Ricardo Diogo Martins Teixeira

Licenciado em Ciência e Engenharia Informática

Controlled Specification and Generation of Spreasheets

Dissertação para obtenção do Grau de Mestre em Engenharia Informática

Orientador: Jácome Cunha,
Prof. Auxiliar, Universidade Nova de Lisboa

Co-orientador: Vasco Amaral,
Prof. Auxiliar, Universidade Nova de Lisboa

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Dezembro, 2016

Controlled Specification and Generation of Spreadsheets

Abstract

Spreadsheets are one of the most popular programming systems in the world, widely used both by individuals as well as large companies. Spreadsheets popularity is due to characteristics such as their low entry barrier, their simple visual interface, their availability on almost any platform, and their multipurposness.

However, spreadsheets' flexibility, which is not accompanied with effective error prevention mechanisms, makes them extremely error prone, as indicated by several studies. Moreover, since spreadsheets often take an important role in business decisions, they can have an enormous impact on critical components of businesses, as numerous audit processes have shown.

As a result, it is necessary to create mechanisms that allow the specification of spreadsheets' structures and their respective business logic correctly, as well as to maintain the correctness of their data during the spreadsheets' life cycle.

The aim of this thesis is to design a Domain Specific Language (DSL) that uses the Model-Driven Development approach (MDD) to address the stated needs. This DSL consists of a UML-based visual language that covers the most common spreadsheets patterns and allows to define arbitrary data constraints. The spreadsheets generated from this DSL will guide the users so they make less mistakes than before.

As a case study, we have used our approach to derive a spreadsheet to a governmental institution. Those in charge were quite happy with the new spreadsheet as it adds several data constraints to the one they used to have.

Keywords: Spreadsheet, Model-Driven Development, Domain Specific Languages, UML

Resumo

Folhas de cálculo são dos sistemas de programação mais populares no mundo, amplamente utilizadas tanto por indivíduos, bem como por instituições. A popularidade das folhas de cálculo deve-se a características tais como a facilidade com que se começa a utilizá-las, a sua interface visual simples, a sua disponibilidade em praticamente qualquer plataforma e o seu multipropósito.

No entanto, a flexibilidade das folhas de cálculo não é acompanhada de mecanismos eficazes de prevenção de erros, tornando-as extremamente sujeitas a erros, como indicado por vários estudos. Tendo em conta que as folhas de cálculo têm frequentemente um papel importante em decisões de negócio, elas podem ter um enorme impacto sobre componentes críticos de empresas ou outras instituições, como tem sido demonstrado por numerosos processos de auditoria.

Como resultado, é necessário criar mecanismos que permitam especificar estruturas de folhas de cálculo e sua respetiva lógica de negócio corretamente, bem como manter a correção dos seus dados ao longo do ciclo de vida da folha de cálculo.

O objetivo do trabalho desta tese é a conceção de uma Linguagem de Domínio Específico (Domain-Specific Language - DSL) que usa a abordagem de Desenvolvimento Dirigido a Modelos (Model Driven Development - MDD) para tratar das necessidades mencionadas. Esta DSL consiste numa linguagem visual baseada em UML que cobre os padrões de folhas de cálculo mais comuns e que permite definir restrições arbitrárias sobre os dados da folha de cálculo. As folhas de cálculo geradas a partir desta DSL irão guiar os utilizadores de forma a que estes não cometam erros.

Como caso de estudo, nós usámos a nossa abordagem para derivar uma folha de cálculo para uma instituição governamental. Os responsáveis ficaram bastante satisfeitos com a nova folha de cálculo, tendo em conta que a mesma adiciona várias restrições de dados à folha de cálculo que estava, até então, a ser usada.

Palavras-chave: Folha de cálculo, Model-Driven Development, Domain Specific Languages, UML

Contents

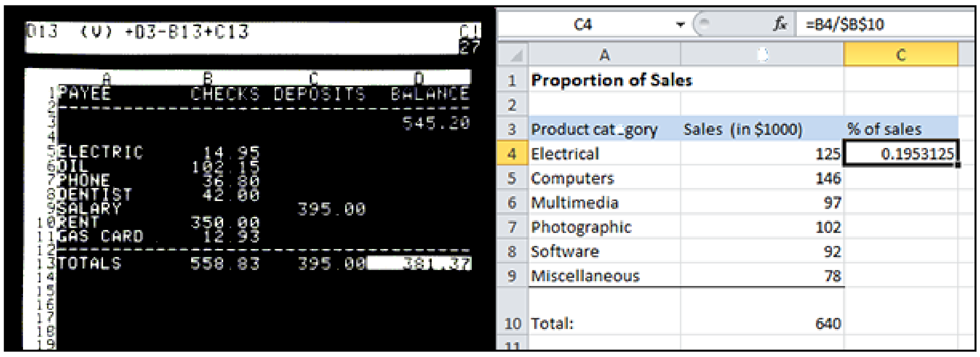
List of Figures

List of Tables

1

Introduction

Spreadsheets are electronic documents in which data is arranged in the rows and columns of a grid and can be manipulated and used in calculation [1].



```
D13  (V) +D3-B13+C13                          C!
                                              27
          A           B        C         D
 1PAYEE            CHECKS DEPOSITS  BALANCE
 2------------------------------------------
 3                                   545.20
 4
 5ELECTRIC          14.95
 6OIL              102.15
 7PHONE             36.80
 8DENTIST           42.00
 9SALARY                    395.00
10RENT             350.00
11GAS CARD          12.93
12------------------------------------------
13TOTALS           558.83   395.00   381.37
14
15
16
17
18
19
```

C4 =B4/B10

	A	B	C
1	**Proportion of Sales**		
2			
3	Product cat_gory	Sales (in $1000)	% of sales
4	Electrical	125	0.1953125
5	Computers	146	
6	Multimedia	97	
7	Photographic	102	
8	Software	92	
9	Miscellaneous	78	
10	Total:	640	
11			

Figure 1.1: VisiCalc spreadsheet versus nowadays Excel spreadsheet

The concept was introduced in the early 1980's with the first spreadsheet tool called VisiCalc [1], followed by Lotus 1-2-3 [1] which added integrated charting, plotting and database capabilities. Later in that same decade, the concept was matured as many companies introduced spreadsheet products with Microsoft (MS) joining the fray with the innovative Excel [1] – one of the most popular spreadsheet tools nowadays and an industry standard for spreadsheets. Excel was one of the first spreadsheets to use a graphical interface with pull-down menus and a point and click capability using a mouse pointing device, forming the basis of the modern spreadsheets. In Figure 1.1 we can see a visual comparison between the VisiCalc and one the latest versions of MS Excel.

Due to characteristics such as the initial small learning effort associated with the use of spreadsheets – it is considerable easy to create a simple spreadsheet without substancial programming for any technology user -, their friendly graphic user interface, their availability on almost any platform and especially their general purpose as a result of the spreadsheets' high flexibility, spreadsheets became one of the most popular programming systems. Being the first "programmer in a box" to come along for technology users, spreadsheets are widely used both by individuals to cope with simple needs like tracking personal finances/expenses, training plans, to-do lists, supplier databases, or any purpose that requires input of data and/or performing calculations; as well as institutions as integrators of complex systems and as support for informing business decisions especially in areas like marketing, business development, sales, and finance. In spite of the existence of business intelligence applications, workers do financial analysis by extracting data from those back-end systems and importing it into spreadsheets because there are easier to use and the corresponding tool is installed on every computer separately.

Spreadsheets are established in institutions. In fact, several studies over the years have shown, clearly and unmistakably, that spreadsheets have a massive presence in the labour force, and that they are used to make important decisions:

- A study made in 2003, shows that 47% of mid-size companies use stand-alone spreadsheets for planning and budgeting [2].

- In 2004, a study in which more than a hundred finance executives were interviewed, of 14 technologies discussed, spreadsheets were one of most widely used technologies [3].

- A study which took place in 2005 states that about 23 million American workers - about 30% of workforce - use spreadsheets [3].

- Finance intelligence firm CODA, estimate that 95% of all U.S.A. firms use spreadsheets for their financial reporting [3].

Despite business intelligence applications have become quite sophisticated, offering more efficiency and report capabilities far beyond spreadsheets functionalities such as performance indicators, their adoption by companies is not exactly taking place as one would have expected. Barriers already mentioned such as the necessary training of the employees and their reluctance to use new technology eventually can be overcome, however, cost and integration problems remain the top barriers. According to a study performed by CFO.com [4] which asked finance companies what barriers prevented them from implementing new technologies, 74% chose integration with existing systems, while 71% selected cost. When asked about the importance of spreadsheets in the future, the most dominant response was that spreadsheets would maintain the same important role (71%), followed by the answer which stated that spreadsheets would gain more importance (20%). So, to put it simply: spreadsheets are widely used in the world of business and are here to stay.

1.1 Motivation

Regardless of the enormous popularity and proliferation of spreadsheets, their use is not at all without problems – the flexibility spreadsheets offer is not accompanied by effective mechanisms for error prevention.

Numerous audit studies to real-world spreadsheets show that erroneous spreadsheets are largely common – it was estimated that 94% of real-world spreadsheets contain errors [5]. Those errors not only are significant regarding occurrence, but also in importance. Several cases of spreadsheet errors with impact on critical components of businesses which resulted in huge money loss and career damage have been known in the past decades. Many of them were documented and made available by the European Spreadsheet Risk Interest Group (EuSpRIG) [6] in the "Horror Stories" section [7].

Errors in spreadsheets occur during their creation and are made as well when users perform modifications on them. That happens because the people who usually create and modify spreadsheets do not completely understand their functionality, because they are not programmers and thus lake programmers' knowledge of how to structure software. Therefore, they do not program spreadsheets in a methodical and well-documented way, ignoring the programming conventions. This situation is particularly critical in the spreadsheets world, since spreadsheet systems encourage the use of "spaghetti" logic, where, for instance, cells point to cells that point to cells, which can grow into random networks of calculation logic. They are prone to several off-by-one errors which in general are difficult to verify, since they do not have any abstractions like modern programming languages do. Moreover, it does not exist a clear separation between data and computation in spreadsheets, and the immediate visual feedback mechanism makes traditional coding and program compilation/execution steps indistinguishable from each other. All these factors plague the creation of well-structured spreadsheets and their maintainability, making it hard and prone to errors.

Nonetheless, the improvements in spreadsheets over the last years did not involve considerable enhancements regarding error-prevention, but have primarily just involved the improvement of the typesetting capabilities of spreadsheets and the creation of add-ins for data manipulation. Moreover, proposed research techniques for error-prevention have not reached commercial spreadsheet applications.

1.2 Approach

Model-Driven Engineering (MDE) is a development methodology in which abstract models of software systems are created and systematically transformed to concrete implementations, as they support visualization, analysis, communication, and validation of different aspects of a system before its implementation [8].

In the spreadsheets context, MDE can be used to specify and validate spreadsheets, as well as validate constraints on them to prevent errors and, additionally, to offer a much better understanding of large spreadsheets through small, precise and compact models.

However, the existing models to spreadsheets solutions do not fully address real-world operational spreadsheets – they cover a limited kind of spreadsheets [9]; or the proposed model lack expressiveness [10] –, and, moreover, none of them possesses mechanisms to apply arbitrary constraints on spreadsheets.

The work proposed in this dissertation aims to provide a model-driven approach to the controlled specification and generation of spreadsheets which conforms with all types of industrial spreadsheets.

With that intent we analyzed the spreadsheets composing two large repositories of real-world spreadsheets made public, also taking in to consideration a previous work analysis made on them [11, 12], to systematize and document the most common spreadsheet patterns which this work presents.

Taking into account those patterns, we defined a Domain Specific Language (DSL) that uses the Model-Driven Development approach (MDD) to address the stated needs. This DSL consists of a UML-based class diagram visual language, complemented with OCL-based invariants to impose business logic constraints. The choice for a UML conceptual model resides in the fact that UML class diagrams are one the most proliferated conceptual models and have a high level of understanding during data models maintenance than other well-known conceptual models, e.g., Entity-relationship diagrams [13].

1.3 Contributions

Upon the approach of this work described in section 1.2, the contributions of this work are:

- The systematization and documentation of the most common patterns in real-world spreadsheets.
- A metamodel of those patterns, extending the UML's metamodel.
- A UML-based class diagram DSL to specify spreadsheets.

- A generation technique to produce spreadsheets from these specifications.

- A parser-generator of OCL-based invariants to spreadsheet formulas, for spreadsheet data validation.

- A tool that implements the techniques this work presents.

- Moreover, based on this dissertation's work, we have a paper: "Ricardo Teixeira, Vasco Amaral, *On the Emergence of Patterns for Spreadsheet Data Arrangements*, In Proceedings of the 3rd International Workshop on Software Engineering Methods in Spreadsheets, 2016", to appear.

1.4 Document Structure

This dissertation is organized as follows:

Chapter 2 introduces the general techniques in which our work is based, gives a perspective of the literature concerning spreadsheet error taxonomies, and describes the state of the art regarding the Model-Driven Spreadsheet Development.

Chapter 3 contains previous research studies regarding spreadsheet patters, and our catalog of spreadsheet data arrangement patterns.

Chapter 4 describes our approach based on the patterns presented in Chapter 3 and the spreadsheet errors addressed by it.

Chapter 5 describes the prototype tool developed which implements the techniques presented in Chapter 4.

Chapter 6 describes the case study of this dissertation. A real-world spreadsheet is presented and a description of how our work can overcome some of its deficiencies is made.

Chapter 7 concludes this dissertation with remarks on the work done and exposing directions for future work.

2

State of the Art

In this Chapter we make an introduction to Model-Driven Development and Domain Specific Languages. Moreover, we describe the state of the art regarding spreadsheet error taxonomies, and then we perform a revision of the state of the art in Model-Driven Spreadsheet Development.

2.1 Model-Driven Development

Typically, a *model* refers to a description of something. In Software Engineering, a model is an artifact that describes a system through various graph-based diagram types. Models are formulated in modeling languages, such as UML, usually visually rendered [14]. A metamodel is a model of a model that typically defines the language and processes from which to form a model, according to rules, properties and relations [14]. A model expressed with a metamodel is called an instance of that metamodel.

Using models as the primary artefacts of the software development process, Model-Driven Development (MDD) [14, 15] is a software development paradigm that aims to automate the programming tasks, some of them quite complex despite routine, by raising the level of abstraction even further – in MDD, all knowledge is explicit and can be modeled, from requirements to code, ensuring compliance and consistency between models.

There are several advantages that MDD brings to software development, such as productivity and maintainability improvement; up-to-date documentation – since the model is the documentation; domain-specific validations that are executed at design-time which can have associated domain-specific error messages; and development is less error-prone. In fact, these advantages are interrelated, e.g., fewer errors favors a higher productivity, and a better documentation provides higher maintainability [15].

2.2 Domain Specific Languages

Domain Specific Language (DSL) is a programing/specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [16, 17].

A DSL can be graphic or textual. There are three approaches for the implementation of textual languages. In the first approach a grammar can be specified along with the creation of parser-generator for conversion. The second approach is to use properties from another language to emulate the capabilities of a DSL. Lastly, the third common approach is to use XML with a schema to help validate documents and provide syntax coloring and autocompletion.

