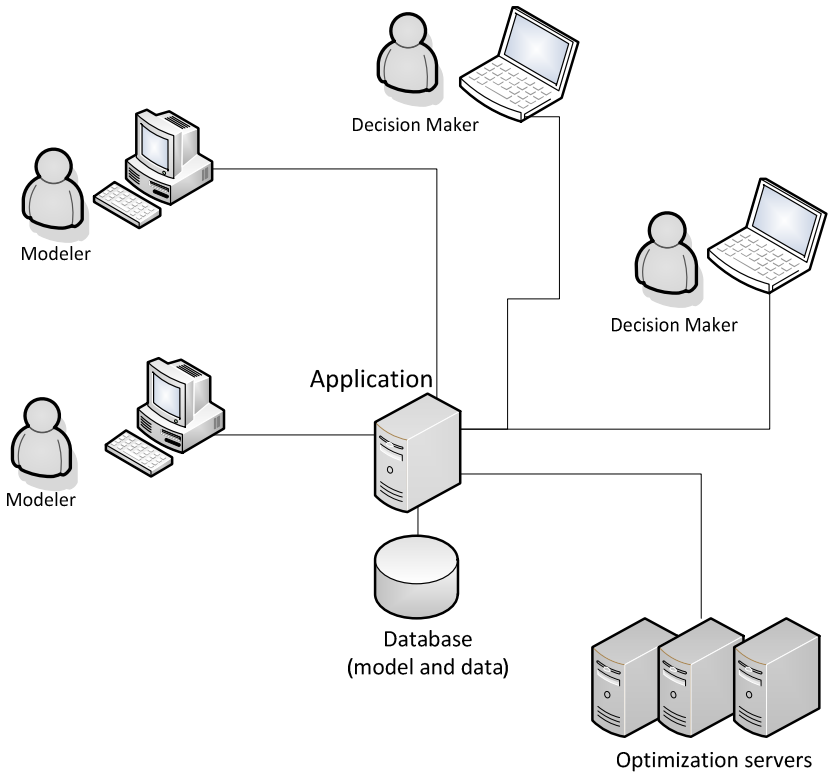

Figure 1: Basic architecture of the system

3.2 MATHEMATICAL MODELING

The main function of GeMM is to support the modeling of linear and integer programming problems. In the GeMM approach, problem modeling is done directly in the system, using formularies, and is stored in a database, which also stores instance data. Thus, in this architecture, the model and the data can be shared among users, allowing integration and collaboration during modeling. The interaction between users of GeMM is discussed in Section 3.5.

Modeling elements are designed according to algebraic representations, especially AMPL (FOURER *et al.*, 1990). GeMM uses a database to store MIP models. The information and data models are in the same database structure, but independently stored. This separation is important because it allows the creation of different instances of the same optimization problem from a mathematical model, by changing, for example, only the input data. GeMM uses five main elements for modeling MIPs: sets, parameters, variables, constraints, and objective function.

Sets are collections of well-defined and distinct objects relevant to the problem. Each set contains indices associated with it, allowing the model to reference a generic element belonging to the set. A very common type of set is, for example, a sequence of integers. In GeMM it is also possible to define subsets.

Parameters are invariants used in modeling. They consist of values directly informed by the users (i.e., input data) or calculated from an expression. This difference exists only from the point of view of the modeling environment, which assigns the value of the parameters before the optimization process begins. From the point of view of the solver, all parameters are input data. They can represent scalar values, when they are not indexed, or arrays, when they are indexed by the elements of one or more sets.

Variables hold values that are calculated by the optimization process. Like parameters, variables can also be indexed by the elements of sets. A variable can be either continuous, integer or binary, since the system allows the modeling of linear, integer, and mixed integer programming problems. Variables may have upper and lower bounds. Constraints are represented by linear equations or inequalities. GeMM allows the use of generating conditions for model variables and constraints. The objective function is a linear function that should be either maximized or minimized.

An important feature of GeMM is the separation between the data and the mathematical model. The structure of the GeMM database must be able to store the model, the input data, and the results for every MIP. The data structure presented in the class diagram of Figure 2 represents the mathematical programming model.

In this diagram, the *Project* class represents a MIP to be solved. A project groups all the model elements that must be created for the correct representation of the problem. The model elements are represented by the class *ModelElement* and its subclasses in the diagram of Figure 2: *Set*, *Variable*, *Constraint*, *Parameter, ObjectiveFunction*. One or more indices can be associated to each set, whose representation is made by the class *Index*. The class *Indexable* generically represents the model elements that have associated indices and therefore can be indexed by them.

This data structure is used to handle and store mathematical models in GeMM. We remark that it includes only the model definition, i.e., it does not address the input data and the results. For example, it is possible to identify which sets are defined, but it is not possible to know which elements compose these sets and what values can be assigned to the indices. The

structure depicted in Figure 2 also does not address the versioning model, which will be discussed later.

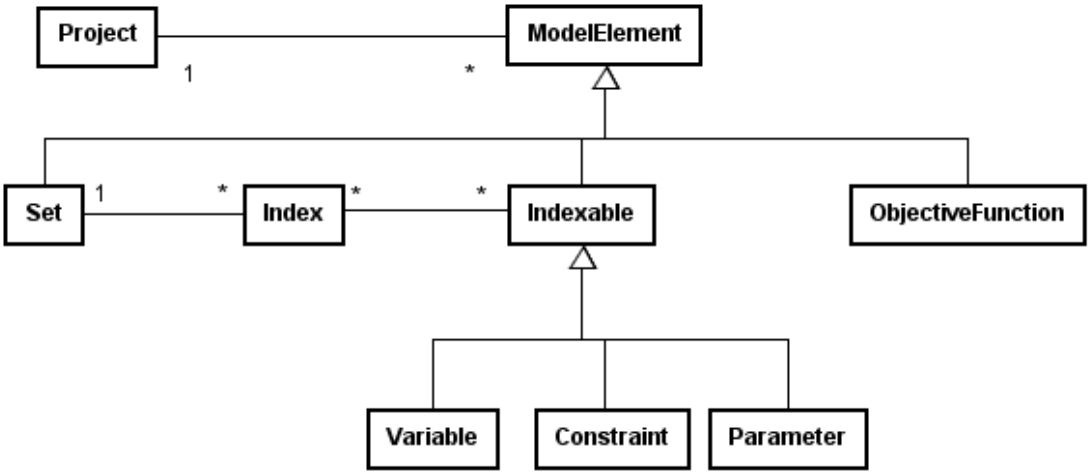

Figure 2: Class diagram representation of the mathematical programming model

3.3 DATA REPRESENTATION

An instance of an optimization problem comprises sets of the parameter values. A solver receives a problem instance as input, applies one or more solution methods, and returns the results (FOURER, 2011), which are the values assigned to the variables and the objective function value.

As described in Section 3.1, GeMM users are divided into two main roles: decision makers and modelers. This latter role should not change the model formulation, i.e., the definition of the variables, constraints, and objective function, but should work with the values of the elements of the sets, with the parameter values, and with the results.

3.3.1 DATA STRUCTURE

Each mathematical model needs data in a different structure. A particular modeling project may require the creation of one or more sets, whose indices can index several parameters and variables. The approach used in GeMM associates indexing elements with the primary keys of entities in the database. Each database entity has a set of attributes that characterize its primary key.

In the following, we illustrate our approach without loss of generality with a linear programming problem in its canonical form (BAZARAA, 1977):

$$\text{Minimize} \quad \sum_{j=1}^{n} c_j . x_j \tag{1}$$

$$\text{Subject to} \quad \sum_{j=1}^{n} a_{ij} . x_j \geq b_i, \quad i = 1, \ldots, m \tag{2}$$

$$x_j \geq 0 \quad j = 1, \ldots, n \tag{3}$$

The indices *i* and *j*, that appear in equations (1), (2) and (3), represent set elements. The index *i* represents an element that belongs to the set $I = \{1, 2, ..., m\}$, as shown in expression (2), and the index *j* denotes an element that belongs to the set $J = \{1, 2, ..., n\}$, as shown in

expression (3). An instance of this model is defined by sets $I = \{1, 2, 3\}$ and $J = \{1, 2\}$ and a tabular structure for storing them can be seen in Figure 3.

I
1
2
3

J
1
2

Figure 3: Data structure of sets I and J

Once the elements of each set are known, we can define the data structure associated with those sets. When modeling the MIP in its canonical form we identify the parameters c_j, b_i, and a_{ij}, and the variable x_j. Figure 4 shows the three entities created to store the data and variable parameters, which are: the entity whose primary key is the combined values of i and j, the entity whose primary key is the value of i, and the entity whose primary key is the value of j.

Entity I e J

i	j	a_{ij}
1	1	
1	2	
2	1	
2	2	
3	1	
3	2	

Entity I

i	b_i
1	
2	
3	

Entity J

j	c_j	x_j
1		
2		

Figure 4: Data structure of parameters and variables

In this example, the parameter a_{ij} is indexed by the indices $i \in I$ and $j \in J$. Therefore, the database entity that stores the values of this parameter must have a primary key combination of elements of the sets I and J. In the same way, we developed the data structure for the b_i parameter, which must be an entity whose primary keys are the elements of the set I. Moreover, parameter c_j and the variable x_j should be entities whose primary keys are the members of set J. Thus, the columns i and j of the entities defined in Figure 4 are foreign keys, respectively to the columns I and J of the data structures defined in Figure 3.

The GeMM approach enforces that, for a given mathematical programming model, there will always be an entity database for each of the sets, where its elements are stored. The values of the elements compound the primary key of the entity, since there are no repeated elements in the same set. Also, there will always exist a database entity for each different combination of indices associated with the values of parameters and variables. These entities must have an index for each attribute that composes their primary key and another attribute for each variable or parameter that has the same combination of indices. All variables or parameters that have the same combination of indices, in the same order, must have a corresponding attribute in the same database entity, as for variable x_j and parameter c_j, in the previous example. An entity database without attributes associated with an index is also created to hold the values of the elements that are not indexed, as the objective function, parameters, and non-indexed variables.

However, this way of handling the data of mathematical programming models requires a different data structure for each model. This occurs because each mathematical model has its

own characteristics, such as different set definitions and indices, different modeling elements, and different combinations of indices that index parameters and variables. Thus, each model managed by GeMM needs a different database scheme to store the data associated with it. As GeMM is intended to manage many different MIPs, metamodeling is a natural solution to store and handle the data structures needed for the models, regardless of how they are modeled and their application area.