A model can be expressed using a DSL, which is an approach for the implementation of a graphic DSL. DSLs can follow the MDD paradigm for modeling and can be used to formalize the application structure, behavior and requirements, imposing domain-specific constraints and performing model checking that can detect and prevent many errors in early stages of the development process.

Regarding the modeling paradigm, the development cycle of a DSL has several steps that should be followed [17]. The first one is the Domain Analysis, where since the DSL has to take into account the particularities of the domain of the DSL to be developed. This analysis is carried out using various sources of information, such as the study of technical documentation, existing implementations, language requirements and through dialogues with domain experts. Then, based on the gathered information, a metamodel is developed, in which are specified the syntax and semantics of the DSL. The third step is the actual implementation of the DSL, based on the metamodel previously defined. The final step is the validation of the DSL in the context of the domain. The whole process is an iterative one, since to be flawless, a DSL must be redesigned several times.

2.3 Spreadsheets Error Taxonomies

Spreadsheet errors can be classified in a variety of ways, for distinct focuses and purposes. Generally accepted taxonomies of spreadsheet errors distinguish errors by effect, cause, stage, form, or risk [18].

A classification by the errors effect on the spreadsheet were purposed by several researches [19, 20, 21, 22, 23, 24], categorizing spreadsheet errors in two general groups: *quantitative* errors and *qualitative* errors. Quantitative errors are errors that produce an immediate incorrect result or logic in the spreadsheet, while qualitative errors are associated to poor spreadsheet design that increases the probability of eventual qualitative errors occurrence during operational use of the spreadsheet.

Galletta et al. [19] differentiated quantitative spreadsheet errors by cause, dividing errors into domain errors – the ones that occur due to lack of domain knowledge as, for instance, choosing a wrong formula to implement an algorithm; and device errors – errors involving using the computer as typing or pointing errors.

Panko and Halverson [20] in their taxonomy also presented an error classification based on the error source of quantitative errors, which included three types of errors: the mechanical errors – the same as the device errors of Galletta –, the logical errors, that included the domain errors of Galleta, and the pure logical errors, which consists of incorrect use of logic or mathematics in general –, and the omission errors, which are parts of the spreadsheet model that were mistakenly left out.

Rajalingham et al. [22] – also split quantitative errors in mechanical and logical errors, naming them accidental errors and reasoning errors, respectively –, introduced in their taxonomy the distinction between developer and end-user errors, that is, the spreadsheet life cycle stage where the errors occur. Moreover, Panko and Aurigemma [24] revisited the Panko-Halverson taxonomy, adding that distinction.

Errors could also be distinguished by their form or context. Besides the cell level, an error can occur, for instance, at the algorithm level, at the module level, and at the level of the spreadsheet as a whole [24].

Another way of categorizing errors is by the risks they pose. Madahar et al. [23] presented a taxonomy for categorization of spreadsheet use and the level of risks involved, proposing a three-dimensional model with the following dimensions: Magnitude of risk – the severity of the consequences of errors within spreadsheets; Dependency – how fundamental the spreadsheet is for the company (operational, analytical/management, or financial); and Urgency – deadlines associated with the spreadsheet use.

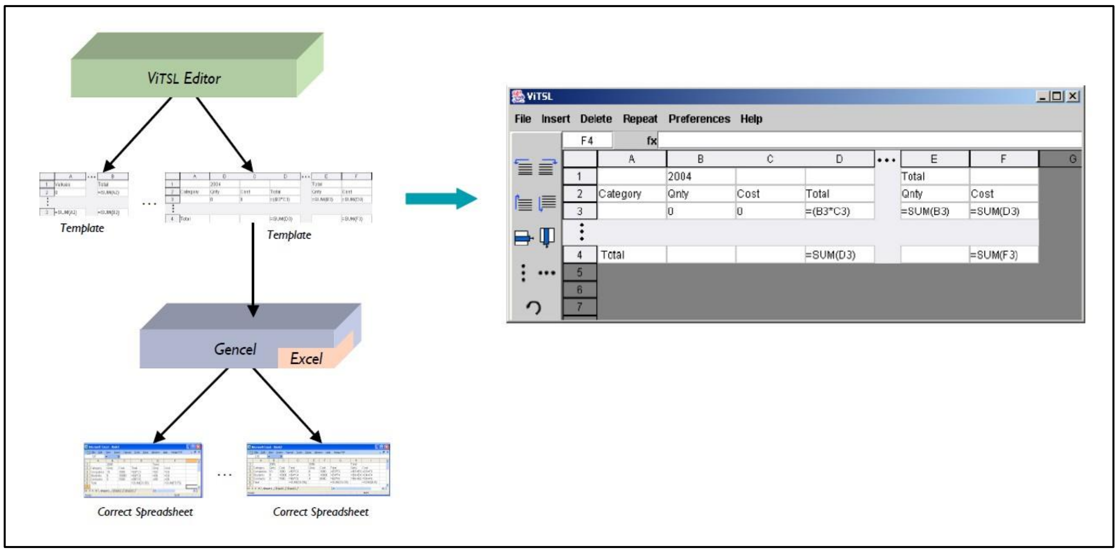
2.4 Model-Driven Spreadsheet Development

Model-Driven approaches to ensure error-free spreadsheets are an object of research, with several spreadsheet models integrating UML concepts already proposed.

Those models allow the developer to express explicitly business object structures and their respective logic within a spreadsheet. A stepwise automatic transformation process generates a spreadsheet application that is consistent with the defined model.

2.4.1 ViTSL

A visual specification language for spreadsheets called ***ViTSL*** [25] (an acronym for Visual Template Specification Language) was introduced to generate MS Excel spreadsheets.



	A	B	C	D	⋯	E	F	G
1		2004				Total		
2	Category	Qnty	Cost	Total		Qnty	Cost	
3		0	0	=(B3*C3)		=SUM(B3)	=SUM(D3)	
⋮								
4	Total			=SUM(D3)			=SUM(F3)	
5								
6								
7								

Figure 2.1: ViTSL/Gencel Architecture and editor screenshot [25]

ViTSL specifications – called templates – are constructed with an editor visually similar to spreadsheets and then loaded into an Excel extension named Gencel which provides an environment that manages the use of the spreadsheet that behaves dependent on the given specification (Figure 2.1), handling all formula generation and spreadsheet structure modification, ensuring that all spreadsheet formulas are correct and allowing the user to focus on data entry and analysis. Moreover, this language also has a formal textual representation.

While this language ensures the creation and the of spreadsheets free from a large class of accidental errors – such as type or pointing errors –, and has a familiar spreadsheet-like specification editor, it does not cover other types of errors (e.g., domain errors), offers a poor understanding of the spreadsheet structure and logic, and after creating the spreadsheet it is not possible to modify the template.

2.4.2 ClassSheets

The last one lead to the introduction of ***ClassSheets*** [26], a higher-level object-oriented spreadsheet model based on ViTSL from which inherits all the safety/correctness features and extends the textual representation. Instead of the ViTSL editor, there is a ClassSheet editor where it is possible to define a worksheet through a ClassSheet.

A ClassSheet represents two aspects of a spreadsheet application: the structure and the relationships of the involved classes and objects, along with the details of how attributes are related and derived from one another – the definition of these aspects are embed into a grid- based layout similar to the ViSTL templates.

Moreover, a UML-like class diagram can be derived from a ClassSheet [26], although forgetting all layout information. This diagram serves as an additional documentation of the class structure in the ClassSheet.

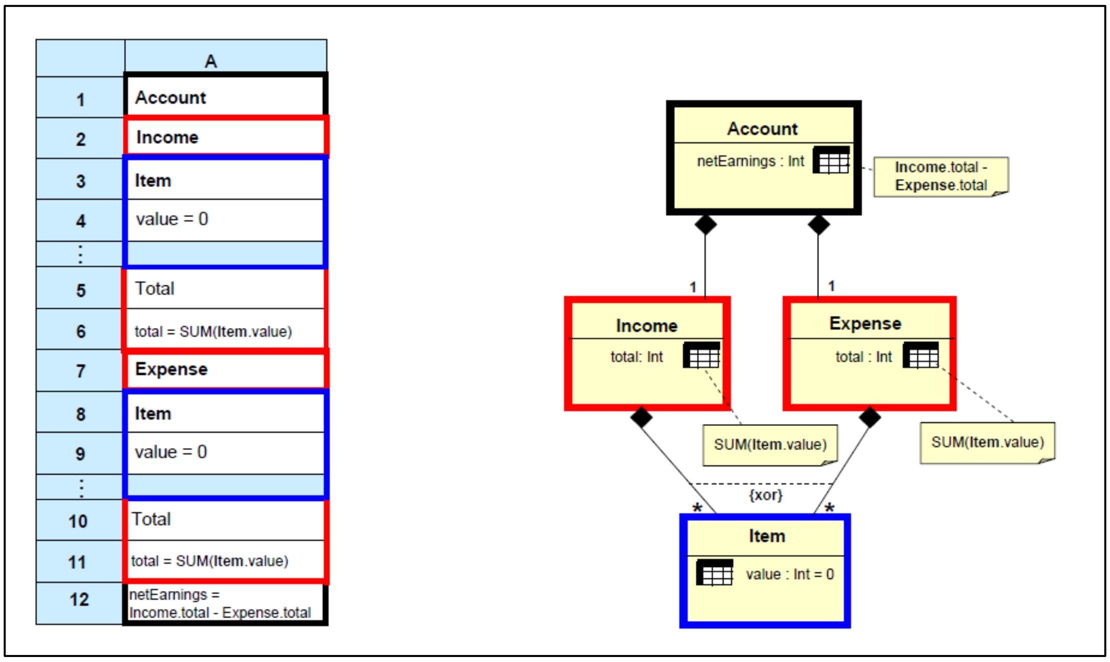


Figure 2.2: A ClassSheet with the corresponding UML class diagram [26]

In Figure 2.2 we can see an illustrative example of a ClassSheet that models an account system. There are three classes: *Account* (represents the account sheet), *Income*, *Expense* and *Item*. *Account* contains *Income* and *Expense*; this last two classes both contain *Item*. *Item* have the attribute value and can be expanded vertically, so *Income* or *Expense* can include several items.

Both Income and Expense have the attribute total which is calculated by the sum of their items value. Finally, Account has also an attribute of its own named netEarnings that is calculated by the subtraction of the attribute total of Income by the attribute total of Expense.

The structures and their respective relationships described above are directly reflected in the UML diagram representation (Figure 2.2), where, from an object-oriented viewpoint, there is an enclosing aggregating class *Account* with two enclosed aggregated classes *Income* and *Expens*e, with both of these two having by their turn the class *Item* as an aggregate from which they can have multiple instances. Also, there is a constraint *xor* which tells that the instances of the *Item* class aggregated to *Income* are disjoint from the ones aggregated to *Expense*.

The annotation of cells with classes has to result in a regularly nested class structure. This condition can be formalized through a type system that infers a spatial structure called *tiling* for ClassSheets. ClassSheets are considered to be well-formed if the type system is able to infer a proper tiling from the modeled structure.

$$
\begin{array}{lcll}
\tau & ::= & \blacksquare \mid \theta \mid \phi & (\mathit{tilings}) \\
\theta & ::= & [\tau] \mid \boxed{\tau} \mid \theta \,|\, \theta & (\mathit{horizontal\ tilings}) \\
\phi & ::= & \langle\tau\rangle \mid \boxed{\tau} \mid \phi\,\hat{}\,\phi & (\mathit{vertical\ tilings})
\end{array}
$$

Figure 2.3: Tiling Syntax [26]

In Figure 2.3 are presented the tillings construction rules. Tillings are principally composed by nested aggregations, which can grow either horizontally or vertically. Tillings can also feature two-dimensional aggregations, which basically are relations between two classes who share attributes.

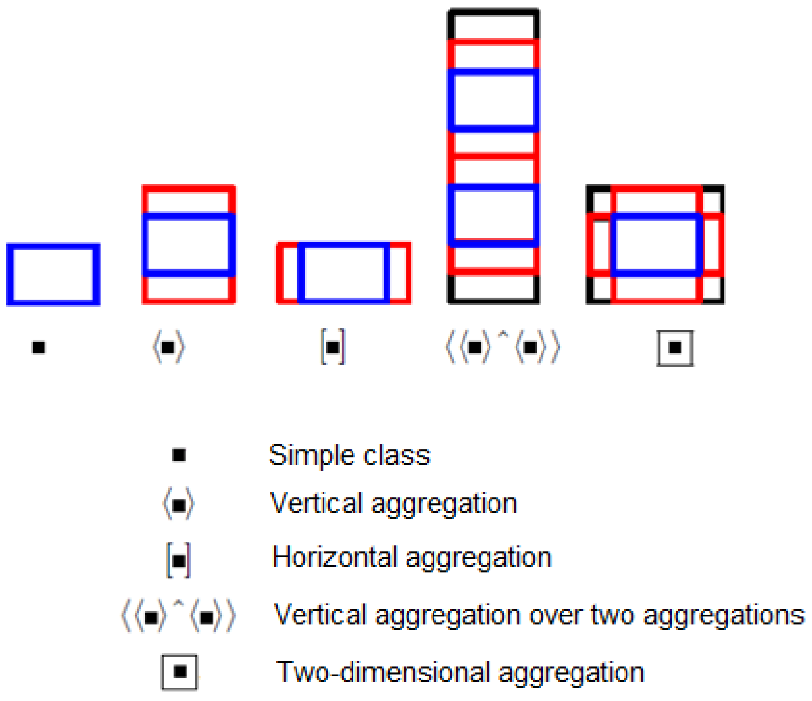

Simple class
Vertical aggregation
Horizontal aggregation
Vertical aggregation over two aggregations
Two-dimensional aggregation

Figure 2.4: Visualization of some tiling structures examples [26]

In Figure 2.4 we can see the visualization of some tilling constructions; the second from the right represents the already mentioned Account Sheet (Figure 2.2).

The first from the right represents the mentioned two-dimension aggregation that is illustrated by a concrete instance in Figure 2.5 – a budget sheet is composed by Budget class which contains *Category* and *Year* as aggregated classes. *Category* can be expanded vertically to set several items using the attribute name; *Year* can be expanded horizontally to include several years using the attribute year. *CpY (Category-per-Year*) relates a year with a category, since each instance is used to store information for all the categories of a year, and all the years of a category.