3.4 METAMODELING

The idea of metamodeling the database, which was used by JEUSFELD and JOHNEN (1994), consists in building a generic data structure to represent a logical data model into a relational database that meets defined specifications. Figure 5 shows the data structure for logically storing the database schemas, i.e., schemas metamodeling. This class diagram represents the schemas of the database. A schema consists of all entities within the same domain. The *Entity* class represents existing entities and is composed of attributes.

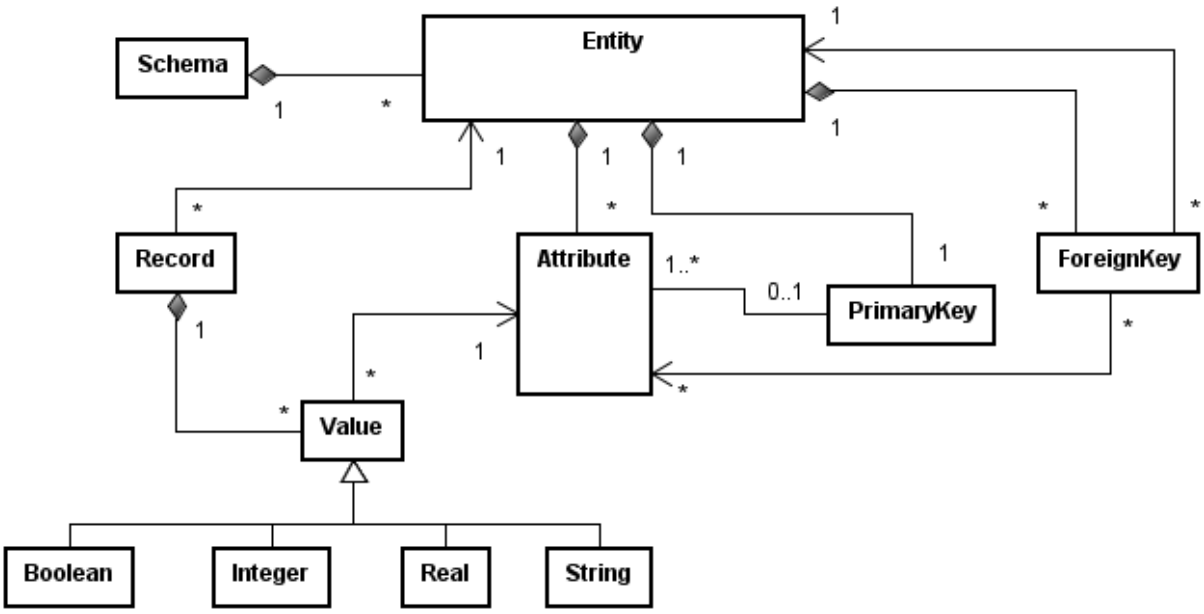

Figure 5: Data structure for storing metamodeling schemas

The *Attribute* class represents the properties of the attributes associated with the entities in the metamodel. This class should contain the necessary information about integrity for the database management systems, such as the possibility of assuming a null value, for example. The relationships between entities are represented by classes *PrimaryKey* and *ForeignKey*. The class *PrimaryKey* indicates which attributes of an entity belong to their respective primary keys. The *ForeignKey* indicates which entity attributes represent links with other related entities.

Figure 5 also presents the *Record* class, which is the logical representation of a row of a database table. Its role is to group the values of the attributes that compose a single record. The *Value* class and its subclasses are identified only by a *Record* (row) and *Attribute* (column). It has the function of storing the data and its subclasses are used in accordance with the attribute type: boolean, integer, real, or string, for example. This data structure allows the identification of two types of metamodeling classes: the classes that represent the structure (which are *Schema*, *Entity*, *Attribute*, *PrimaryKey*, and *ForeignKey*) and those that represent the data (which are *Record*, *Value*, and its subclasses).

Metamodeling allows using a unique schema database to store different data models. GeMM uses metamodeling to work with the data structures that store and manage data from mathematical models. The flexibility provided by metamodeling is necessary to enable changes in the definition of the models without having to change the schema of the database, allowing GeMM to manage more than one model in the same database. The structure proposed in GeMM to handle data models through metamodeling is shown in Figure 6. This structure was developed from the basic structure of metamodeling, shown in Figure 5, and adapts as necessary to manage the data of the mathematical programming models. The classes that are subclasses of *ModelElement* are used to represent the mathematical model.

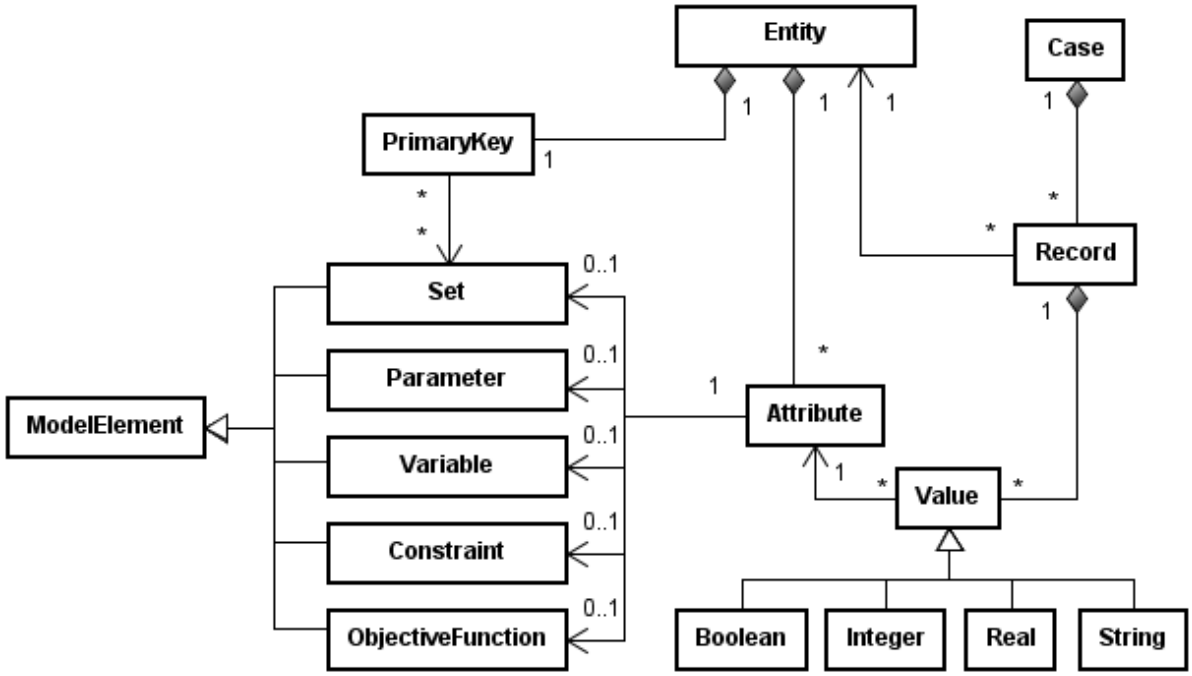

Figure 6: Data structure using metamodel

In this approach, the entities needed to represent the data structure of the mathematical models are handled by the class *Entity*. As described earlier, the primary keys of entities are attributes related to indices of the sets, which index the modeling elements. The foreign key concept was omitted from the diagram in Figure 6, since the attributes related to the indices of the sets are the only ones that may be foreign keys. The model elements that can receive a value, such as parameters and variables, are associated with an attribute of an entity. The entity that holds a given attribute is defined by the combination of indices that index the associated model element.

Therefore, *Entity*, *PrimaryKey*, and *Attribute* are classes directly related to the modeling elements and define the data schema for a particular mathematical model. Each model, once specified, has only one data schema. *Case*, *Record*, *Value*, and its subclasses are used to store data associated with the model.

The *Value* class is used to store a value of one model element. The *Attribute* class knows the data type of the model element and the *Value* class knows how to store it. The model elements that have one or more indices must have several instances of *Value*, one for each combination of the values of the indices that index these elements. The *Record* class is responsible for grouping all values of the same entity, having the same primary key. Finally, the

Case class is responsible for grouping all data in a single set of data. It allows different scenarios with different parameter values, and hence different results, for the same mathematical model.

3.5 VERSION CONTROL

An important feature of GeMM is the versioning of mathematical programming models and their scenarios with associated data. GeMM allows different users to collaborate designing and maintaining the mathematical models, managing instance of data, and optimizing problems. GeMM has versioning as a key element in its design to accompany this interaction of users with the mathematical models and data, and to ensure the integrity of information. Among its responsibilities is saving the history of changes and managing the evolution of models and data over time, which enables control over corrective and perfective maintenance.

3.5.1 MODEL VERSION CONTROL

CONRADI and WESTFECHTEL (1998) discuss the characteristics and classifications of version models for software configuration management. An important point of this type of system, which includes version control, is the definition of the product space. The product space describes the complete structure of what should be versioned. In the case of GeMM, the product is a mathematical model consisting of model elements. It is also fundamental to define the versioning space, which determines which items must be versioned, how versions are organized, when new versions are created, and the granularity of the versioned items. The granularity of the versioned items is defined as the size of the smallest versioned object.

In the GeMM approach, the evolution of mathematical models occurs by creating different versions of a given modeling project. A version of the modeling project is a group of versions of the model elements that compose it. The granularity of versioning in GeMM is the model elements, i.e., sets, parameters, variables, constraints, and objective function have their independent versions. Each of these elements is individually versioned and the version of a modeling project is given by the most recent version of its modeling elements. A new version is created due to the need of modifying one or more model elements within the project.

We used the example of a linear programming problem in its canonical form to illustrate the versioning of mathematical models in GeMM. In this example, we created three versions of the same mathematical model. First, we created the modeling project called “LP problem in canonical form” and its first version, containing the sets I and J, the parameters a_{ij} and b_i, the variable x_j, the constraint, and the objective function, as shown in Figure 7. At this point, all model elements are in their first version.

Then, suppose that we have identified an error in the model implementation of the in its canonical form: for example, the parameter a_{ij} has been modeled as an integer and not as a real number, as expected according to expression (3). After creation of the first version, there is still another modification to be made with respect to the model presented in expressions (1), (2), and (3): the costs of the variables were erroneously implemented as a constant. This second version creates a parameter c_j and changes the expression of the objective function to include this parameter. Since the project is in its second version, so are the objective function and the parameter c_j. This change led to the creation of a second version of this modeling project, which appears in Figure 8.

We observe that all model elements associated with the second version are still in the first version, except the parameter a_{ij}, c_j and the objective function. The modeling elements that did not change are associated with the first and second versions of the modeling project.

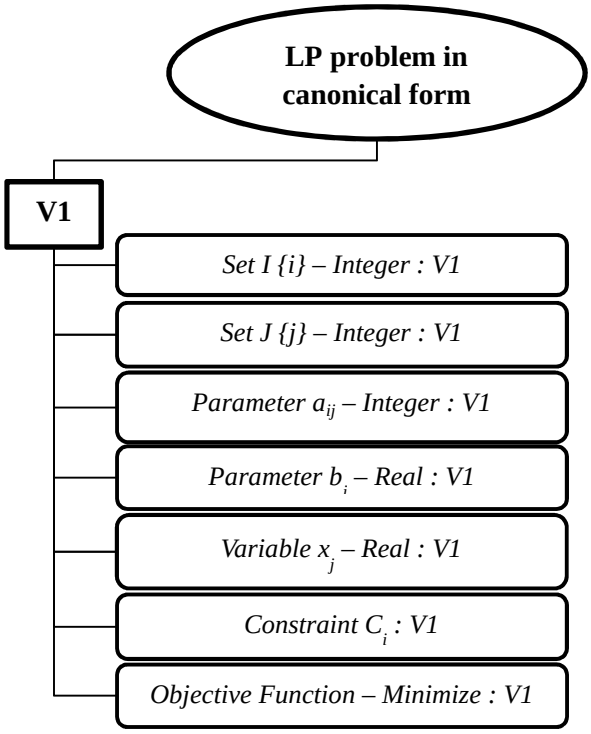

Figure 7: Initial version of a linear programming problem in the canonical form

Each element has its individual version, but their life cycle is always associated with a particular modeling project. Once edited, the element version is replaced by the same version of the project after the change. Thus, it is possible to identify which are the versions of a single model element and, also, given a version of the modeling project, which elements were changed in that version. It is important to notice that the versioning information related to a model element is not altered nor replicated. The modeling project only links to each specific version on model elements. This link is broken only when there is a change in the element and a new version is created.

This versioning strategy is defined by CONRADI and WESTFECHTEL (1998) as *Product Versioning*, in which versions of the elements are within the global version of the modeling project. With this type of versioning, the conventional reading "Version 2 of the objective function" should instead be read as "The objective function in the second version of the LP problem in its canonical form".

As a client-server and multi-user application, GeMM allows more than one user to concurrently access a particular modeling project. This requires a policy for concurrency control in order to prevent loss of information. According to ESTUBLIER (2001), a scenario with *N* users simultaneously working over the same project leads to *N*+1 different product versions, one for each user and one for the original product. If these changes occur in parallel, it is necessary to merge all of them to generate the final product.

However, according to MENS (2002), the need to combine different versions of the same project depends on the selected concurrency control mechanism. With pessimistic concurrency control, in which only one user at a time can modify elements of a project via lock mechanisms, changes do not occur in parallel and are not subject to merge. On the other hand, in optimist concurrency control, where users change the project in parallel, the process of merging different modifications of complex objects may become error-prone and counter-productive

(PRUDENCIO et al., 2012), being necessary to combine the syntax and the semantics of the product in question. Due to that, GeMM adopted pessimistic concurrency control, by locking the project that is being changed. For this reason, GeMM does not address merge of modifications. However, the fine granularity adopted by GeMM allows the identification of differences between two versions in terms of added, removed, or changed model elements.

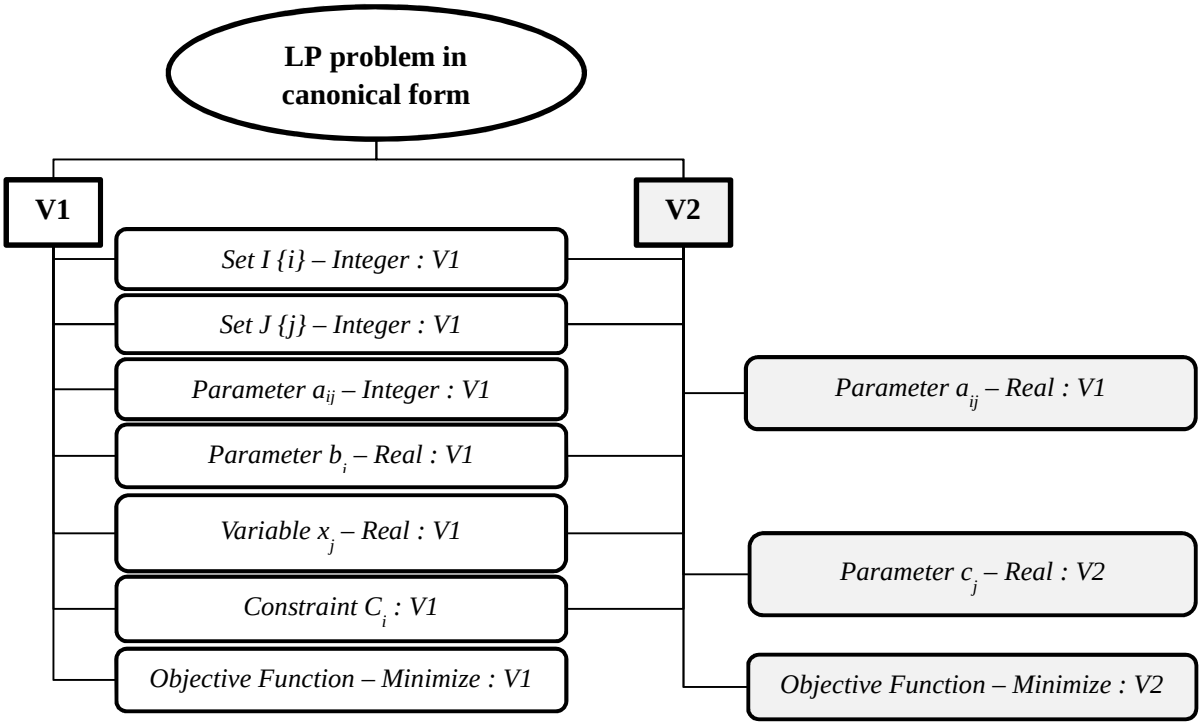

Figure 8: Second version of the linear programming problem in the canonical form

Once a user starts editing the mathematical model in GeMM, the entire project is locked for editing. When the modification is completed, the user can release the project lock to allow others to edit it, through the creation of a new version. Figure 9 shows the modeling project in its third version. The creation of this version was motivated by a change being made by the user “Modeler 1”. The earlier versions V1 and V2, which have already been released, can only be accessed for reading. Further modifications may be made by creating a new version.

While “Modeler 1” is still editing the third version, shown in Figure 9, the other users can only read the model. This situation persists until “Modeler 1” releases or discards the lock. When a version is released, it cannot be removed or edited. Thus, it is guaranteed that the whole life cycle is registered. Before the release, elements can be freely added, changed, and removed to the current version of the project by the user that holds the lock. However, once released, any change in the project requires the creation of a new version.

3.5.2 DATA VERSION CONTROL

In addition to model version control, GeMM also provides data version control. Each user, in the role of decision maker, can create data cases for any released version of the modeling project, once its structure cannot be further modified. Each case has its versions and the same concept of concurrency control used for the model is applied. The product to be versioned is the data associated with the models, namely, the elements of the sets and the parameters values. Figure 10 shows an example of the association between data versions and model versions.

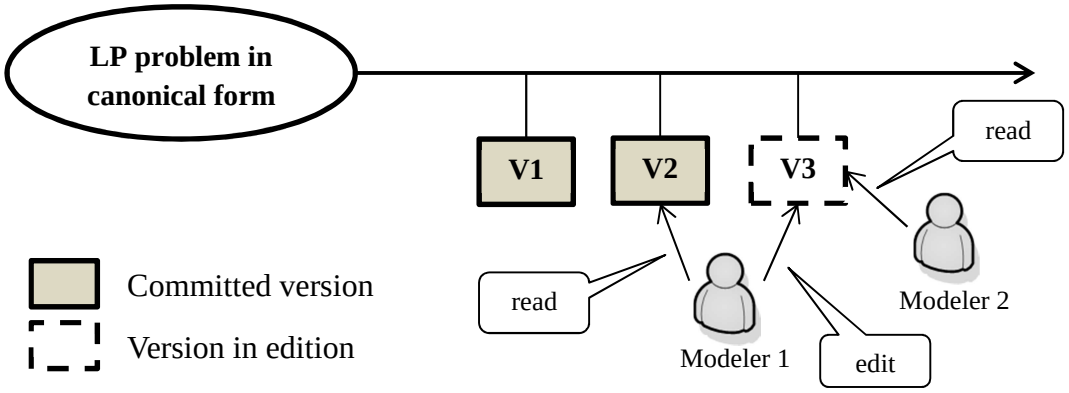

Figure 9: Concurrency control for a modeling project

The concurrency control mechanism is also pessimistic. Once a data case is being edited, it is locked and other users can only read it. When a data case is released, it can only be modified by the creation of a new data case version. In the example shown in Figure 10, two data cases were created for the third version of the "linear programming problem in its canonical form", which has been released. "Case 1" has a certain set of values for model elements, and is in the second version. As the user "Decision Maker 1" is editing the "Case 1", the user "Decision Maker 2" cannot change it in parallel. This user can only view and copy its information to another data case. In this example, the "Decision Maker 2" created "Case 2" with its own data to edit.

Once the versions of the data cases have been released and cannot be modified or removed, the historical data used for making decisions is also stored. Thus, it is possible to trace back the optimization model and the data that motivated a particular decision. This kind of traceability is fundamental for reproducibility and auditability of the results, and is called data provenance in the literature (FREIRE et al., 2008).