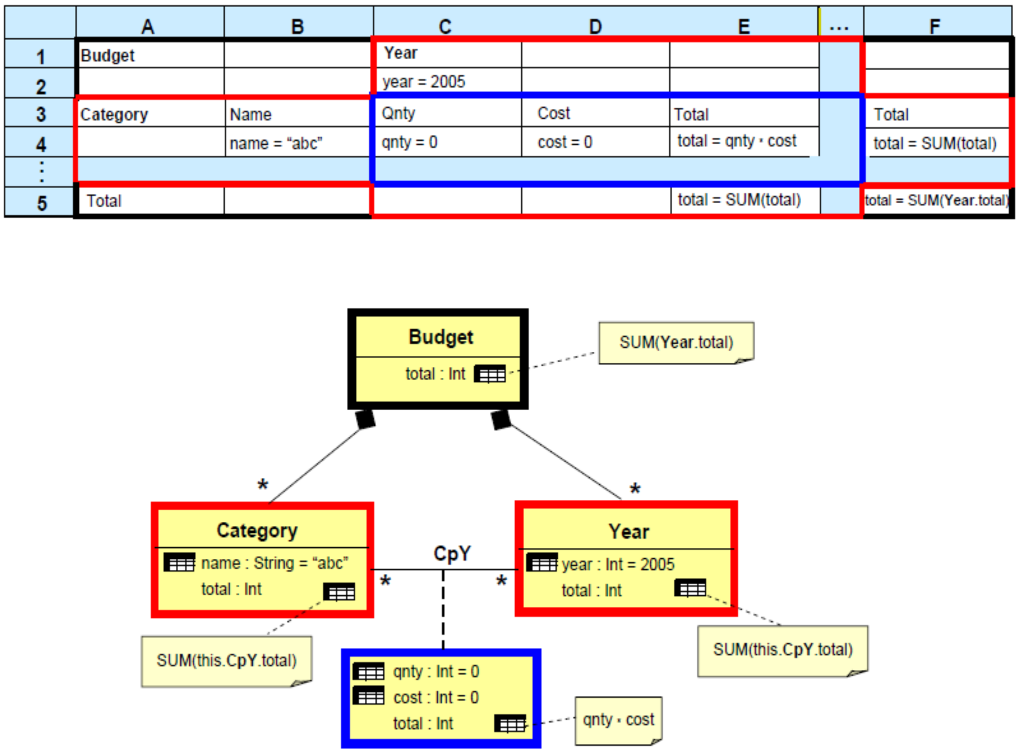


Figure 2.5: A two-dimensional ClassSheet with the corresponding UML class diagram [26]

Although ClassSheets offer an additional object-oriented modeling layer to further reduce the semantic gap between the application domains and the (automatically generated) applications, this model have several considerable drawbacks:

- ClassSheets do not cover some other common spreadsheet structures.
- Despite the higher-level modeling introduced, the understanding of the spreadsheet structure and logic is still quite poor for large-

spreadsheets; the UML-based documentation does not how the classes are physically structured (for instance, it is not expressed in which order the aggregated classes of a class appear in the grid).

- Domain errors are not addressed, that is, there is no way to express arbitrary constraints related to the underlying business logic.

- The model creation and the data insertion/edition occur in different environments.

- When a template is modified, a new spreadsheet needs to generated and the user has to migrate the data manually.

2.4.3 Embedded ClassSheets

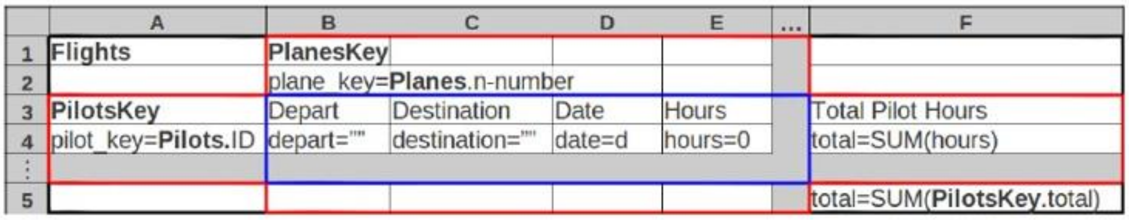
To address the last two drawbacks of ClassSheets, Cunha et. al. introduced ***Embedded ClassSheets*** [9], which consists of the native embedding of ClassSheet models in a spreadsheet host system. As a result, the model creation and data insertion/ modification occur in the same environment that end users are familiar with, and both model and instance are stored in the same file (see Figure 2.7 and 2.8).

Due to some limitations related to syntactic restrictions of the host system, the ClassSheet visual language suffered some modifications, namely:

- Classes are filled with a background color instead of using colored border lines. This change is not mandatory, but it easies the identification of classes' parts.

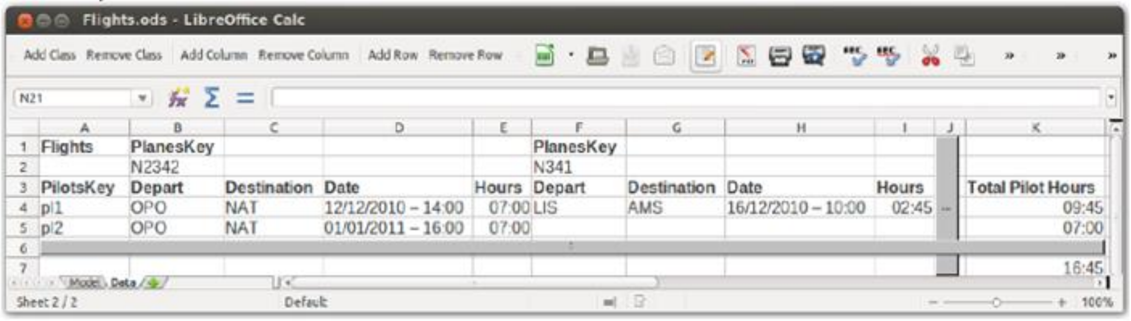
- Horizontal and vertical expansions are identified using cells instead of using columns/rows labels.

- Expansion limitation is defined with a black line in the spreadsheet instead of using column/row indices.



	A	B	C	D	E	...	F
1	Flights	PlanesKey					
2		plane_key=**Planes**.n-number					
3	PilotsKey	Depart	Destination	Date	Hours		Total Pilot Hours
4	pilot_key=**Pilots**.ID	depart=""	destination=""	date=d	hours=0		total=SUM(hours)
⋮							
5							total=SUM(**PilotsKey**.total)

Figure 2.6: Flights' visual ClassSheet model [9]

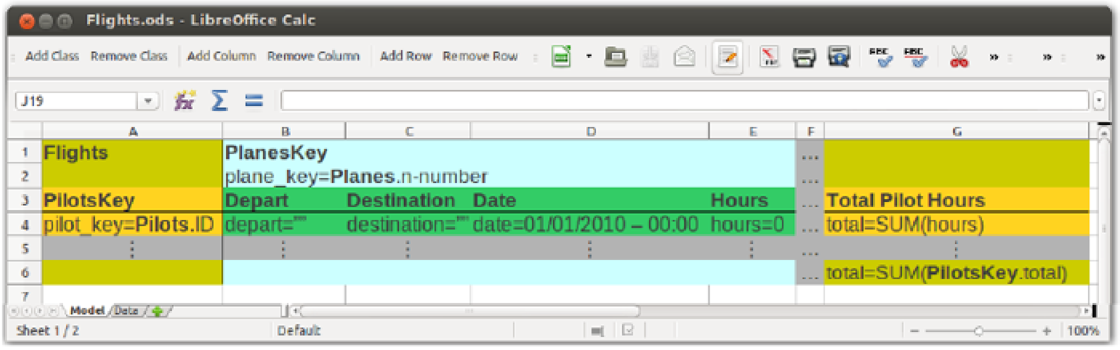


Flights.ods - LibreOffice Calc

	A	B	C	D	E	F	G	H	I	J	K
1	Flights	PlanesKey				PlanesKey					
2		N2342				N341					
3	PilotsKey	Depart	Destination	Date	Hours	Depart	Destination	Date	Hours		Total Pilot Hours
4	pl1	OPO	NAT	12/12/2010 – 14:00	07:00	LIS	AMS	16/12/2010 – 10:00	02:45	..	09:45
5	pl2	OPO	NAT	01/01/2011 – 16:00	07:00						07:00
6											
7											16:45

Sheet 2 / 2 Default 100%

Figure 2.7: Flights' visual embedded ClassSheet model [9]



Flights.ods - LibreOffice Calc

	A	B	C	D	E	F	G
1	Flights	PlanesKey				...	
2		plane_key=**Planes**.n-number				...	
3	PilotsKey	Depart	Destination	Date	Hours	...	Total Pilot Hours
4	pilot_key=**Pilots**.ID	depart=""	destination=""	date=01/01/2010 – 00:00	hours=0	...	total=SUM(hours)
5	⋮	⋮	⋮	⋮	⋮	...	⋮
6						...	total=SUM(**PilotsKey**.total)
7							

Sheet 1 / 2 Default 100%

Figure 2.8: Flights' visual embedded ClassSheet instance [9]

We can compare the mentioned differences between the original ClassSheet and the embedded one using a flights' spreadsheet example (Figures 2.6 and 2.7). We have a *Flight* class which aggregates classes *PlanesKey* and *PilotsKey*, with this two having a relationship. With that being said, the differences between the two ClassSheet models are:

I. In the original ClassSheet, the class Flights is identified by a black border line, the classes PlanesKey and PilotsKey are both identified by a red border line, and the relation class between these two are identified by a blue border line; in the embedded ClassSheet, Flights is identified by a green background, the class PlanesKey by a cyan background, the class PilotsKey by a yellow background, and the relation class between them by a green background.

II. Relatively to the expansion identification, in the original model we have two expansions: one is a horizontal expansion denoted by the column between columns E and F, the other one, a vertical expansion, is denoted by a row between rows 4 and 5; in the new model the two expansions are represented by the column F and the row 5, respectively.

III. To define the expansion limits in the original ClassSheet, there are no separation lines between the column headers of columns B, C, D and E which makes the horizontal expansion use three columns and the vertical expansion only use one row. In the embedded ClassSheet, there is instead a line between columns A and B and another line between rows 3 and 4.

Additionally, embedded ClassSheets permit the model evolution alongside their model instance, so spreadsheets evolve correctly regarding their business logic. The evolution is bidirectional, that is, the instance can evolve from the instance evolution and vice versa. To accomplish this, two sets of transformation rules were defined for the model and the instance, so that any transformation on either artifact implies a sequence of one or more transformations on the other, keeping the instance always conformed to the model.

Furthermore, an empirical study on a non-industrial environment was conducted to analyze the impact of this solution – although at that time the implementation only supported a model-to-instance evolution –, and results showed that embedded ClassSheets favored a less erroneous spreadsheets and also lowered the spent time on building spreadsheets.

However, some drawbacks earlier mentioned remain; the model does not cover all of the usual real-world spreadsheets structures, it is still changeling to reason about large spreadsheets and it is not possible to define arbitrary constraints with respect to the business logic to address domain errors.

2.4.4 Automatic Spreadsheet Generation from a UML class diagram extended with OCL

A previous work proposed a MDE approach to the generation of spreadsheets using UML class diagrams augmented with OCL expressions, using a model-to-model transformation in the process [10].

The generation consists of two steps. First, a UML class diagram – the conceptual model – is defined. Although there is no word relatively to how exactly restricted is that class model in terms of elements involved, it is safe to assume there are some constraints on the kind of relationships the diagram can have, since none of the transformation rules written-down includes other relationships between classes aside association, that is, the other common class relationships such as aggregation, composition and generalization are ignored; also, association relationships are restricted to many-to- one associations. In Figure 2.9 we can see a UML class diagram that respects those restrictions, and is used further in the explanation of this approach.

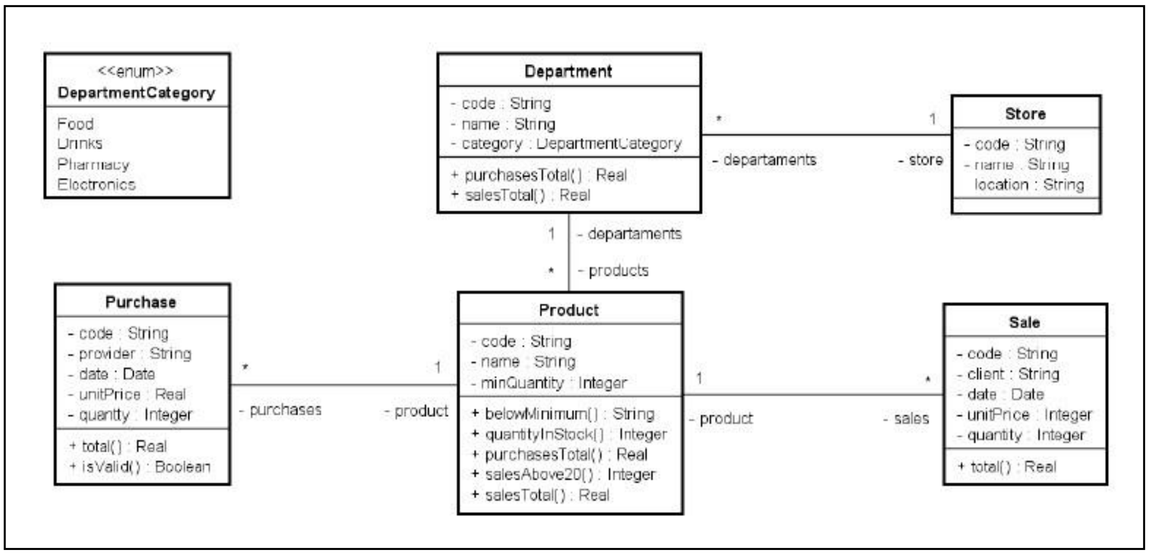


Figure 2.9: UML class diagram that supports the discussion on transformation rules [10]

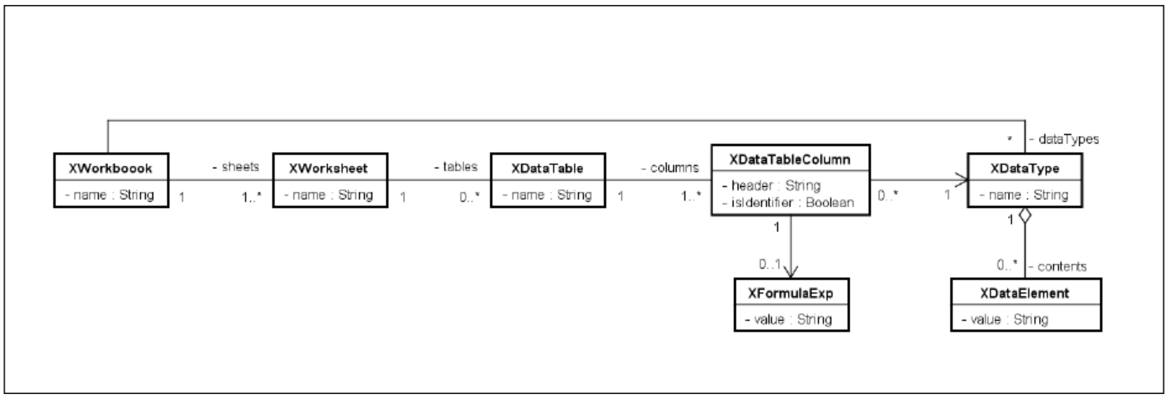


Figure 2.10: The spreadsheet metamodel [10]

The mentioned class diagram is used to define an another class diagram which basically represents the spreadsheet to generate – the spreadsheet model which conforms to a metamodel, the spreadsheet metamodel (Figure 2.10).