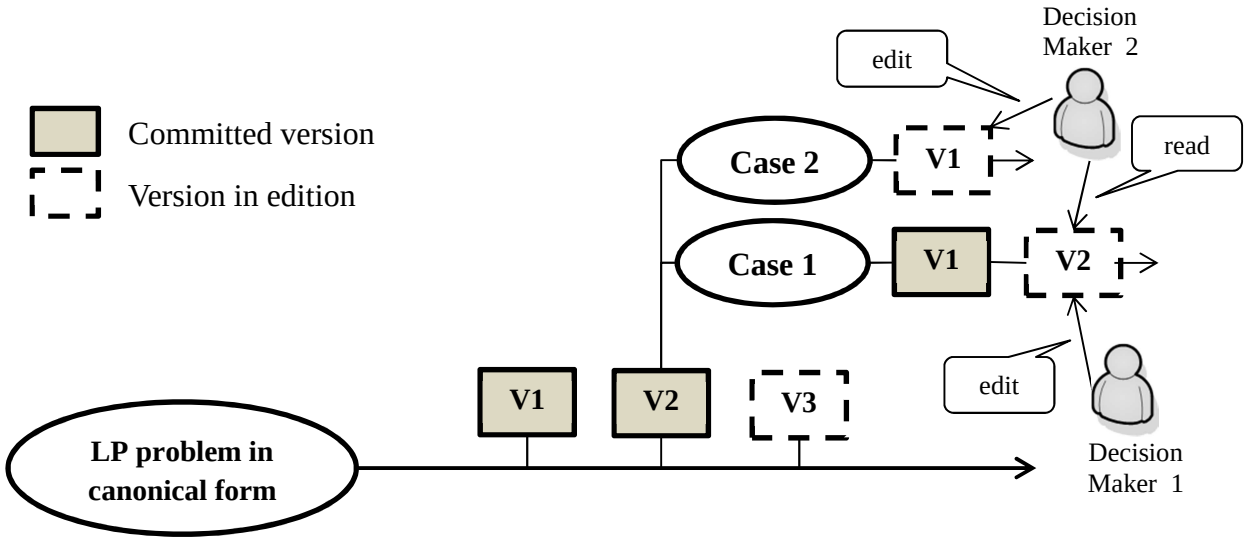

Figure 10: Versioning of a modeling project and associated data

4 IMPLEMENTATION AND USE OF GEMM

The previous section presented the main characteristics of our approach, while this section details the implementation issues for the development of the modeling environment. GeMM was implemented using the Java programming language, which has many development tools and libraries available, in addition to being portable to different platforms. It also has interoperability with other languages, including C and C++, for example.

In order to illustrate how problems can be modeled in GeMM, we considered the Problem of Scheduling Delivery Orders (PSDO) of oil products. We present the optimization model in the following, in order to illustrate how the modeling is made in GeMM. This example shows how users can interact with the environment to model MIPs.

This section presents how GeMM stores and processes information of mathematical programming models and their data in the data structures presented in the previous section, also using PSDO as a guiding example. Subsequently, the implementation features of version control are detailed for the cases of both the mathematical model and the data. Finally, we describe the characteristics of server optimization, an important element of the GeMM architecture. The servers are responsible for receiving requests for optimization of a model and a data case, generating instances of MIPs, and passing them to solvers.

4.1 MATHEMATICAL MODELING

In the context of the PSDO, a company is responsible for delivering products to its customers and ensuring that their demands are met. The delivery is made by road and the objective of the company is to schedule deliveries respecting bounds on demand, customer inventory, and carrying capacity, minimizing the costs involved in these deliveries. It must be solved for a given time horizon.

The mathematical model of this problem was developed based on the inventory management system presented by BERTAZZI *et al.* (2005), whose main feature is the control of customer inventories being done through a distribution center. In this policy of inventory management, the distribution center knows the inventory levels and the demands of their customers. Thus, it can determine the transport policy in order to meet the constraints of storage and demand of its customers, minimizing the total cost of transportation. BERTAZZI *et al.* (2005) show that this policy inventory management significantly reduces transportation costs compared to the traditional model, in which each client manages its inventory alone.

The problem of scheduling the delivery of orders is an integer programming problem. The time horizon is split into discrete periods. Each period is referenced by an index *t* of the set PERIODS. We use the index *c* to refer to a customer of the set CUSTOMERS, and the index *p* to represent a product of the set PRODUCTS. The main decision variables are $Delivery_{pct}$, which represents the quantity in kilograms of product *p* to be delivered to customer *c* in period *t*; and $Stock_{pct}$, which denotes the inventory in kilograms of product *p* in customer *c* in period *t*; and $IfDelivery_{pct}$, a binary variable that indicates whether or not to deliver product *p* to customer *c* in period *t*. The objective of this problem is to reduce the transportation cost, which minimizes the total number of deliveries.

The main input data for this problem are: $Demand_{pct}$, the demand for product *p* of customer *c* in period *t*; $TransportCapacity_{t}$, the maximum amount that can be transported in period *t*; $InitialStock_{pc}$, the initial inventory of product *p* for customer *c*; $MinStock_{pc}$ and $MaxStock_{pc}$, lower and upper bounds on the inventory of product *p* for customer *c*.

To introduce this model in GeMM, the first step is to create a modeling project. At this time, the only attributes required are the name and description of the modeling project, as shown in Figure 11.

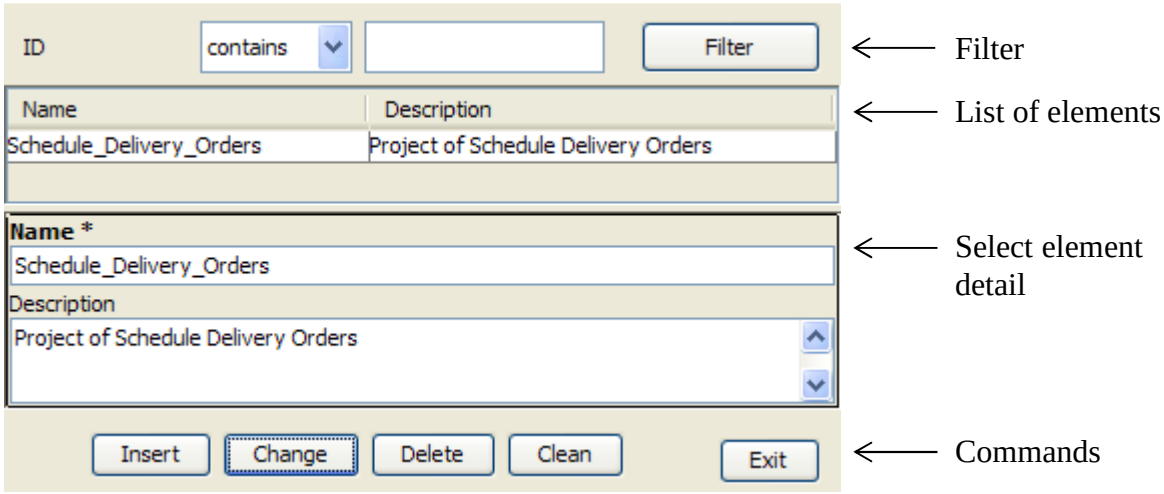

Figure 11: Modeling project

GeMM's screens have a standard format, similar to the screen shown in Figure 11. At the top there are fields to filter the elements in the list that appears below and contains elements of the same type. When selecting an element from the list, the form displays its details. Through the form, it is also possible to edit or to insert new elements in the list. Fields marked in bold and with an asterisk are required, such as the field *Name* in Figure 11. Commands that can be applied to the selected element appear at the bottom of the screens.

Next, the model is built by defining each element. The construction of the model is done through the creation of the elements in the formularies of the GeMM, where each model element has a specific screen where the user can input its attributes. Unlike some modeling languages, such as GAMS (BISSCHOP; MEERAUS, A., 1982) and AMPL (FOURER *et al.*, 1990), in which the mathematical models are textually created through a specific language, GeMM builds the model using a formulary for each one modeling element.

Usually, the first elements to be defined are sets and their respective indexes. Other elements, like parameters, variables, and constraints, may be indexed by indexes defined for these sets. Three sets have been defined for the model of the PSDO: PRODUCTS, CUSTOMERS, and PERIODS, with indexes *p*, *c*, and *t*, respectively, as shown in Figure 12. The attributes of a set are its name, its description, the definition of the data type elements, and their index identifiers. The possible data types are integers and real numbers, strings, and dates. Subsets can also be defined. At this time, only the definition of the sets is done, as their elements are not part of the model, but of the data cases.

Each parameter has a name, a description, and a data type. Again, integers, real numbers, strings, and dates are the available data types. There are two types of parameters: primary, whose values are defined directly in the data case (such as input data) or calculated, whose values are evaluated from expressions that may have constants and other parameters.

The arithmetic operators that can be used in expressions of calculated parameters are shown in Table 1. The analysis of expressions in GeMM is done with the help of the open source library JEP (JEP JAVA, 2011). It uses the operators of Table 1 and allows the addition of new functions by creating a class in Java that implements the interfaces defined by the JEP API.

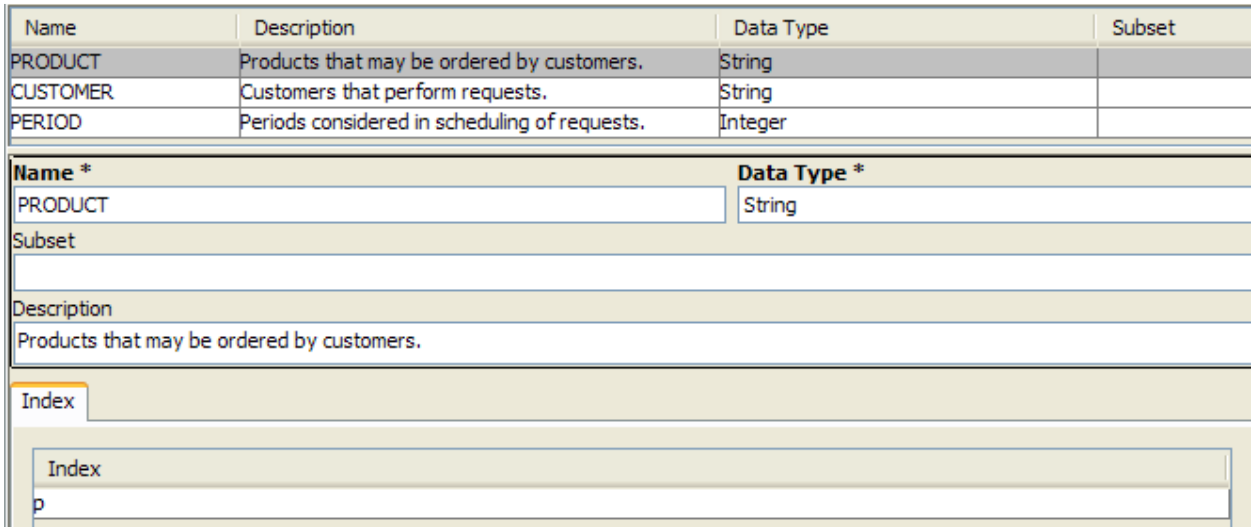

Figure 12: Sets of model

Variables are declared in GeMM, in the same way as parameters. They can be indexed by the indices of the sets and their types may be continuous, integer, or binary. This model involves two types of variables: binary ($IfDelivery_{pct}$) and continuous ($Delivery_{pct}$ and $Stock_{pct}$). The modeler should also define upper and lower limits for each variable.

Table 1: Arithmetic operators

Operator	Description
+	Sum
-	Subtraction
%	Modulo
/	Division
*	Multiplication
^	Power
( )	Parentheses

The main attributes of a constraint are its description, indexes, the definition of the left-hand side (LHS) expression and the right-hand side (RHS) expression. GeMM supports both equality (=) and inequality (≤ or ≥) constraints. With the information of the constraints, GeMM generates its representation in LaTeX (LAMPORT, 2011).

The objective function (maximization or minimization) must have a name, a description and the coefficients of the variables. GeMM also generates the representation in LaTeX of its expression.

This form-based modeling strategy adopted by GeMM has the advantage of not requiring knowledge of a modeling language or a programming language. As it is more structured than a textual language, it guides the modeler through the necessary information, alleviating problems with incomplete model definitions. However, GeMM could be easily adapted to allow importing models defined in textual languages.

4.1.1 GENERATION CONDITIONS

Generation conditions are important for model building (FOURER *et al.*, 1990). Generation conditions are boolean expressions that can use constants, parameters, and indices to determine the condition in which the variables and constraints must be generated for a specific instance of a MIP. Generation conditions can have arithmetic operators and logical and comparison operators, that is presented in Table 2. GeMM also allows the use of generation conditions for variables used in constraints and in the objective function.

Table 2: Logical and comparison operators

Comparison		**Logical**	
Operator	***Description***	***Operator***	***Description***
>	Greater than	!	Not
>=	Greater than or equal to	&&	And
<	Less than	||	Or
<=	Less than or equal to		
==	Equal to		
!=	Not equal to		

4.1.2 MODEL DOCUMENTATION

GeMM allows the automatic generation of a LaTeX documentation with all data provided about the problem, obtained from the information provided by users during the development of a model. The attributes, such as name and description of the modeling elements in GeMM screens, allow the model to be documented during development, avoiding an extra effort for this task.

4.1.3 REPRESENTATION OF THE MATHEMATICAL MODEL

In Section 3 we presented the data structure used to represent mathematical models. This representation is used by GeMM both for processing the application data and for storing it in a database. These structures are completely transparent to GeMM users, because the interaction with the modeling environment is done through forms.

Figure 13 shows the class diagram that represents mathematical models in GeMM. This diagram details how GeMM treats and stores models in a database. For better understanding of the data structures, we used UML class diagrams (BOOCH *et al.*, 2005) with the data types of the Java language. Thus, the description of the treatment of the mathematical model is made by classes, which are mapped to database tables.

The main class is the *Project*, which aggregates all the elements of a model. It is composed of model elements that are represented by the classes *ModelElement* and *ModelElementVersion*, whose difference regards the treatment of versioned attributes. Although version control in GeMM will be treated later in this section, class *ModelElementVersion* is important at this time, since it controls all the attributes of model elements. Despite not having attributes in *ModelElement* class, since all of them belong to *ModelElementVersion*, it is important to identify that different object versions belong to the same model element. The basic attributes of each modeling elements is its name, which must be a unique identifier, and its description, used to document the user model.

The subclasses of *ModelElementVersion* are *Set*, *Indexable*, and *ObjectiveFunction*. The *Set* class represents the sets that can be created in a mathematical model. Each set may have

different indices, represented by the *Index* class. Indices are used to refer to an element of a given set in the model, so an index must have unique identifiers. The *ObjectiveFunction* class has the attribute direction, which determines whether the problem is of maximization or minimization.

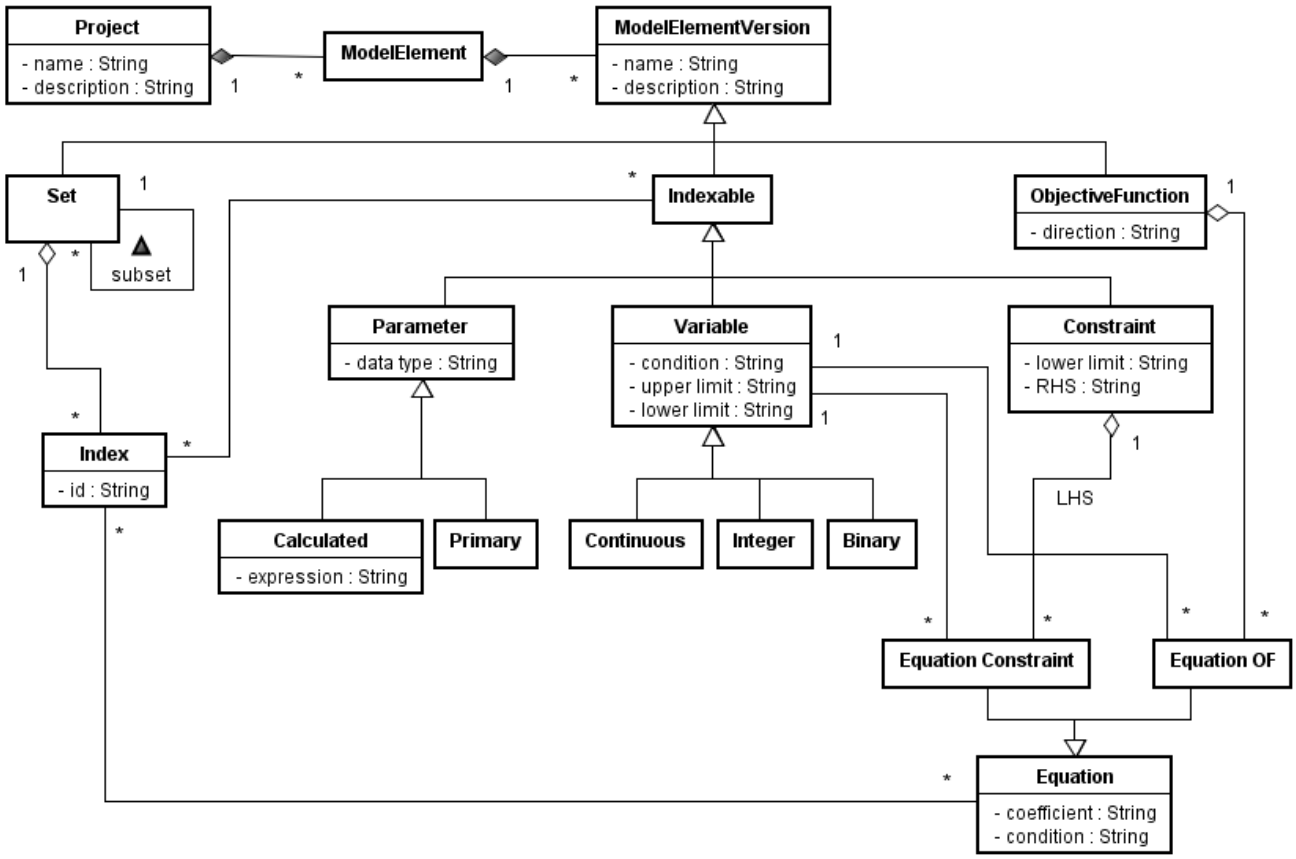

Figure 13: Class diagram of the mathematical model

The *Indexable* class represents the modeling elements that can receive an index in its definition. The relationship between an index and the *Indexable* class should be ordered, since the order of the indices is important during the evaluation of the expressions. Model elements that can receive indices are: *Parameter*, *Variable*, and *Constraint*. The class diagram shown in Figure 13 describes two types of parameters: *Primary* and *Calculated*. It also identifies three types of variables: *Continuous*, *Integer*, and *Binary*. The *Variable* class has expressions for the upper and lower limits, and, optionally, a generation condition. The third *Indexable* type is *Constraint*, which has an expression of its RHS and may also have a generation condition.

Finally, the class *Equation* and its derivatives, *EquationConstraint* and *EquationOF*, represent the expression of the LHS of constraints, as well as the objective function. The *EquationConstraint* class represents the terms of the LHS of the constraints. It refers to an object of the *Variable* class and has an expression of the coefficient of such term in the constraint. Likewise, the *EquationOF* class represents expression terms of the objective function. It refers to an object of the *Variable* class and also has an expression of the coefficient of such term in the objective function. Whenever GeMM generates an instance of a MIP to the solver, both the LHS of the constraints and the expression of the objective function are interpreted as the sum of terms composed of the variables and their coefficients. The coefficients can be expressions containing constants, parameters, and operators.

The terms of the equations may also have a generation condition, indicating a situation where a particular term exists in a constraint or in the objective function. The relationship between *Equation* and *Index*, which represents the indices of a particular variable in the LHS or in the objective function, have the order attribute, since the order of the indexes has significance in modeling.

4.2 DATA CASES

According to FOURER *et al.* (1990), the values of the elements in sets and parameters, which compose a data case, must be combined with a mathematical model to generate an instance of a MIP.

Once model of the problem of scheduling delivery orders has been implemented, GeMM automatically creates the data structure (to store the information) and the screens (for data entry and viewing results). Figure 14 shows the screen to input the values of primary parameters of the model. Through these screens the decision makers can enter and edit parameter values that are necessary for the problem optimization.

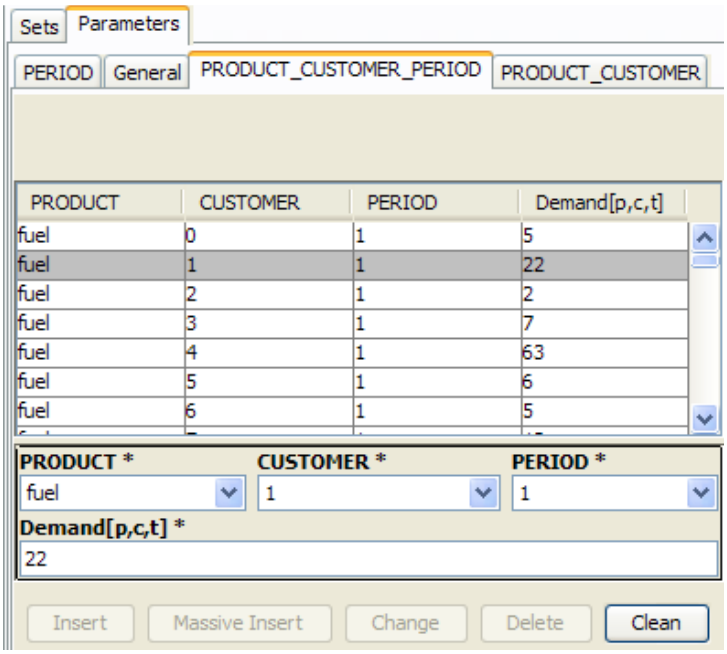

Figure 14: Data of primary parameters

With the screens generated by GeMM, it is possible to provide the necessary data in order to create a problem instance, which is afterwards transferred to a solver. In situations where a huge amount of data must be informed, GeMM allows the massive inclusion of data via the command “Massive Insert”, which can be seen in Figure 14. With this command, the data is imported from text files or spreadsheets. GeMM also generates the screens for displaying the results. It shows the summary of the results, with the values of the objective function and of the variables.