The proposed metamodel is very compact and consists of seven elements: the *XWorkboo*k – which represents the entire spreadsheet – may contain one or more *XWorksheets*, each corresponding to a spreahsheet tab. A *XWorksheet* can be connected to zero or more instances of *XDataTable*. An *XDataTable* is a data table represented in a worksheet, consisting of rows and columns. Each column is represented by an instance of *XDataTableColumn* which has an attribute named *header* that gives a name to the contents of its cells, and another attribute named *isIdentifier* that indicates whether the data in its cells can be used to uniquely identify a row in the table. A table can have one or more identifier columns. *XDataType* specify the type of data represented in the cells of a *XDataTableColumn* which can be either a standard type (number, string, date, among others) or an enumeration. Enumerations define a set of possible values (instances of *XDataElement*) which represent the domain of values that can be assumed by its associated cells. An *XDataTableColumn* may also contain a formula, and instance of *XFormulaExp* which has an attribute named *value* that describes the formula in his native textual format.

The transformation of the UML model to the spreadsheet model is specified by a set of rules, which can be summarized as follows:

I. A spreadsheet (an instance of the *XWorkbook*) element is created from a UML class diagram.

II. Each class from the UML class diagram is transformed into a worksheet (*XWorksheet*) with a single data table (*XDataTable*) whose name is given by the class name.

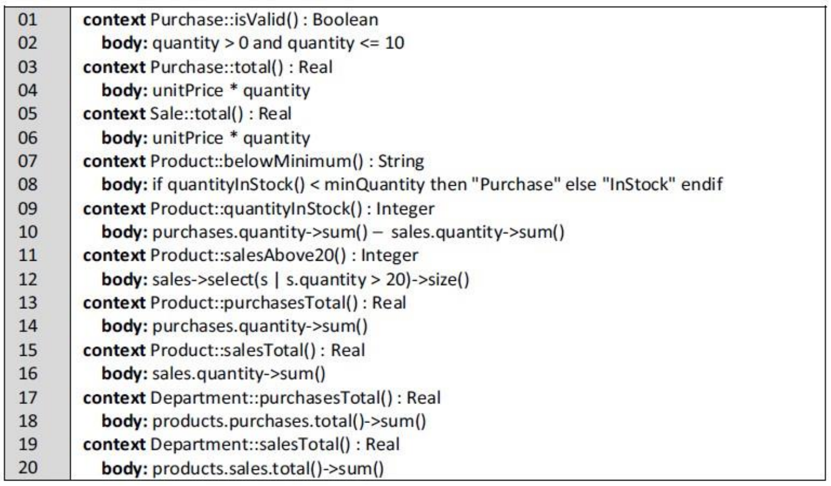
III. Each attribute of a class is transformed into a column (*XDataTableColumn*) in the data table corresponding to that class. An attribute associated to a stereotype named *Id* means that this attribute is either the identifier or part of the identifier of the data table.

IV. Each modifiable operation of a class (that is, the operation can change the state of the system) is transformed into a column in the data table corresponding to that class. The column type is defined according to the return type of the operation.

V. Each class from the UML model with the stereotype enumeration is transformed into an *XDataType* instance in the spreadsheet model linked to a set of *XDataElement*, one for each enumeration literal defined as an attribute of the enumeration.

VI. Each association from the UML model is transformed into a column. As previously stated, the model is restricted to many-to-one associations, so the transformation adds a new column to each table corresponding to the many side of each many-to-one binary association.

VII. The body of the class operations are expressed with OCL constrains, which are transformed into spreadsheet cell formulas (containing arithmetic, relational and conditional operators) and aggregation formulas using OCL collection operations, namely *sum()* and *size*(), which are transformed to spreadsheet common SUMIF and COUNTIF functions. Furthermore, the OCL association navigation expressions are transformed into INDEX and MATCH spreadsheet functions, used to return values form other data tables. In Figure 2.11 we can see some OCL expressions to specify formulas associated to the UML class diagram of Figure 2.9.



```
context Purchase::isValid() : Boolean
   body: quantity > 0 and quantity <= 10
context Purchase::total() : Real
   body: unitPrice * quantity
context Sale::total() : Real
   body: unitPrice * quantity
context Product::belowMinimum() : String
   body: if quantityInStock() < minQuantity then "Purchase" else "InStock" endif
context Product::quantityInStock() : Integer
   body: purchases.quantity->sum() – sales.quantity->sum()
context Product::salesAbove20() : Integer
   body: sales->select(s | s.quantity > 20)->size()
context Product::purchasesTotal() : Real
   body: purchases.quantity->sum()
context Product::salesTotal() : Real
   body: sales.quantity->sum()
context Department::purchasesTotal() : Real
   body: products.purchases.total()->sum()
context Department::salesTotal() : Real
   body: products.sales.total()->sum()
```

Figure 2.11: OCL expressions associated to the UML model [28]

The transformations mentioned were implemented in C#, where the input is a UML model annotated with OCL expressions defined in Visual Studio 2010, and the output is a MS Excel spreadsheet. Moreover, an experimental study was conducted to analyze this approach within IT professionals. The study showed that the solution provided benefits the creation of less erroneous spreadsheets. However, the validity of this study is questionable, since the number participants in the study were too small (six to be exact), and the diversity in terms of spreadsheet and domain used in the evaluation were limited to the UML model showed in Figure 2.9).

Although this work takes an initial approach of its value on the conceptualization of spreadsheets within a UML model, the solution presented has some major drawbacks:

- The spreadsheet model obviously lacks expressiveness, since it covers a very limited spreadsheet structures, namely, database-like tables.
- This approach only addresses development errors.

3

Towards Spreadsheet Patterns

Understanding the characteristics of real-world spreadsheets, namely industrial spreadsheets is essential to provide effective mechanisms to prevent the presence of errors in spreadsheets, especially when talking about Model-Driven Spreadsheet Engineering techniques. This Chapter presents previous studies concerning the characteristics of real-world spreadsheets and a catalogue of spreadsheet data arrangements, with the latter consisting in a contribution of the work in this dissertation.

The patterns described were obtain from two large real-word MS Excel spreadsheets data sets:

- **EUSES corpus** – published in 2005 and made available only to researchers, is a dataset of over 4,500 spreadsheets gathered from the public world-wide-web.

- **Enron corpus** – a large dataset containing around 15,000 industrial spreadsheets extracted from the Enron Corporation e-mail archive made public during the legal investigation concerning the company after it went bankrupt.

3.1 Characteristics of Industrial Spreadsheets

Previous works [11, 12] presented an analysis concerning the EUSES corpus and the Enron corpus spreadsheets, which focused mainly on the dimensions of size and coupling of spreadsheets, how functions are used in spreadsheets, and the presence of errors in spreadsheets. In Table 3.1 there is an analysis overview of both datasets.

Table 3.1: An overview of the spreadsheets in the Enron and EUSES set

	EUSES	Enron
Number of spreadsheets analyzed	4,447	15,770
Number of spreadsheets with formulas	1,961	9,120
Number of worksheets	16,853	70,983
Maximum number of worksheets	106	175
Number of non-empty cells	8,209,095	97,636,511
Average number of non-empty cells per spreadsheet	1,846	6,191
Number of formulas	730,186	20,277,835
Average of formulas per spreadsheet with formulas	372	2,223
Number of unique formulas	65,143	913,472
Number of unique formulas per spreadsheet with formulas	33	100

From those studies, it was possible to verify that there is a considerable lack of diversity of patterns regarding the analyzed characteristics among the different spreadsheets:

- The majority of the spreadsheets of both datasets are small, with a short number of worksheets (around 3 to 5, with some of them empty because, by default, a spreadsheet is created with 3 worksheets, which, in fact, contributes significantly to the mentioned average number).

- Concerning the degree of coupling, the majority of the spreadsheets (89%) is not linked to other spreadsheets and also within the spreadsheet only 20% of them have links between worksheets. On the cell level, the median path depth (calculation chain) is one and a cell with a formula has a median of three transitive precedent cells.

- Formulas are relatively simple; the diversity of built-in functions are very low (see Table 3.2 and 3.3), 76% of the Enron spreadsheets use only 5 distinct functions and the most used functions are the basic arithmetic ones, SUM, IF, NOW and AVERAGE, with the rest of the functions appearing in the spreadsheets within a percentage below 10%.

- User-defined functions are very uncommon (only 47 of the Enron spreadsheets have functions created by the user).
- Named ranges also have a poor utilization (only 5% of the Enron spreadsheets use them).
- The spreadsheets of the two datasets are quite similar with some considerable differences related to the average number of formulas and their calculation chains' length – the Enron spreadsheets have a higher average number of formulas and longer calculation chains.

Table 3.2: A comparison between Enron's and EUSES most used built-in functions

	Enron	EUSES
1	**SUM**	**SUM**
2	**IF**	**IF**
3	**AVERAGE**	**ROUND**
4	**VLOOKUP**	HYPERLINK
5	**ROUND**	**CONCATENATE**
6	SUBTOTAL	AND
7	OFFSET	COUNTIF
8	**CONCATENATE**	**AVERAGE**
9	NOW	OR
10	DAVERAGE	INDIRECT
11	SUMIF	MIN
12	INDEX	ISNUMBER
13	MATCH	MAX
14	LOOKUP	**VLOOKUP**
15	MONTH	ISBLANK

In what concerns to the presence of errors, it is impossible to determine what spreadsheet cells contain semantic errors, as we do not know the intention of a formula, nevertheless, syntactical errors occurrence in formulas are not possible, since the spreadsheet application (in this case Excel) does not allow an insertion of a syntactically incorrect formula, displaying named errors like #DIV/0 or #REF. Based on those errors, an automated analysis was made and, as we can see in Table 3.4 and 3.5, reference related errors were the most common ones, as more than 50% of the syntactically incorrect formulas consists of a reference that was not found, and over 40% of the spreadsheets with at least one syntax error contains an error type caused by an invalid reference.

Table 3.3: The most used functions and corresponding percentages in the Enron datset

Rank	Functions	# Spreadsheets	Percentage
1	SUM	6493	72.0%
2	+	5571	61.8%
3	-	4866	54.0%
4	/	3527	39.1%
5	*	3112	34.5%
6	IF	1827	20.3%
7	NOW	1501	16.7%
8	AVERAGE	879	9.8%
9	VLOOKUP	763	8.5%
10	ROUND	606	6.7%
11	TODAY	537	6.0%
12	SUBTOTAL	385	4.3%
13	MONTH	325	3.6%
14	CELL	321	3.6%
15	YEAR	287	3.2%
	Any above	8961	99.4%

Table 3.4: Spreadsheets containing Excel errors in the Enron dataset

Error Type	Spreadsheets	Formulas	Unique Formulas
#DIV/0!	580	76,656	4,779
#N/A!	635	948,194	6,842
#NAME?	297	33,9365	29,442
#NUM!	52	4,087	178
#REF!	931	18,3014	6824
#VALUE!	423	11,1024	1751
Total	2,205	1,662,340	49,796

Table 3.5: Spreadsheets Error Type Explanation

Error Type	Explanation
#DIV/0!	Trying to divide by 0
#N/A!	A formula or a function inside a formula cannot find the referenced data
#NAME?	Text in the formula is not recognized
#NULL!	A space was used in formulas that reference multiple ranges; a comma separates range references
#NUM!	A formula has invalid numeric data for the type of operation
#REF!	A reference is invalid
#VALUE!	The wrong type of operand or function argument is used

3.2 Common Data Arrangements of Spreadsheets

Knowing the typical data arrangement patterns of spreadsheets, in other words, what users usually want to model in a spreadsheet and what they usually expect to see in a spreadsheet, can be very useful insight in how to build mechanisms and strategies to specify less erroneous spreadsheets, namely addressing qualitative errors.

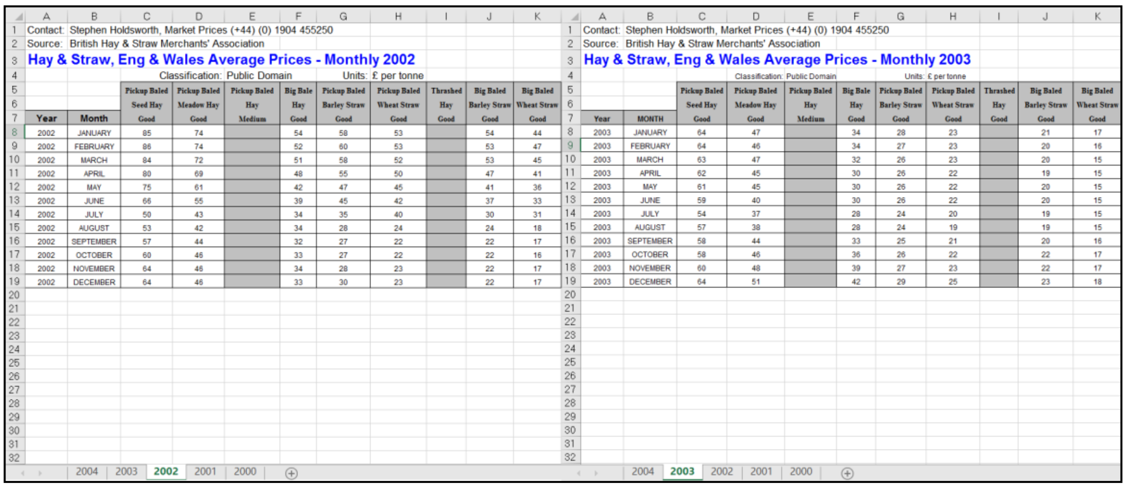
Works proposing spreadsheet models [9, 26, 27] already systematize common templates of table structures. Other works created a library containing common spreadsheet patterns [28] for later use of pattern matching algorithms in order to extract models from them. Other works implemented a header inference system for spreadsheets [29], describing the relation between the headers and their association with data. However, these patterns are quite far from covering all existing kinds of spreadsheet's data arrangements and do not take in consideration the domains where those patterns are generally applied.

In the work presented in this dissertation a step is taken on the extension of the current perception of the emerged spreadsheet patterns regarding the data arrangements. We studied EUSES corpus and Enron corpus (already described in Section 3.1) in terms of template structures, with the respective spreadsheets being manually observed and analyzed.

The analysis method consisted of manually selecting random spreadsheet samples from the two spreadsheet corpora, until the patterns observed were becoming redundant. Due to the low diversity verified, only 80 spreadsheets representative of all of the spreadsheets existing in the datasets were selected and. Next is presented the systemization of those data arrangement patterns.

3.2.1 Table Replication

In a spreadsheet, it is often observed the replication of table structures, only differing semantically in a certain aspect. In Fig. 3.1 we can see two structure replicas of a total of five replicas of a table, only differing in the year in which the table data concerns. In this case, the replicas are distributed by different worksheets, however, the replication can also occur on a single worksheet as shown in the example in Fig.3.2, where to calculate the "INCOME" and the "EXPENSES" the same table structure can be used.