4.2.1 REPRESENTATION OF DATA CASES

To treat the data cases, the structure is capable of working with any model that can be prepared using GeMM. Figure 15 shows the class diagram for representing data cases. This class diagram shows in detail how GeMM implements the data cases and how they are persisted in a database. We also use UML class diagrams with the data types of the Java language to detail the implementation.

As described in Section 3.4, GeMM uses the concept of metamodeling to treat the data case of the mathematical models, since each different model needs its data in a specific structure. Thus, with the use of metamodeling, GeMM has a single database schema to work with any MIP model.

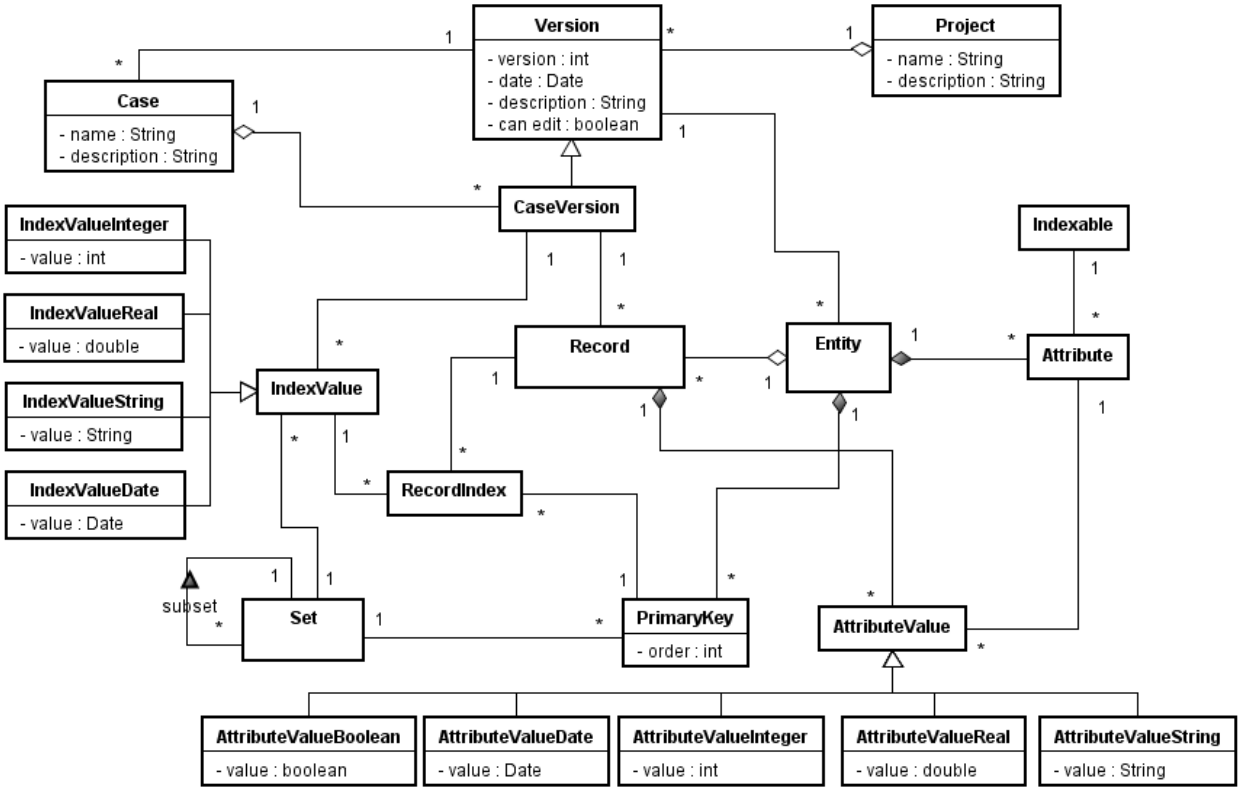

Figure 15: Class diagram of data cases

The main classes are *Case* and *CaseVersion*. *Case* represents a specific scenario created by a user (a decision maker), and has only a name and a description. A data case belongs to a particular version of the modeling project in GeMM. Each data case can have multiple versions, which are represented by the class *CaseVersion*.

The class *IndexValue* represents the possible values for the indices, i.e., the elements belonging to the sets. The values of the elements of the sets are associated with a specific version of a data case. Its subclasses *IndexValueInteger, IndexValueReal*, *IndexValueString*, and *IndexValueDate* are used to implement the data type defined for the set. The *Entity, Attribute*, and *PrimaryKey* classes allow the connection of the mathematical model with the metamodeling data structure. They have no direct relationship with the information of data cases, but represent the metamodeling data structure. Each different combination of indices must be associated with an object of the *Entity* class. In the example of the PSDO there are basically three different entities: the entity of model elements indexed only by *t*, by *p* and *c*, and by the *p*, *c* and *t*.

The *PrimaryKey* class indicates which indices of an entity represent the primary key. It has an attribute *order*, since the order of the indices of the modeling elements is important. Modeling elements that are indexed by the same index, but in a different order, should be associated with different entities. The *PrimaryKey* class relates the *Entity* class with the *Set class*. The *Attribute* class is a modeling element, which is indexed by the combination of specific indices of the entity it belongs. Thus, *Attribute* has a particular relationship with *Indexable* whose subclasses are *Constraint*, *Variable* and *Parameter*, and *Entity*. The *Attribute* class is analogous to a column in a database table.

In the example of the PSDO, the attribute indexed only by *t* is the parameter $TransportCapacity_t$. The primary key of this entity is the index *t*. The attributes indexed by *p* and *c* are the parameters $InitialStock_{pc}$, $MinStock_{pc}$ and $MaxStock_{pc}$. The primary key of this entity is given by indices *p* and *c*. Finally, the attributes indexed by *p*, *c* and *t* are the variables $Delivery_{pct}$, $Stock_{pct}$ and $IfDelivery_{pct}$ and the parameter $Demand_{pct}$. The primary key of this entity is given by indices *p*, *c* and *t*. Figure 16 shows a class diagram that represents the structure for storing the model data of PSDO. GeMM generates theses classes to treat the instances of this model.

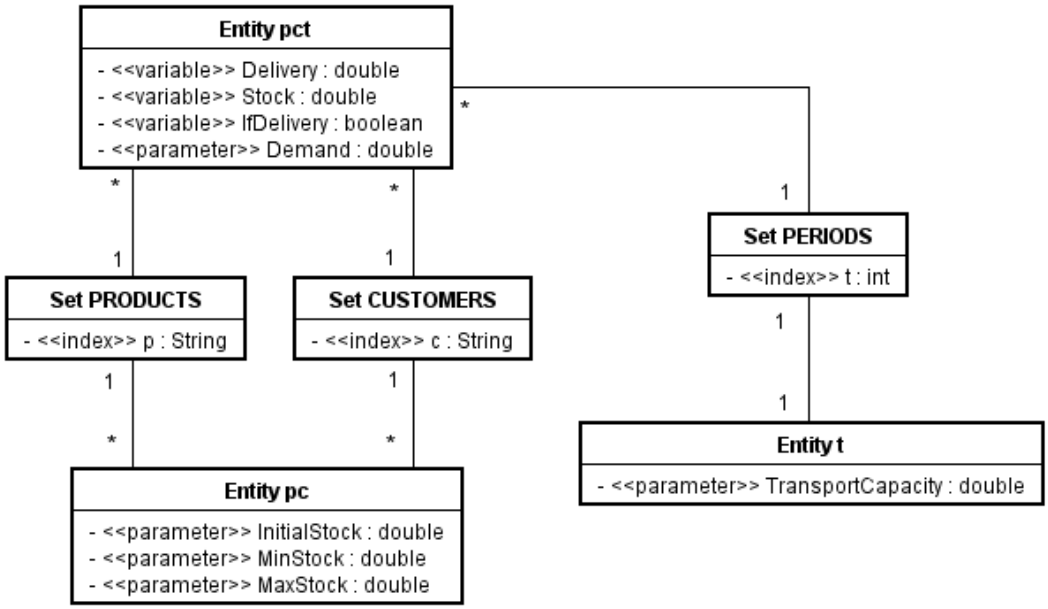

Figure 16: Class diagram of the entities of PSDO

The attributes of metamodel entities, i.e., the values of parameters, variables, and the LHS evaluated constraints, are treated by class *AttributeValue* and its subclasses *AttributeValueBoolean*, *AttributeValueDate*, *AttributeValueInteger*, *AttributeValueReal*, and *AttributeValueString*. Subclasses of *AttributeValue* are used according to the data type defined for the parameters or the type of variables and constraints. The *Record* class is used to group all the attributes that are indexed by the same index value in an entity. An object of this class represents a database record, or a row of a table. The *RecordIndex* class is a ternary relationship between *Record*, *IndexValue*, and *PrimaryKey*, which is used to store the index values that uniquely identify a specific record of an entity.

Classes *IndexValue*, *Record*, *RecordIndex*, and *AttributeValue* of the class diagram shown in Figure 15 are directly related to a version of a data case. The objects of these classes vary with different data cases and versions of the same mathematical model. The classes *Entity*, *PrimaryKey*, and *Attribute* represent the metamodeling data structure to handle the data cases of a model. Thus, the objects of those classes represent the data structure of a given mathematical model. They do not change if the mathematical model is not changed, as the same objects are capable of dealing with any data case.

4.3 VERSION CONTROL

Besides allowing the modeling of linear and integer programming problem and generating screens and data structures to handle the data cases of specific optimization problems, GeMM also provides version control. As discussed before, version control is applied for both mathematical models and their data cases.

4.3.1 VERSION CONTROL OF MATHEMATICAL MODELS

Versioning is handled at fine grain and is based on the entities of the domain. When versioning at fine grain, we are allowed to individually control each model element, assigning independent version numbers when some change occurs. Class *ModelElementVersion*, which can be seen in the class diagram shown in Figure 17, represents the concrete versions of model elements and their attributes that are under version control. The *ModelElement* class represents a particular model element in a mathematical model, regardless of its versions. It clusters the versions of the same model element, as well as all model element attributes are under version control. Thus, if a model element is renamed, for example, GeMM considers it as a change that motivates the creation of a new model element version associated with the same model element, keeping track of the old names.

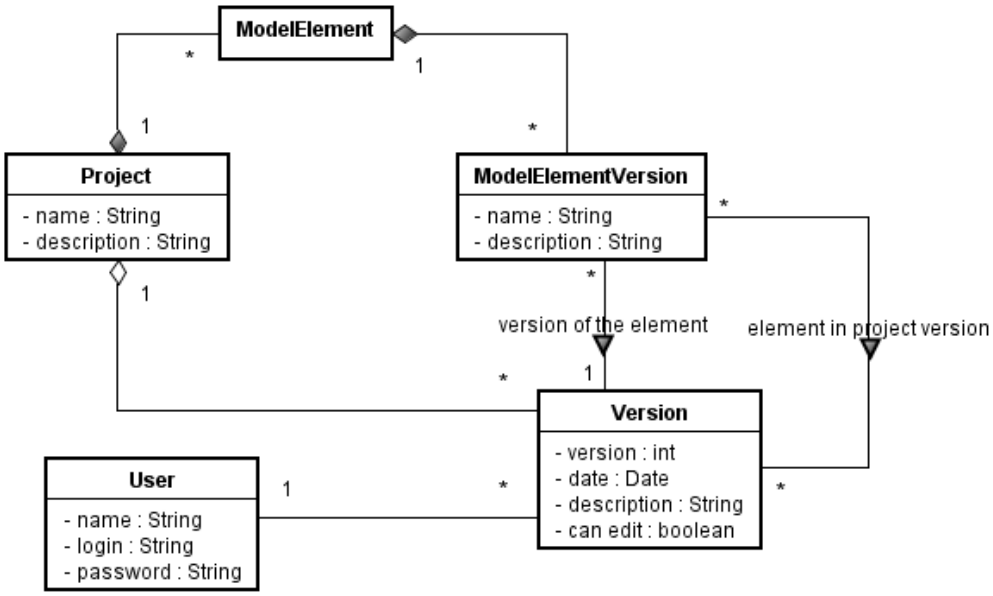

Figure 17: Class diagram version control of mathematical models

The class diagram shown in Figure 17 represents the data structure used to version control of mathematical models. As seen before, the *Project* class is the identification of the optimization problem being solved. It has attributes such as name and description. The details of the mathematical model are associated with a version of the modeling project, represented by class *Version*. Each version is associated with a GeMM user. The class *ModelElementVersion* has two relationships with the class *Version*: one that indicates the version of the model element and another that indicates which versions of the model project contains the model element. Thus, GeMM considers model elements as unities of versioning (MURTA *et al.*, 2007), once each model element within a modeling project has its own version history. The association named *"element in project version"* in the diagram of Figure 17 indicates which versions of model elements belong to a version of the modeling project. The version of the model element itself is represented by the association *"version of the element"* in the same diagram. Thus, given a version of the modeling project, GeMM can identify the associated mathematical model and enables the creation of data cases for it.

GeMM implements pessimistic concurrency control on mathematical models using the *"can edit"* attribute of *Version class*. This attribute indicates when the version of a modeling project may or may not be edited. If it can be edited, the change can only be made by the user who created the version. Any other user cannot edit the project until the current user commits or

discards his version. Each modeling project can have only one editable version at the same time, since the current version of GeMM does not address parallel editing for the same modeling project. When the modeling project does not have any editable version, any user can create a new version. If this is the first version of the modeling project, GeMM creates an empty version. Otherwise, GeMM creates a new version associated with the versions of the model elements that belong to the previous project version. This way, it inherits all previous model elements and allows the user to perform modifications.

The version numbers of the model elements are equal or less than the version number of the modeling project, once the creation or edition of a model element implies the edition of the modeling project. This strategy is also adopted in other conventional version control systems, such as Subversion (THE APACHE SOFTWARE FOUNDATION, 2011). GeMM uses two basic commands: “commit” and “discard”. In the case of the “commit” command, GeMM understands that the version is finished and changes its “can edit” attribute to false, indicating that any user can only edit the model element by creating a new version. Thus, the version that was committed is unchangeable. In the case of the “discard” command, GeMM deletes the version of the modeling project that is under edition. This command can only be applied over versions that are editable and is restricted to the user who created it. Versions of modeling projects that have been committed can no longer be altered, removed, or discarded, ensuring consistency to the change history of mathematical models. A user can always make changes over modeling projects by creating a new version.

4.3.2 VERSION CONTROL OF DATA CASES

GeMM also implements versioning of data cases that can be created for the mathematical models. The class diagram shown in Figure 18 describes the data structure that implements the versioning of data cases. The *CaseVersion* class represents a version of a given data case. In the same way we do with mathematical models, all information contained in the data cases are associated with a particular version. Consequently, the unity of versioning is a data case, and then the data within a case does not have individual version numbers and are always associated with a particular version of a data case. The *Case* class represents only the identification of a case of a mathematical model. Its attributes are just name and description, which are not versioned. Similarly, the *CaseVersion* class is to *Case* as *Version* is to the *Project*.

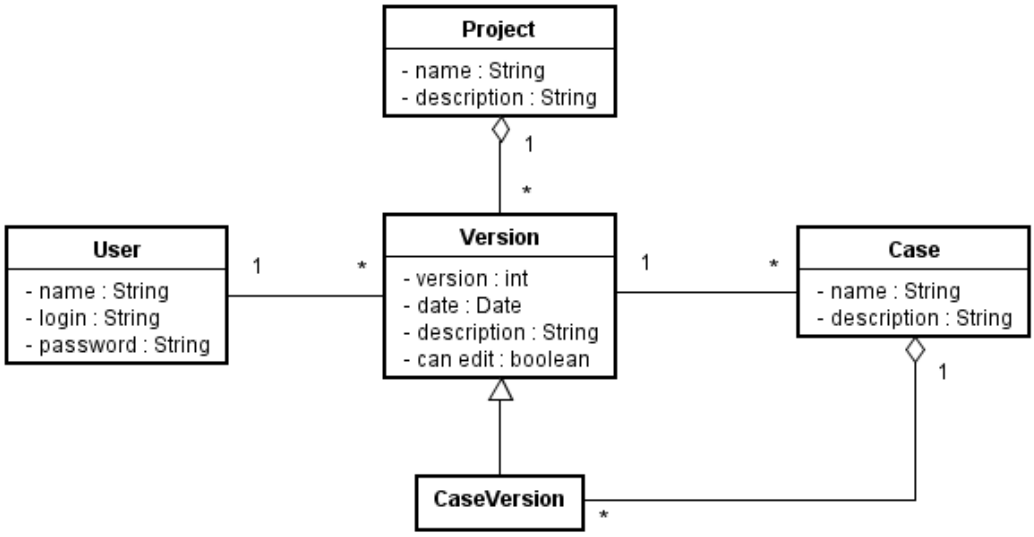

Figure 18: Class diagram version control of data cases

Version control and concurrency are done in the same way as for mathematical models. The “can edit” attribute inherited from *Version* indicates when a version of the data case can be edited or not. When the value of this attribute is true, only the user who created the version can edit, commit, or remove it. When the user commits the version of the data case, another user can edit it, creating a new version of the case. This way, a new version is always necessary to edit the data case. If the case does not have any version, GeMM creates a blank version. Otherwise, GeMM creates a new version referencing all data presented in the last existing version. The removal of the versions of data cases can only be done by the user who created them and only before committing, because once it is committed it can no longer be deleted or changed, thus maintaining the historical data associated with mathematical models. This mechanism enforces that only the latest version can be removed or changed, if it has not already been committed.

4.4 OPTIMIZATION SERVERS

The optimization servers are responsible for performing the optimization requests for the MIP modeled using GeMM. The process starts when GeMM receives a request from a user to optimize a version of a data case. It generates an instance of an MIP from the mathematical model and data case, transfers this instance to a solver, receives the problem solution, and returns this solution to the user. According to the architecture shown in Figure 1, the optimization servers can be deployed on separate computers. GeMM also allows a monolithic installation, in which the modeling application and the optimization servers are in the same computer.

Independently of the deployment strategy, the optimization server is responsible to handle requests of optimization and to communicate with a solver. However, when the modeling application and the optimization server are deployed in different machines, requests to solve data cases are made using web services through the network. The web service is used only to notify the server that a case should be optimized. The information regarding the data case and its mathematical model are obtained directly from the database, as shown in Figure 19. The results are also stored in the database.

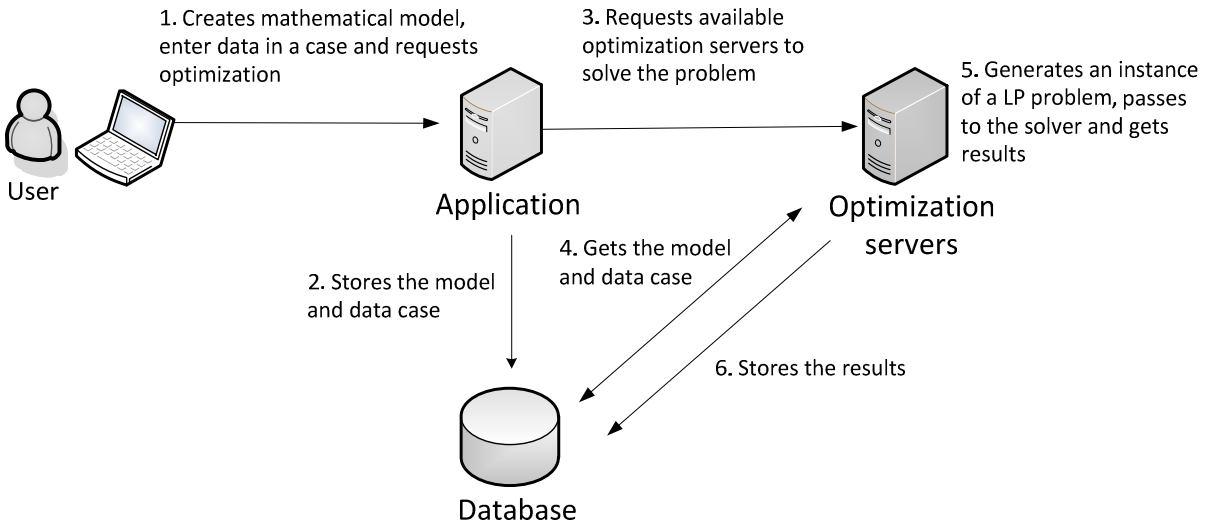

Figure 19: Scheme for performing optimizations

Figure 20 shows a class diagram that represents an instance of an MIP. GeMM uses the mathematical model, whose structure is depicted diagrammatically in Figure 13, and the data case, whose structure is depicted diagrammatically in Figure 15, to generate an instance of the MIP. Although the class diagram shown in Figure 13 has some classes with the same name of the

class diagram shown in Figure 20, they are different, since the first diagram shows the definition of the mathematical model and the second is an instance of an MIP.