Contact: Stephen Holdsworth, Market Prices (+44) (0) 1904 455250
Source: British Hay & Straw Merchants' Association
Hay & Straw, Eng & Wales Average Prices - Monthly 2002
Classification: Public Domain Units: £ per tonne

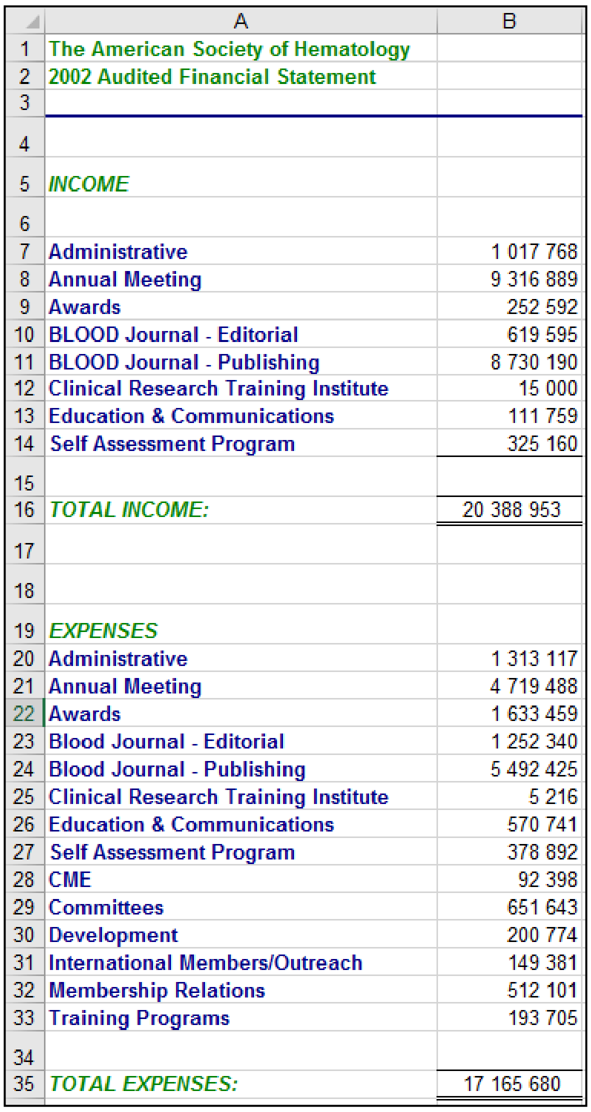
Year	Month	Pickup Baled Seed Hay Good	Pickup Baled Meadow Hay Good	Pickup Baled Hay Medium	Big Bale Hay Good	Pickup Baled Barley Straw Good	Pickup Baled Wheat Straw Good	Thrashed Hay Good	Big Baled Barley Straw Good	Big Baled Wheat Straw Good
2002	JANUARY	85	74		54	58	53		54	44
2002	FEBRUARY	86	74		52	60	53		53	47
2002	MARCH	84	72		51	58	52		53	45
2002	APRIL	80	69		48	55	50		47	41
2002	MAY	75	61		42	47	45		41	36
2002	JUNE	66	55		39	45	42		37	33
2002	JULY	50	43		34	35	40		30	31
2002	AUGUST	53	42		34	28	24		24	18
2002	SEPTEMBER	57	44		32	27	22		22	17
2002	OCTOBER	60	46		33	27	22		22	16
2002	NOVEMBER	64	46		34	28	23		22	17
2002	DECEMBER	64	46		33	30	23		22	17

Contact: Stephen Holdsworth, Market Prices (+44) (0) 1904 455250
Source: British Hay & Straw Merchants' Association
Hay & Straw, Eng & Wales Average Prices - Monthly 2003
Classification: Public Domain Units: £ per tonne

Year	MONTH	Pickup Baled Seed Hay Good	Pickup Baled Meadow Hay Good	Pickup Baled Hay Medium	Big Bale Hay Good	Pickup Baled Barley Straw Good	Pickup Baled Wheat Straw Good	Thrashed Hay Good	Big Baled Barley Straw Good	Big Baled Wheat Straw Good
2003	JANUARY	64	47		34	28	23		21	17
2003	FEBRUARY	64	46		34	27	23		20	16
2003	MARCH	63	47		32	26	23		20	15
2003	APRIL	62	45		30	26	22		19	15
2003	MAY	61	45		30	26	22		20	15
2003	JUNE	59	40		30	26	22		20	15
2003	JULY	54	37		28	24	20		19	15
2003	AUGUST	57	38		28	24	19		19	15
2003	SEPTEMBER	58	44		33	25	21		20	16
2003	OCTOBER	58	46		36	26	22		22	17
2003	NOVEMBER	60	48		39	27	23		22	17
2003	DECEMBER	64	51		42	29	25		23	18

Figure 3.1: Table replicated in different worksheets

The choice between the two replication options seem to depend on the table dimensions: larger table structures will naturally fit better in a spreadsheet on distinct worksheets (Fig. 3.1), while smaller ones can perfectly fit on the same worksheet (Fig. 3.2); and on the table purpose: if the spreadsheet analysis mainly relies on the comparison of the output data from the distinct replicas, it is convenient that the replicas stay physically close, which is the case of the example in Fig. 3.2 – besides the fact that the structures are quite small, the obvious object of analysis of the worksheet is the comparison between the "TOTAL INCOME" and the "TOTAL EXPENSES".



	A	B
1	The American Society of Hematology	
2	2002 Audited Financial Statement	
3		
4		
5	*INCOME*	
6		
7	Administrative	1 017 768
8	Annual Meeting	9 316 889
9	Awards	252 592
10	BLOOD Journal - Editorial	619 595
11	BLOOD Journal - Publishing	8 730 190
12	Clinical Research Training Institute	15 000
13	Education & Communications	111 759
14	Self Assessment Program	325 160
15		
16	*TOTAL INCOME:*	20 388 953
17		
18		
19	*EXPENSES*	
20	Administrative	1 313 117
21	Annual Meeting	4 719 488
22	Awards	1 633 459
23	Blood Journal - Editorial	1 252 340
24	Blood Journal - Publishing	5 492 425
25	Clinical Research Training Institute	5 216
26	Education & Communications	570 741
27	Self Assessment Program	378 892
28	CME	92 398
29	Committees	651 643
30	Development	200 774
31	International Members/Outreach	149 381
32	Membership Relations	512 101
33	Training Programs	193 705
34		
35	*TOTAL EXPENSES:*	17 165 680

Figure 3.2: Table replicated in the same worksheet

In Figure 3.3 we present the generic structures of the workbook and worksheet compositions. The tables displayed in a worksheet could be replicated or not, and different types of tables can compose a worksheet, namely *Vertical Table* (*Tv*), *Horizontal Single Entry Table* (*Th*) and *Relationship Table* (*Tr*). The three different types are talked in the next Section.

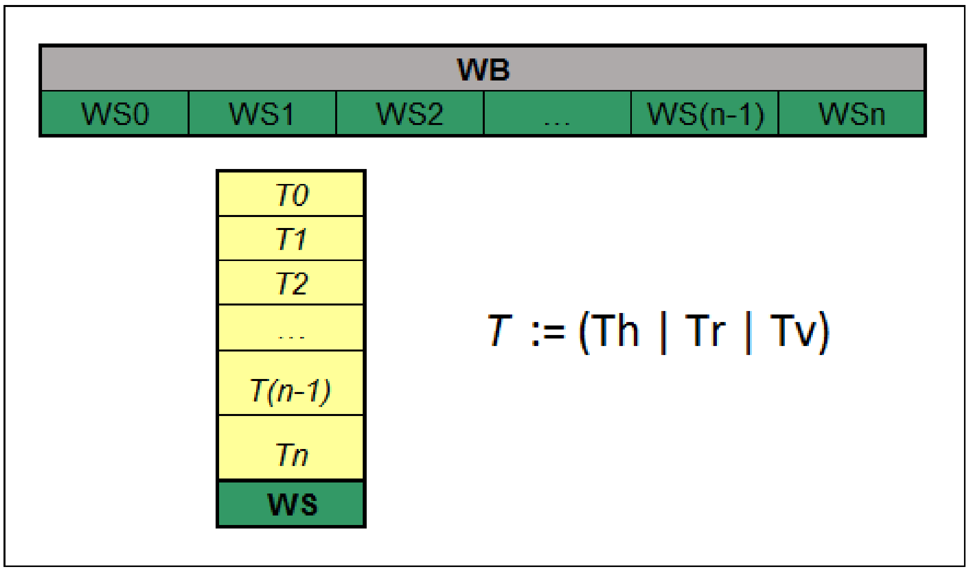


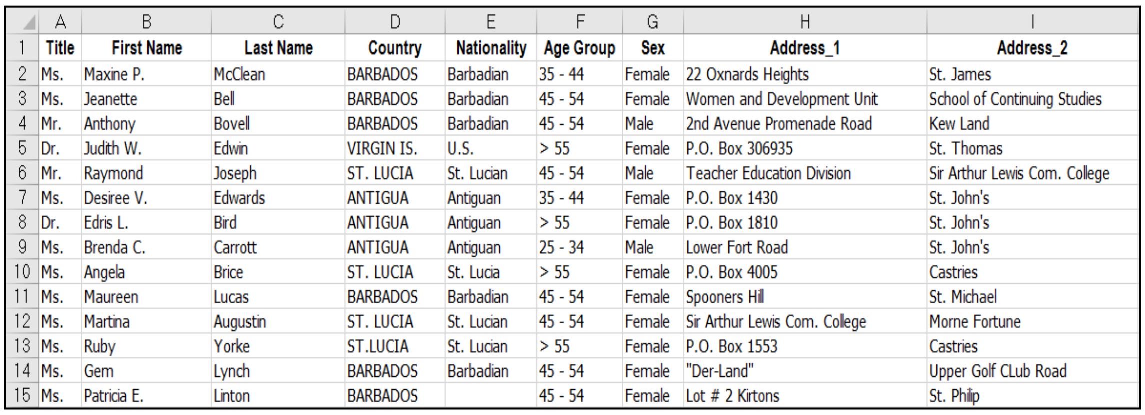
Figure 3.3: Generic structure of workbook and worksheet

3.2.2 Table Structures

When thinking about spreadsheets we immediately conceive tabular forms composed by a set of labels associated with a set of values. Based on the spreadsheets observed, it is possible to catalogue the common tables structures into three distinct groups which are defined by the table growth orientation and purpose.

Vertical Table

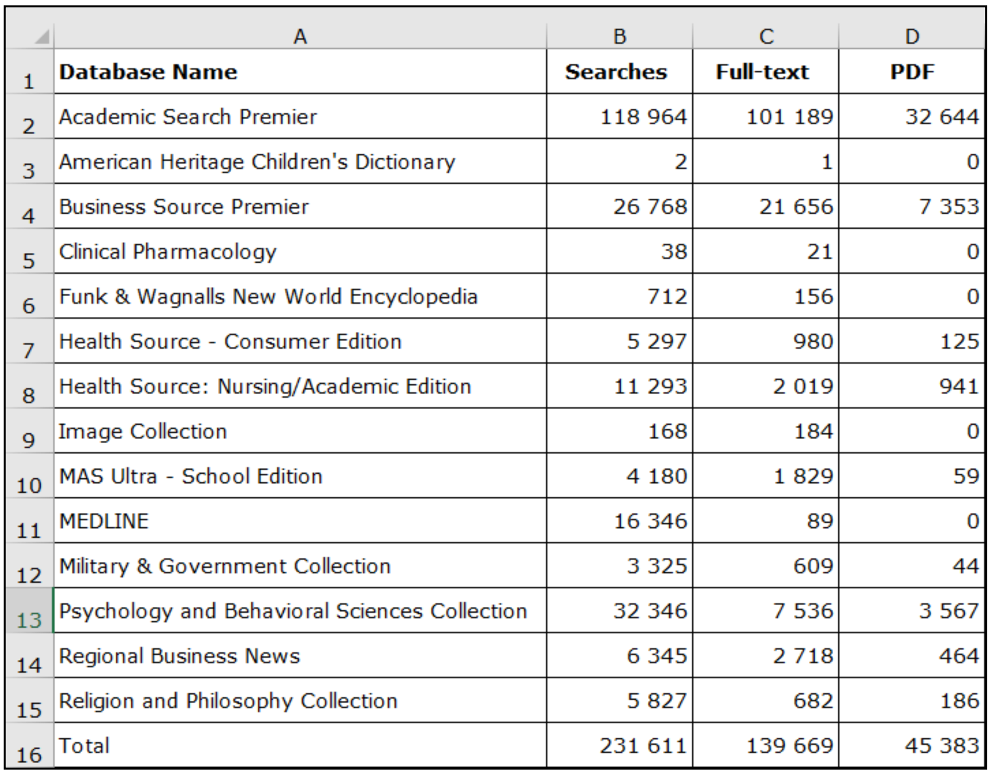
The most linear table structure consists of a simple grow-vertically table where there is a set of labels in the first row. Each label is associated with the set of values of its column. A label can either represent an entry value or a formula referring other row's entry values. This structure is commonly associated with inventory, database (Fig. 3.4) or statistical data (Fig. 3.5).



	A	B	C	D	E	F	G	H	I
1	Title	First Name	Last Name	Country	Nationality	Age Group	Sex	Address_1	Address_2
2	Ms.	Maxine P.	McClean	BARBADOS	Barbadian	35 - 44	Female	22 Oxnards Heights	St. James
3	Ms.	Jeanette	Bell	BARBADOS	Barbadian	45 - 54	Female	Women and Development Unit	School of Continuing Studies
4	Mr.	Anthony	Bovell	BARBADOS	Barbadian	45 - 54	Male	2nd Avenue Promenade Road	Kew Land
5	Dr.	Judith W.	Edwin	VIRGIN IS.	U.S.	> 55	Female	P.O. Box 306935	St. Thomas
6	Mr.	Raymond	Joseph	ST. LUCIA	St. Lucian	45 - 54	Male	Teacher Education Division	Sir Arthur Lewis Com. College
7	Ms.	Desiree V.	Edwards	ANTIGUA	Antiguan	35 - 44	Female	P.O. Box 1430	St. John's
8	Dr.	Edris L.	Bird	ANTIGUA	Antiguan	> 55	Female	P.O. Box 1810	St. John's
9	Ms.	Brenda C.	Carrott	ANTIGUA	Antiguan	25 - 34	Male	Lower Fort Road	St. John's
10	Ms.	Angela	Brice	ST. LUCIA	St. Lucia	> 55	Female	P.O. Box 4005	Castries
11	Ms.	Maureen	Lucas	BARBADOS	Barbadian	45 - 54	Female	Spooners Hill	St. Michael
12	Ms.	Martina	Augustin	ST. LUCIA	St. Lucian	45 - 54	Female	Sir Arthur Lewis Com. College	Morne Fortune
13	Ms.	Ruby	Yorke	ST.LUCIA	St. Lucian	> 55	Female	P.O. Box 1553	Castries
14	Ms.	Gem	Lynch	BARBADOS	Barbadian	45 - 54	Female	"Der-Land"	Upper Golf CLub Road
15	Ms.	Patricia E.	Linton	BARBADOS		45 - 54	Female	Lot # 2 Kirtons	St. Philip

Figure 3.4: Vertical Table used as a database

Also, sometimes there is an additional bottom row that applies an aggregation function to some specific labeled columns. In Figure 3.5 we can see a SUM function applied to columns B, C and D.



	A	B	C	D
1	**Database Name**	**Searches**	**Full-text**	**PDF**
2	Academic Search Premier	118 964	101 189	32 644
3	American Heritage Children's Dictionary	2	1	0
4	Business Source Premier	26 768	21 656	7 353
5	Clinical Pharmacology	38	21	0
6	Funk & Wagnalls New World Encyclopedia	712	156	0
7	Health Source - Consumer Edition	5 297	980	125
8	Health Source: Nursing/Academic Edition	11 293	2 019	941
9	Image Collection	168	184	0
10	MAS Ultra - School Edition	4 180	1 829	59
11	MEDLINE	16 346	89	0
12	Military & Government Collection	3 325	609	44
13	Psychology and Behavioral Sciences Collection	32 346	7 536	3 567
14	Regional Business News	6 345	2 718	464
15	Religion and Philosophy Collection	5 827	682	186
16	Total	231 611	139 669	45 383

Figure 3.5: Vertical Table used to display statistical data

Figure 3.6 presents a generic data arrangement structure of this type of table. Vertical table *Tv* is composed by several column labels (*Lh0…Lhn*) that are displayed next to each other horizontally, with the table records (*R0…R1*) displayed vertically. Those labels can be referring to an entry column *E*, a formula column *F*, or a horizontal header *Hh* (see further description in Section 3.2.2.). Additionally, aggregation function labels (*A0…An*) can optionally appear on the last table's rows associated to the respective functions applied vertically to specific columns.

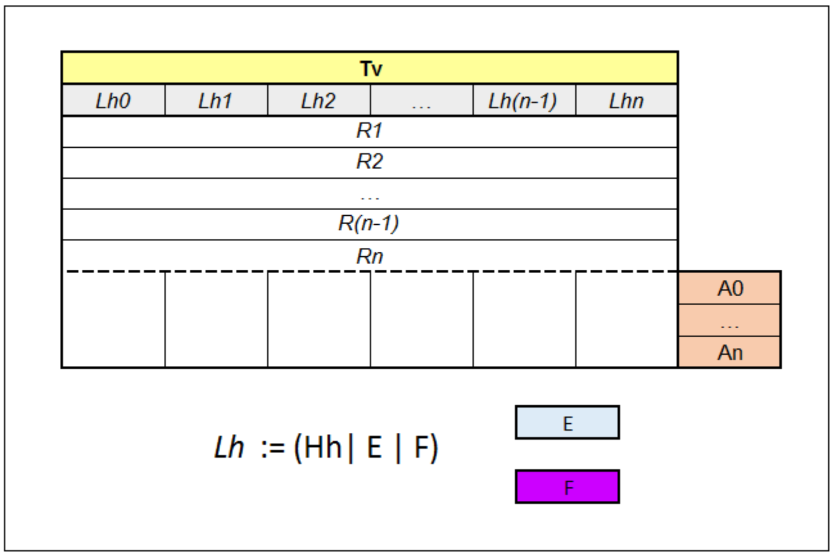
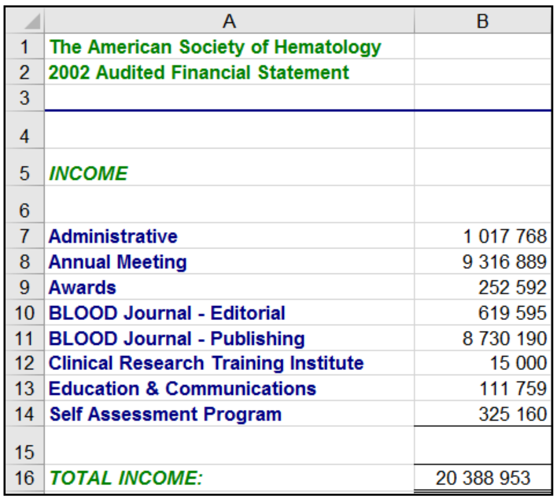


Figure 3.6: Generic structure of Vertical Table

Horizontal Single Entry Tables

A second table structure is a table whose labels are disposed vertically, and in which there is only one table record. Typically, the purpose of this kind of tables is to display summary data, and usually aggregation functions are applied on the solo record values.



	A	B
1	The American Society of Hematology	
2	2002 Audited Financial Statement	
3		
4		
5	*INCOME*	
6		
7	Administrative	1 017 768
8	Annual Meeting	9 316 889
9	Awards	252 592
10	BLOOD Journal - Editorial	619 595
11	BLOOD Journal - Publishing	8 730 190
12	Clinical Research Training Institute	15 000
13	Education & Communications	111 759
14	Self Assessment Program	325 160
15		
16	*TOTAL INCOME:*	20 388 953

Figure 3.7: Horizontal Single Entry Table example

In Figure 3.7, a SUM function is used to calculate the "TOTAL INCOME" from the above values.

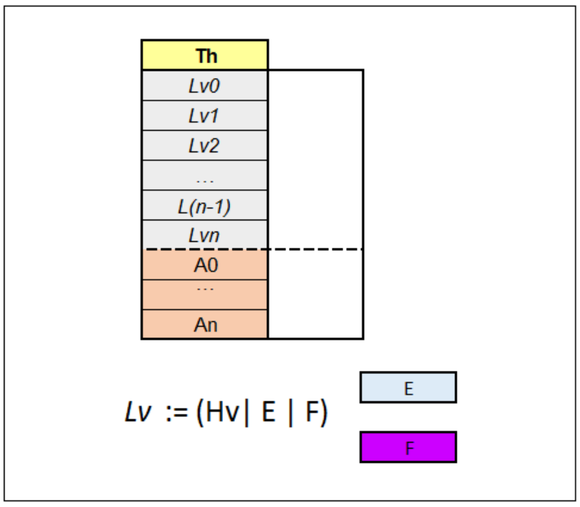
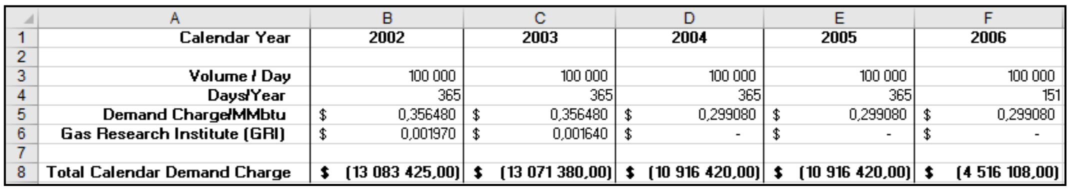


Figure 3.8: Generic structure of Horizontal Single Entry Table

Similar to Figure 3.6, Figure 3.8 shows a generic structure of this kind of table. In this case, the labels (including the aggregations) are displayed vertically and, instead of a horizontal header, a label could be referring to a vertical header (see further description in section 4.3), besides an entry or a formula.

Relationship Tables

A third group of table structures are the relationship tables, consisting of tables that grow horizontally, with a highlighted label – the top one. The top label values are themselves labels, that is, without those labels' entry value, the other label entry values are meaningless. Sometimes the top label is omitted, being only displayed its values. Aggregation functions are also commonly used on this tables, both vertically (see row "8" in Figure 3.9) and horizontally (see last column in Figure 3.11).



	A	B	C	D	E	F
1	Calendar Year	2002	2003	2004	2005	2006
2						
3	Volume / Day	100 000	100 000	100 000	100 000	100 000
4	Days/Year	365	365	365	365	151
5	Demand Charge/MMbtu	$ 0,356480	$ 0,356480	$ 0,299080	$ 0,299080	$ 0,299080
6	Gas Research Institute (GRI)	$ 0,001970	$ 0,001640	$ -	$ -	$ -
7						
8	Total Calendar Demand Charge	$ (13 083 425,00)	$ (13 071 380,00)	$ (10 916 420,00)	$ (10 916 420,00)	$ (4 516 108,00)

Figure 3.9: Relationship Table example

This table structure pattern dominates spreadsheets used for financial modeling and analysis, with the top header usually representing calendar years, year quarters, months, etc.

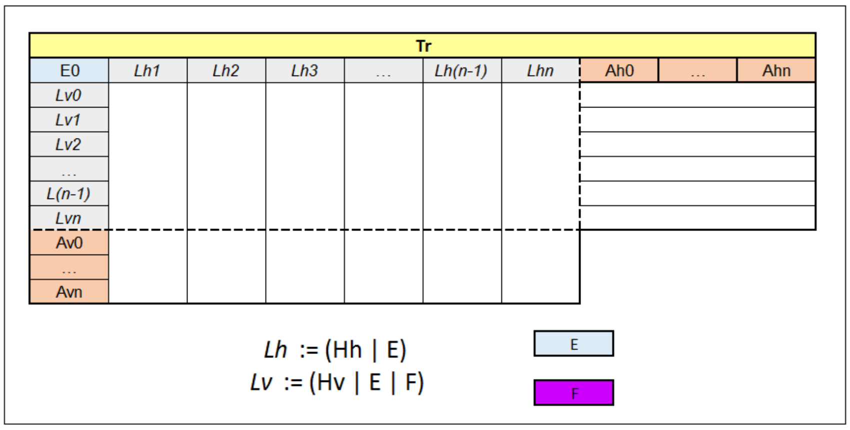


Figure 3.10: Generic structure of Relationship Table

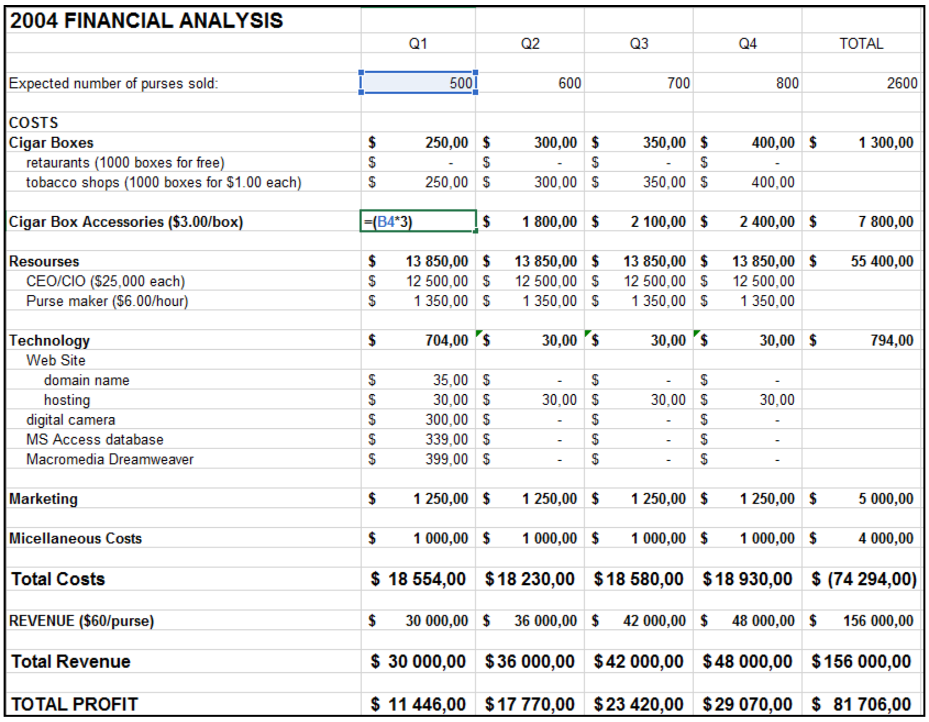
The generic structure of the Relationship Table is shown in Figure 3.10. In this case, both horizontal (*Lh1*...*Lhn*) and vertical (*Lv0*...*Ln*) labels exist, as well as horizontal (*Ah0...Ahn*) and vertical (*Av0...Avn*) aggregation functions. However, in this table, the horizontal headers cannot be formulas, since the formulas are applied vertically by the vertical labels.

3.2.2 Header Structures

Header Composition

In horizontal tables, it is usual to see labels composed by other labels. The main labels – the ones who are composed – typically represent categories, and the coupled ones are labels belonging to the category of the main label where they are attached. In this case, this main label constitutes a header.

Commonly, the header's associated value consists of an aggregation function – usually SUM – applied to the coupled labels' entry values.



2004 FINANCIAL ANALYSIS					
	Q1	Q2	Q3	Q4	TOTAL
Expected number of purses sold:	500	600	700	800	2600
COSTS					
Cigar Boxes	**$ 250,00**	**$ 300,00**	**$ 350,00**	**$ 400,00**	**$ 1 300,00**
retaurants (1000 boxes for free)	$ -	$ -	$ -	$ -	
tobacco shops (1000 boxes for $1.00 each)	$ 250,00	$ 300,00	$ 350,00	$ 400,00	
Cigar Box Accessories ($3.00/box)	=(B4*3)	**$ 1 800,00**	**$ 2 100,00**	**$ 2 400,00**	**$ 7 800,00**
Resourses	**$ 13 850,00**	**$ 13 850,00**	**$ 13 850,00**	**$ 13 850,00**	**$ 55 400,00**
CEO/CIO ($25,000 each)	$ 12 500,00	$ 12 500,00	$ 12 500,00	$ 12 500,00	
Purse maker ($6.00/hour)	$ 1 350,00	$ 1 350,00	$ 1 350,00	$ 1 350,00	
Technology	**$ 704,00**	**$ 30,00**	**$ 30,00**	**$ 30,00**	**$ 794,00**
Web Site					
domain name	$ 35,00	$ -	$ -	$ -	
hosting	$ 30,00	$ 30,00	$ 30,00	$ 30,00	
digital camera	$ 300,00	$ -	$ -	$ -	
MS Access database	$ 339,00	$ -	$ -	$ -	
Macromedia Dreamweaver	$ 399,00	$ -	$ -	$ -	
Marketing	**$ 1 250,00**	**$ 1 250,00**	**$ 1 250,00**	**$ 1 250,00**	**$ 5 000,00**
Micellaneous Costs	**$ 1 000,00**	**$ 1 000,00**	**$ 1 000,00**	**$ 1 000,00**	**$ 4 000,00**
Total Costs	**$ 18 554,00**	**$ 18 230,00**	**$ 18 580,00**	**$ 18 930,00**	**$ (74 294,00)**
REVENUE ($60/purse)	**$ 30 000,00**	**$ 36 000,00**	**$ 42 000,00**	**$ 48 000,00**	**$ 156 000,00**
Total Revenue	**$ 30 000,00**	**$ 36 000,00**	**$ 42 000,00**	**$ 48 000,00**	**$ 156 000,00**
TOTAL PROFIT	**$ 11 446,00**	**$ 17 770,00**	**$ 23 420,00**	**$ 29 070,00**	**$ 81 706,00**