With the data structure shown in Figure 20, GeMM can send the data to the solvers through their API, when it is installed and configured in the same machine, or generate files in the format developed for CPLEX, but also used by other solvers.

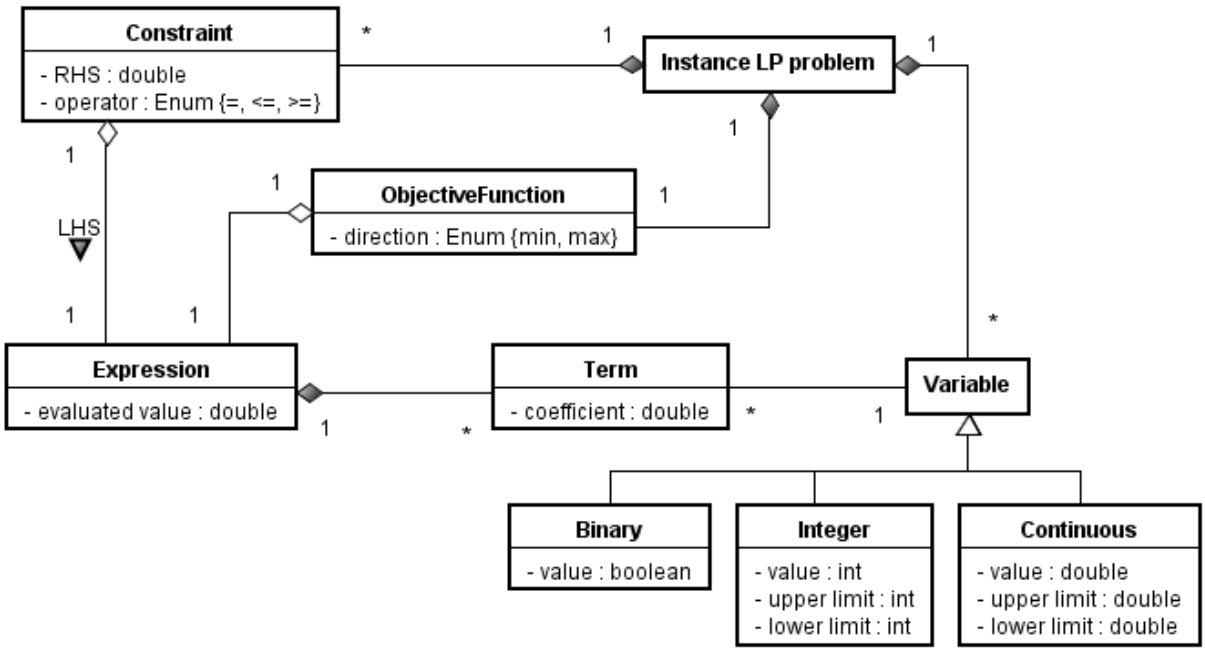

Figure 20: Class diagram for representing an instance of a LP problem

5 CONCLUSION

The main goal of this paper is to describe the development of the GeMM environment for modeling linear and integer programming problems. It aims not only to treat the modeling activity, but also to support the management of the life cycle of mathematical models and associated data.

According to MAKOWSKI (2005), features such as version control and configuration management of the mathematical models are poorly explored, although they are quite relevant. We could also conclude that characteristics such as the separation of data and model, the use of databases, the integration with other software, the use of graphical user interfaces for data entry and results analysis (both performed by decision makers), and the efficient communication with solvers are critical in developing this type of application.

The main contributions of this work, compared with the existing modeling tools, are its architecture in a client-server style, which allows the sharing of information and interaction among modelers and decision makers, and its ability to control the evolution of mathematical models and data cases through version control. GeMM therefore presents two special differential features: GeMM is a multiuser environment (in which people can work on different versions through an integrated environment) and is not concerned by conflicting changes (thanks to version control). Other modeling tools do not support these features, requiring users to externally perform this control.

The use of version control brings concepts of Software Engineering to mathematical modeling. It helps on managing the life cycle of models and data used to obtain the results,

which are essential in the decision making process of an organization. Such control leverages auditing the results, since all results are associated with a particular version of a model and a particular version of a data case. Moreover, GeMM automatically generates an application from a mathematical model for the input data and results analysis. The proper treatment of data cases and their versions allows GeMM users to perform various analyses on models without modifying their definitions.

The GeMM architecture also segregates models, data cases, and solvers. This separation is recommended by RAMIREZ *et al.* (1993). The use of metamodeling to store information of mathematical models and the associated data in databases also provides contributions to the treatment of general persistence of mixed integer programming problems. FOURER (1997) and DUTTA and FOURER (2008) use relational databases to store specific problems, with structures that meet a given problem or class of problems. In these cases, for each model, a new database schema must be created. GeMM uses the same database schema for storing different data models. Despite being in the same structures, they are logically separated.

An important contribution of this work is the structure, shown in Figure 13, to treat linear and integer programming models. This structure allowed a general representation of the MIPs and was the basis for the data processing through metamodeling and for the versioning of mathematical models. Through this data structure, GeMM makes problem modeling independent of the interface with users. We adopted formularies to implement such interface in the current version of GeMM, but this interface can be replaced since the representation of the problem is made by the structure shown in Figure 13.

Another important feature provided by GeMM is the generation of the documentation of mathematical models in the LaTeX format. The mathematical representation of the objective functions and constraints proved to be very useful to understand and document models. This documentation can be exported as a PDF file, which completely describes the mathematical model.

REFERENCES

BAZARAA, M., **Linear programming and network flows**. Wiley, New York, 1977.

BERTAZZI, L.; PALETTA, G.; SPERANZA, M. G., “Minimizing the total cost in an integrated vendor—managed inventory system”, **Journal of Heuristics** 11 (2005), 393–419.

BISSCHOP, J.; MEERAUS, A., “On the development of a general algebraic modeling system in a strategic planning environment”, **Mathematical Programming Studies** 20 (1982), 1–29.

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J., “The Unified Modeling Language User Guide”. Addison-Wesley, Upper Saddle River, 2005.

BROOK, A.; KENDRICK, D.; MEERAUS, A., “GAMS, A User’s Guide”, **ACM SIGNUM Newsletter** 23 (1988), 10–11.

CONRADI, R.; WESTFECHTEL, B., “Version models for software configuration management”, **ACM Computing Surveys** 30 (1998), 232–282.

DUTTA, G.; FOURER, R., “Database structure for a class of multi-period mathematical programming models”, **Decision Support Systems** 45 (2008), 870–883.

ESTUBLIER, J., “Software configuration management: a roadmap”, **Proceedings of the 22nd International Conference on Software Engineering,** Limerick, pages 279-289, 2000.

ESTUBLIER, J., “Objects control for software configuration management”. **Lecture Notes in Computer Science** 2068 (2001), 359-373.

FOURER, R., “Database structures for mathematical programming models”, **Decision Support Systems** 20 (1997), 317–344.

FOURER, R., “Predictions for web technologies in optimization”, **INFORMS Journal on Computing** 10 (1998), 388–389.

FOURER, R., “Software survey: Linear programming”, **OR/MS TODAY** 38 (2011), 60–69.

FOURER, R.; GAY, D. M.; KERNIGHAN, B. W., “A modeling language for mathematical programming”, **Management Science** 36 (1990), 519–554.

FOURER, R.; MA, J.; MARTIN, K., “Optimization services: A framework for distributed optimization”, **Operations Research** 58 (2010), 1624–1636.

FREIRE, J.; KOOP, D.; SANTOS, E.; SILVA, C., “Provenance for computational tasks: A survey”, **Computing in Science & Engineering** 10 (2008), 11–21.

GEOFFRION, A. M., “An introduction to structured modeling”, **Management Science** 33 (1987), 547–588.

GEOFFRION, A. M., “Computer-based modeling environments”, **European Journal of Operational Research** 41 (1989), 33–43.

HAVERLY, C. A., “OMNI model management system”, **Annals of Operations Research** 104 (2001), 127–140.

IBM CORPORATION, **IBM mathematical programming language extended 370 (MPSX / 370)**, Program Reference Manual, 1975.

JEP JAVA. **Jep Java - Math Expression Parser Open Source**, online reference available at http://sourceforge.net/projects/jep/, last access in July 13, 2014.

JEUSFELD, M. A.; JOHNEN, U. A., “An executable meta model for re-engineering of database schemas”, **Proceedings of the 13th International Conference on the Entity-Relationship Approach**, Manchester, pages 533–547, 1994.

KITCHENHAM, B., **Procedures for performing systematic reviews**, Technical Report, Keele University, Staffordshire, 2004.

LAMPORT, L., **LaTeX**, online reference available at http://www.latex-project.org/, last access in July 13, 2014.

LEE, J. S., “A model base for identifying mathematical programming structures”, **Decision Support Systems** 7 (1991), 99–105.

MAKOWSKI, M., “A structured modeling technology”, **European Journal of Operational Research** 166 (2005), 615–648.

MATURANA, S.; FERRER, J. C.; BARAÑAO, F., “Design and implementation of an optimization-based decision support system generator”, **European Journal of Operational Research** 154 (2004), 170–183.

MENS, T., “A state-of-the-art survey on software merging”, **IEEE Transactions on Software Engineering** 28 (2002), 449–462.

MURPHY, F. H.; STOHR, E. A.; ASTHANA, A., “Representation schemes for linear programming models”, **Management Science**. 38 (1992), 964–991.

MURTA, L.; OLIVEIRA, H.; DANTAS, C.; LOPES, L.; WERNER, C., “Odyssey-SCM: An integrated software configuration management infrastructure for UML models”, **Science of Computer Programming** 65 (2007), 249–274.

PRESSMAN, R., **Engenharia de software**, McGraw-Hill, São Paulo, 2006.

PRUDÊNCIO, J. G.; MURTA, L.; WERNER, C.; CEPÊDA, R., “To lock or not to lock: That is the question”, **Journal of Systems and Software** 85 (2012), 277-289.

RAMIREZ, R.; CHING, C.; ST. LOUIS, R. D., “Independence and mappings in model-based decision support systems”, **Decision Support Systems** 10 (1993), 341–358.

THE APACHE SOFTWARE FOUNDATION, **Subversion**, online reference available at http://subversion.apache.org/, last access on July 13, 2014.