Figure 3.11: Relationship Table with Header Composition

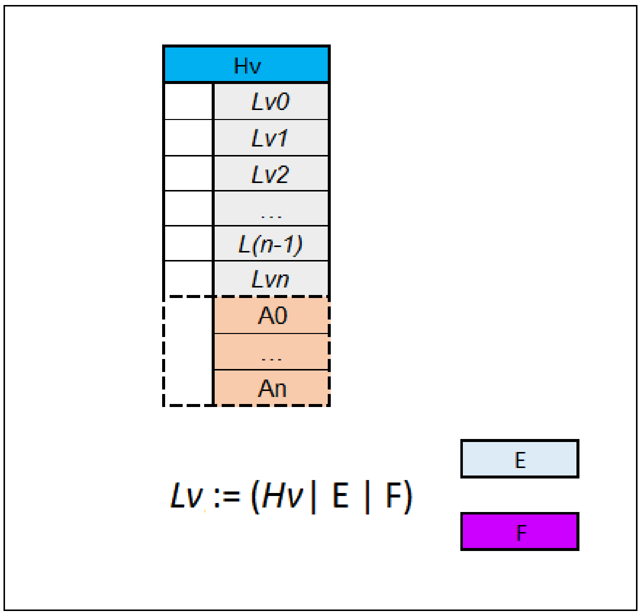


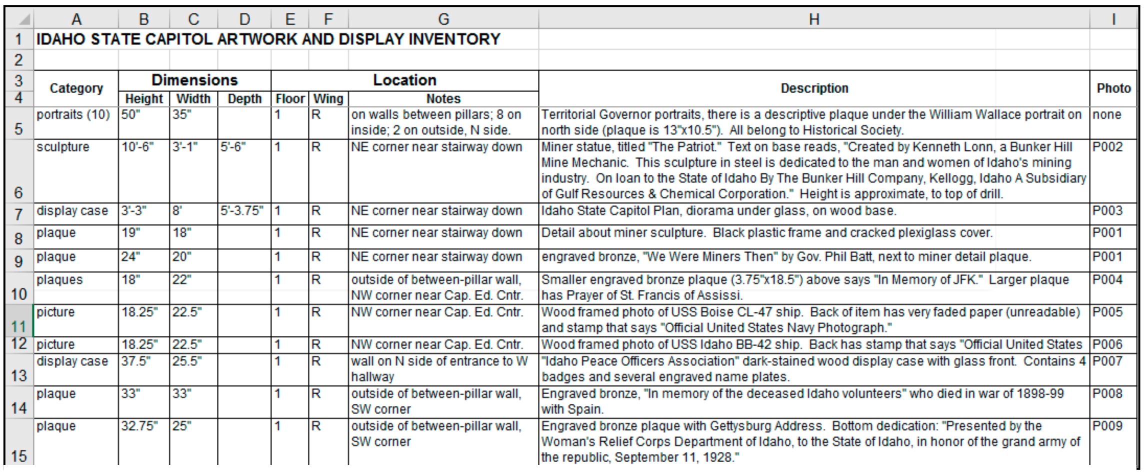
Figure 3.12: Generic structure of Header Composition

In Figure 3.11, we can see a relationship table composed by six headers: "Expected number of purses sold:", "COSTS", "Total Costs", "REVENUE ($60/purse)", "Total Revenue" and "TOTAL PROFIT", with the last four ones consisting of formulas. The header "COSTS" is composed by other six headers, with three of them – namely: "Cigar Boxes", "Recourses" and "Technology" – having attached headers of their own.

It is also possible to verify that "COST" has no table entry values associated, functioning as a pure categorization label, meanwhile the lower level headers, such as "Cigar Boxes", have entry values consisting of a SUM aggregation function applied to the labels' values they have attached. Figure 3.12 presents the generic structure of header composition.

Header Hierarchy

Similar to the composed labels, there are the hierarchically organized labels. Although in the header composition there is some sort of hierarchy, there are actually some major differences between the two label arrangements: in this type of label arrangement, the hierarchy is explicit, that is, the headers and their sub-labels are not physically on the same level.



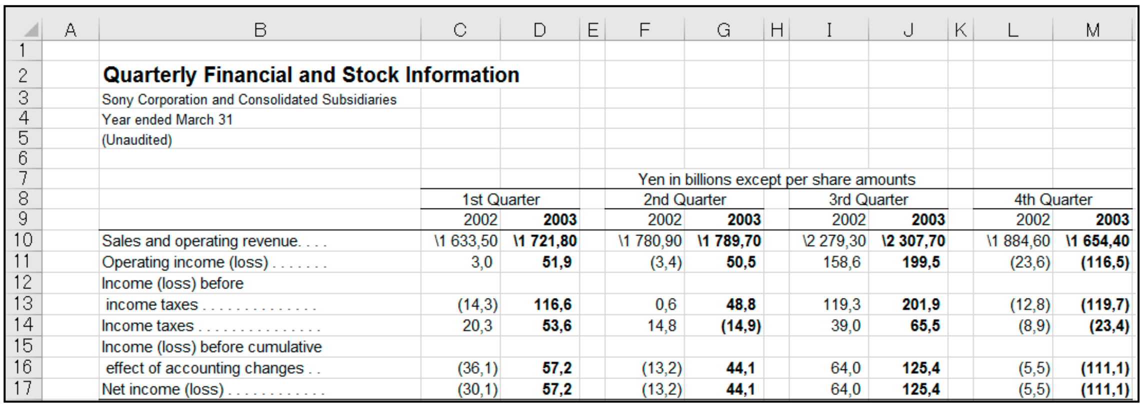
IDAHO STATE CAPITOL ARTWORK AND DISPLAY INVENTORY

Category	Dimensions			Location			Description	Photo
	Height	Width	Depth	Floor	Wing	Notes		
portraits (10)	50"	35"		1	R	on walls between pillars; 8 on inside; 2 on outside, N side.	Territorial Governor portraits, there is a descriptive plaque under the William Wallace portrait on north side (plaque is 13"x10.5"). All belong to Historical Society.	none
sculpture	10'-6"	3'-1"	5'-6"	1	R	NE corner near stairway down	Miner statue, titled "The Patriot." Text on base reads, "Created by Kenneth Lonn, a Bunker Hill Mine Mechanic. This sculpture in steel is dedicated to the man and women of Idaho's mining industry. On loan to the State of Idaho By The Bunker Hill Company, Kellogg, Idaho A Subsidiary of Gulf Resources & Chemical Corporation." Height is approximate, to top of drill.	P002
display case	3'-3"	8'	5'-3.75"	1	R	NE corner near stairway down	Idaho State Capitol Plan, diorama under glass, on wood base.	P003
plaque	19"	18"		1	R	NE corner near stairway down	Detail about miner sculpture. Black plastic frame and cracked plexiglass cover.	P001
plaque	24"	20"		1	R	NE corner near stairway down	engraved bronze, "We Were Miners Then" by Gov. Phil Batt, next to miner detail plaque.	P001
plaques	18"	22"		1	R	outside of between-pillar wall, NW corner near Cap. Ed. Cntr.	Smaller engraved bronze plaque (3.75"x18.5") above says "In Memory of JFK." Larger plaque has Prayer of St. Francis of Assissi.	P004
picture	18.25"	22.5"		1	R	NW corner near Cap. Ed. Cntr.	Wood framed photo of USS Boise CL-47 ship. Back of item has very faded paper (unreadable) and stamp that says "Official United States Navy Photograph."	P005
picture	18.25"	22.5"		1	R	NW corner near Cap. Ed. Cntr.	Wood framed photo of USS Idaho BB-42 ship. Back has stamp that says "Official United States	P006
display case	37.5"	25.5"		1	R	wall on N side of entrance to W hallway	"Idaho Peace Officers Association" dark-stained wood display case with glass front. Contains 4 badges and several engraved name plates.	P007
plaque	33"	33"		1	R	outside of between-pillar wall, SW corner	Engraved bronze, "In memory of the deceased Idaho volunteers" who died in war of 1898-99 with Spain.	P008
plaque	32.75"	25"		1	R	outside of between-pillar wall, SW corner	Engraved bronze plaque with Gettysburg Address. Bottom dedication: "Presented by the Woman's Relief Corps Department of Idaho, to the State of Idaho, in honor of the grand army of the republic, September 11, 1928."	P009

Figure 3.13: Vertical Table with a Header Hierarchy

In Fig. 3.13 it is possible to see a vertical table with two header hierarchies ("Dimensions" and "Location") which have a mere organizational purpose, with the intend of offering a clearer and focused table understating.

However, header hierarchies can be use with a comparison purpose in mind. As we can see in Fig. 3.14, there is a hierarchy for each header naming a year quarter ("1st Quarter", "2nd Quarter", "3rd Quarter" and "4th Quarter") with all of them sharing the same semantic yet physically different sub-labels.



Quarterly Financial and Stock Information

Sony Corporation and Consolidated Subsidiaries
Year ended March 31
(Unaudited)

	Yen in billions except per share amounts							
	1st Quarter		2nd Quarter		3rd Quarter		4th Quarter	
	2002	**2003**	2002	**2003**	2002	**2003**	2002	**2003**
Sales and operating revenue. . . .	\1 633,50	**\1 721,80**	\1 780,90	**\1 789,70**	\2 279,30	**\2 307,70**	\1 884,60	**\1 654,40**
Operating income (loss)	3,0	**51,9**	(3,4)	**50,5**	158,6	**199,5**	(23,6)	**(116,5)**
Income (loss) before								
income taxes	(14,3)	**116,6**	0,6	**48,8**	119,3	**201,9**	(12,8)	**(119,7)**
Income taxes	20,3	**53,6**	14,8	**(14,9)**	39,0	**65,5**	(8,9)	**(23,4)**
Income (loss) before cumulative								
effect of accounting changes . .	(36,1)	**57,2**	(13,2)	**44,1**	64,0	**125,4**	(5,5)	**(111,1)**
Net income (loss)	(30,1)	**57,2**	(13,2)	**44,1**	64,0	**125,4**	(5,5)	**(111,1)**

Figure 3.14: Relationship Table with a Header Hierarchy

Using this kind of arrangement obviates the need for multiple tables, whose physical separation makes it difficult to compare the analogous data from the distinct tables; or obviates the need for unique labels – for instance, using "1st Quarter 2002", "2nd Quarter 2002", etc., that also complicates the data analysis. Figure 3.15 presents the generic structure of header hierarchy.

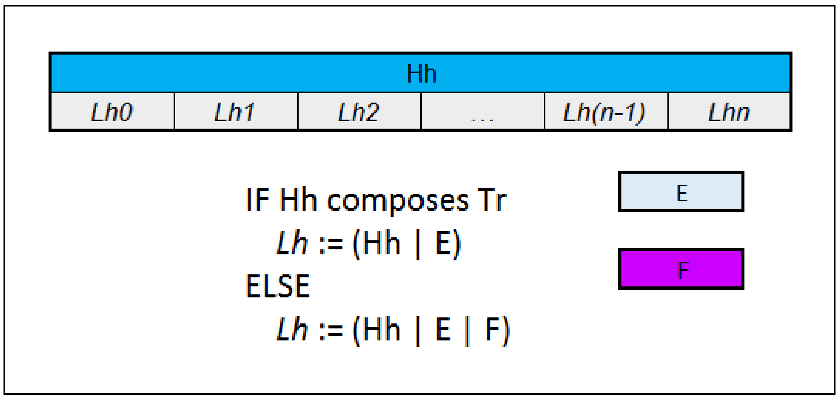


Figure 3.15: Generic Structure of Header Hierarchy

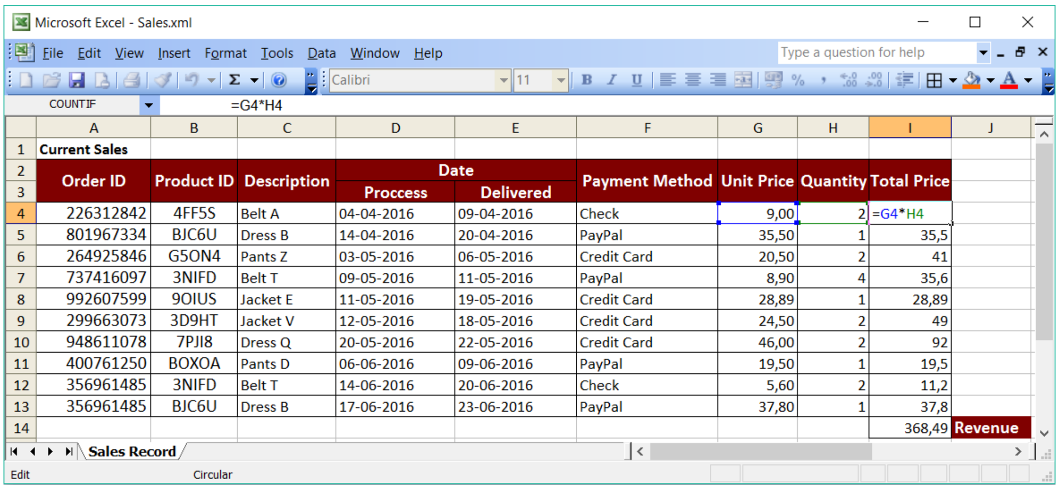
4 A UML-based DSL to Specify Spreadsheets

In this Chapter we describe our approach regarding the controlled specification of spreadsheets. First, we show an intuitive example of a custom spreadsheet that contains some of the patterns described in Chapter 3, and we present the corresponding specification using this work's DSL. Then, we describe the metamodel of our DSL and the mappings between the metamodel and the different spreadsheet patterns described in the previous Chapter. Moreover, we present a systemization of the errors addressed by this technique, favoring less erroneous spreadsheets.

4.1 Spreadsheet Specification

In Figure 4.1 we can see an illustrative example of a custom built spreadsheet named ***Sales.*** This spreadsheet has a solo worksheet named ***Sales Record*** which, in turn, has on it a grow-vertically table used to register sales orders of single clothing products. Each table record consists of nine fields: the ***Order ID*** –

uniquely identifies the order; the ***Product ID*** – uniquely identifies the product; the ***Description*** – textually describes the corresponding product; the ***Date*** – composed by two concrete dates: ***Process***, the date in which the order was processed and ***Delivered***, the date in which the product was actually delivered to buyer; the ***Payment Method*** – order's payment method which can be by Credit Card, Check or PayPal; the ***Unit Price*** – product price per unit; the ***Quantity*** – number of product replicas ordered; and ***Total Price*** – order's total amount to pay, which is the mathematical product of *Unit Price* by *Quantity*. Additionally, there is also an aggregation function – a SUM function – applied to the *Total Price* column values that gives the ***Revenue***.


Microsoft Excel - Sales.xml

COUNTIF =G4*H4

	A	B	C	D	E	F	G	H	I	J
1	Current Sales									
2	Order ID	Product ID	Description	Date		Payment Method	Unit Price	Quantity	Total Price	
3				Proccess	Delivered					
4	226312842	4FF5S	Belt A	04-04-2016	09-04-2016	Check	9,00	2	=G4*H4	
5	801967334	BJC6U	Dress B	14-04-2016	20-04-2016	PayPal	35,50	1	35,5	
6	264925846	G5ON4	Pants Z	03-05-2016	06-05-2016	Credit Card	20,50	2	41	
7	737416097	3NIFD	Belt T	09-05-2016	11-05-2016	PayPal	8,90	4	35,6	
8	992607599	9OIUS	Jacket E	11-05-2016	19-05-2016	Credit Card	28,89	1	28,89	
9	299663073	3D9HT	Jacket V	12-05-2016	18-05-2016	Credit Card	24,50	2	49	
10	948611078	7PJI8	Dress Q	20-05-2016	22-05-2016	Credit Card	46,00	2	92	
11	400761250	BOXOA	Pants D	06-06-2016	09-06-2016	PayPal	19,50	1	19,5	
12	356961485	3NIFD	Belt T	14-06-2016	20-06-2016	Check	5,60	2	11,2	
13	356961485	BJC6U	Dress B	17-06-2016	23-06-2016	PayPal	37,80	1	37,8	
14									368,49	Revenue

Sales Record

Edit Circular

Figure 4.1: Sales Spreadsheet

This spreadsheet can be mapped to the data arrangements structures systemized and formalized in Section 3.2 of Chapter 3, giving us the structure in Figure 4.2.

This structure consists of a Workbook ***WB*** with a solo Worksheet ***WS*** containing a Vertical Table ***Tv*** which, in turn, is composed by seven columns, six of them being an Entry (***E0***, ***E1***, ***E2***, ***E5***, ***E6*** and ***E7***), one being a Formula ***F*** that uses entries *E6* and *E7*, and a Horizontal Header ***Hh*** composed by two entries (***E3*** and ***E4***). Lastly, there is also an aggregation function ***A*** that uses the values of formula *F* column.

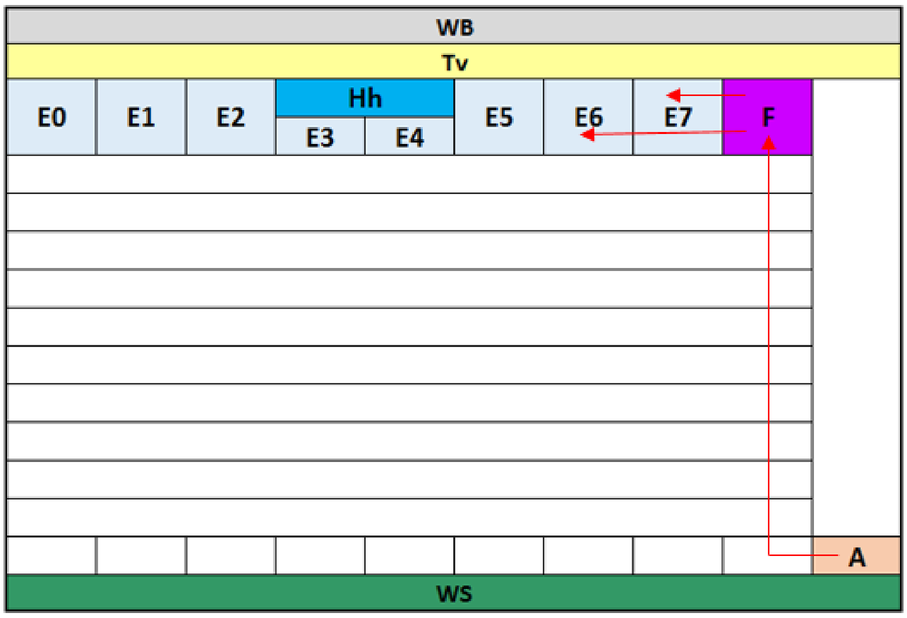


Figure 4.2: Sales spreadsheet's structure

Describing the previous structure, we used certain terms, namely "Composition", "Usage" and "Aggregation", which are very familiar to UML modeling concepts. This brings us to the DSL proposed in this work.

Figure 4.3 presents a UML-based specification of the *Sales* spreadsheet structure from our DSL. Each one of the object entities of the diagram match an element of the spreadsheet's structure (Figure 4.2): Sales – the Workbook WB; SRec – the Worksheet WS; CSales – the Vertical Table Tv; OrderID, ProductID, Dscr, PayM, UPrice, Qty – the entries E0, E1, E2, E5, E6, E7, respectively; TtPrice – the Formula F; Processed and Delivered – the entries E3, E4, respectively; and Rev – the Aggregation Function A. Finally, there are two object entities on the diagram that share the name "Date" and that match the Header Hh in the spreadsheet's structure: A Header instance and an Entry instance. This entry has no type (NONE) and serves only to define the order in which the header structure is physically arranged between the table entries.

Composition and *Usage* connection entities are used to specify the connections (and their type) among the object entities. For instance, there is a *Composi-*

tion connection between *SRec* and *CSales*. This connection, in terms of the spreadsheet's structure, means that *SRec* – the Worksheet *WS* – is physically composed by CSales – the Vertical Table *Tv*. Another example is the *Usage* connection from *TtPrice* to *Uprice* and *Qty*, meaning that the Formula *F* uses the values of Entry *E6* and Entry *E7*. An *Aggregation* connection between *Rev* and *CSales* is also used to specify the aggregation function applied by Aggregation Function *A* to the Vertical Table *Tv*, with the *Usage* connection additionally used to specify, in particular, which column cells are aggregated (*TtPrice* – Formula *F*).

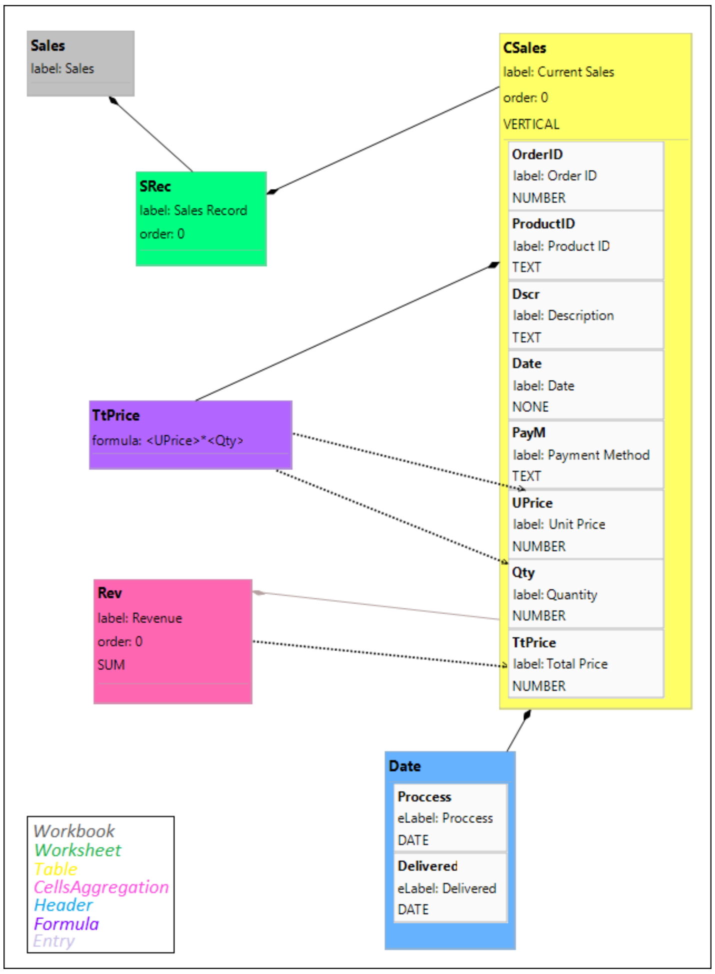


Figure 4.3: Sales spreadsheet's specification

With the use of this specification, we guarantee that a correct spreadsheet structure its created, and that the value types of its cells are not violated.

4.1.1 DSL Metamodel

The object entities described are extensions from the UML metamodel [30]. As it is possible to see in Figure 4.4, ***Workbook***, ***Worksheet***, ***Table***, ***CellsAggregation***, ***Header*** and ***Formula*** are all entities that extend the entity *Class*, inheriting the associations to *Composition*, *Aggregation* and *Usage* connections.

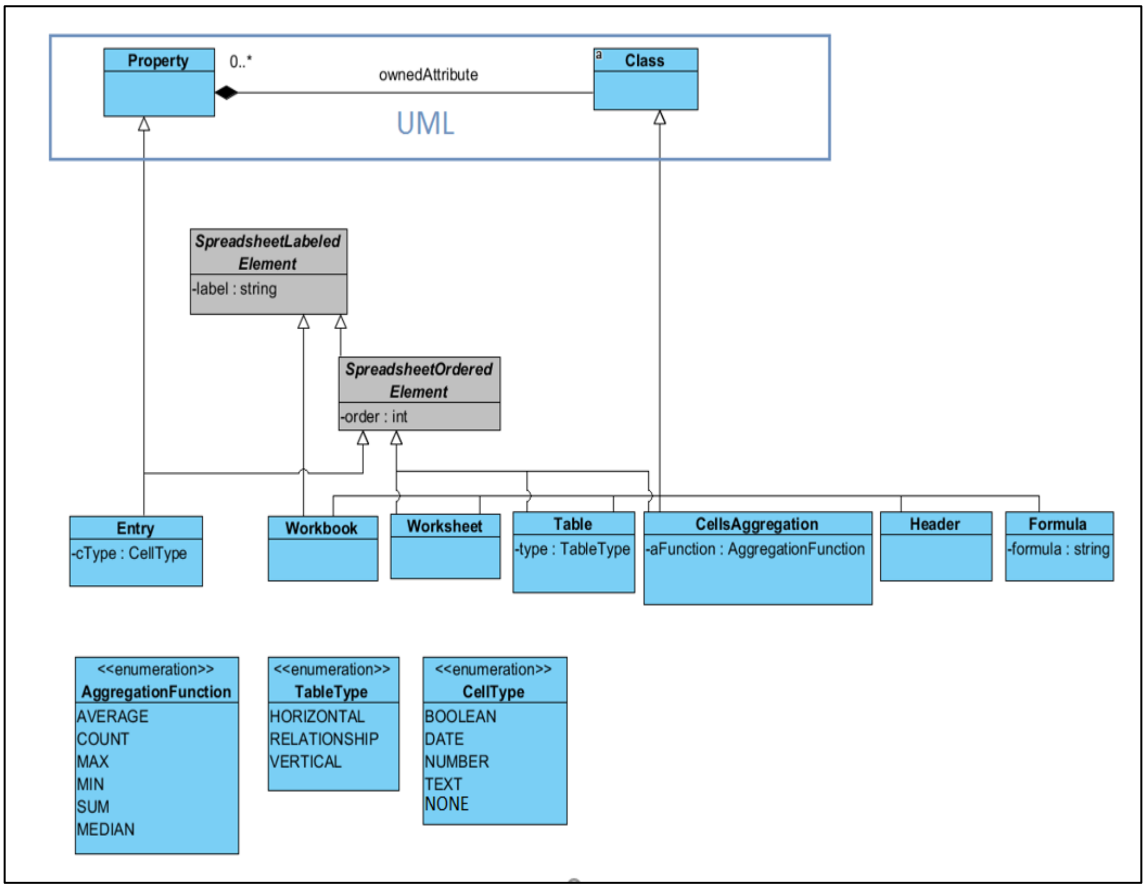


Figure 4.4: Part of the UML class diagram metamodel extended using our DSL metamodel

On the other hand, ***Entry*** extends the *Property* entity, inheriting only the association to *Usage* (which is common for both *Class* and *Property*), and the association composition to *Class*, being an *ownedAttribute* of this last entity.

Moreover, all those seven spreadsheet entities inherit the attribute name from *Class* and *Property* (which, in turn, inherit that same attribute from the *NamedElement* abstract class of UML).

There are also two abstract spreadsheet entities: ***SpreadsheetLabeledElement*** and ***SpreadsheetOrderedElement.*** The first represents a spreadsheet element that has a label; for example, the *Table* instance *CSales* in Figure 4.3 has a label with the value "Current Sales", which is the value that appears on the spreadsheet (cell A1) in Figure 4.1 identifying the table.

Entities such as *Formula* have no label attribute because it does not specify a physical structure element on the spreadsheet, but rather a dependency. The same goes for the *Header*, which specifies a structure organization. The second abstract spreadsheet entity represents all the concrete spreadsheet entities that have a sequence order; worksheets on a workbook appear on a sequence order, the same for the tables on a worksheet, and the entries and the aggregations function on a table, therefore, each one of the respective metamodel entities has an *order* attribute which specifies the sequence order of an instance in relation to the others.

Some concrete spreadsheet entities have their own attributes. *Entity* has an attribute named ***cType*** which specifies the type of the respective cell values. This attribute consists of an enumeration named ***CellType*** that has four literals: *BOOLEAN*, *NUMBER*, *TEXT* and *DATE*. The first three are the Excel's three data value types, and the latter is a very common formatting of a number value – in Tables 3.2 and 3.3 of Chapter 3 is possible to see that some of the most used bult-in function on the Enron dataset work with date formatted values, namely, YEAR, MONTH, NOW).

Table has an attribute named ***type*** which specifies the table's type: *HORIZONTAL*, *RELATIONSHIP* and *VERTICAL*, the three literals of the enumeration ***TableType***. This attribute specifies the data arrangement structure of table, being one of the three types described in Section 3.2.2 of Chapter 3. In Figure 4.3 this attributed is set *VERTICAL*, which makes the table grow vertically, with the entries, headers, and formula displayed horizontally (see Figure 4.2).

CellsAggregation entity has also his own attribute ***aFunction***, which specifies what aggregation function it is used, being one of the enumeration ***AggregationFunction*** literals: *AVERAGE*, *COUNT*, *MIN*, *MAX*, *SUM* and *MEDIAN* (in Table 3.2 of Chapter 3 it is possible to see that these functions are heavily used in spreadsheets).

Finally, ***Formula*** owns attribute ***formula*** which specifies the formula's expression to be applied to the respective cells. The expression's syntax is the same as the formula syntax of the spreadsheet system, except the cell references being replaced by entry names (see *TtPrice* in Figure 4.2).

4.1.2 Patterns 's Metamodel

The data arrangement patterns identified in Chapter 3 can be mapped to our DSL metamodel. In Figure 4.5 it is shown the part of metamodel concerning the workbook structure of a spreadsheet. A *Workbook* entity can have multiple *Worksheet* entities associated through a *Composition* connection entity, whereas a *Worksheet* entity is associated with only one *Workbook*.

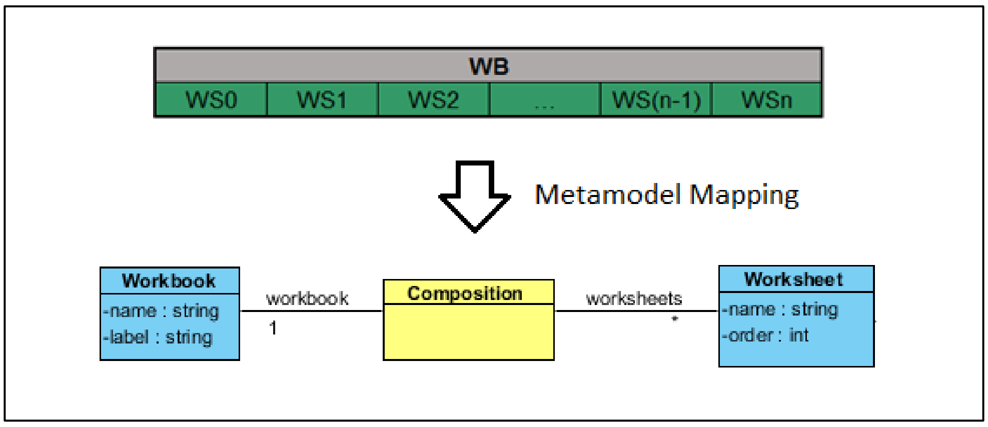


Figure 4.5: Workbook's Metamodel

In Figure 4.6 we can see the part of the metamodel regarding the structure of a spreadsheet. A *Worksheet* entity can have multiple Table entities - which in the spreadsheet are displayed next to each other vertically - through an association with a *Composition* connection entity. On the reverse side, a *Table* entity can have only one *Worksheet*, and the attribute *type* is used to define the type of the table (see *TableType* enumeration in Figure 4.4), since a spreadsheet can have distinct types of tables.

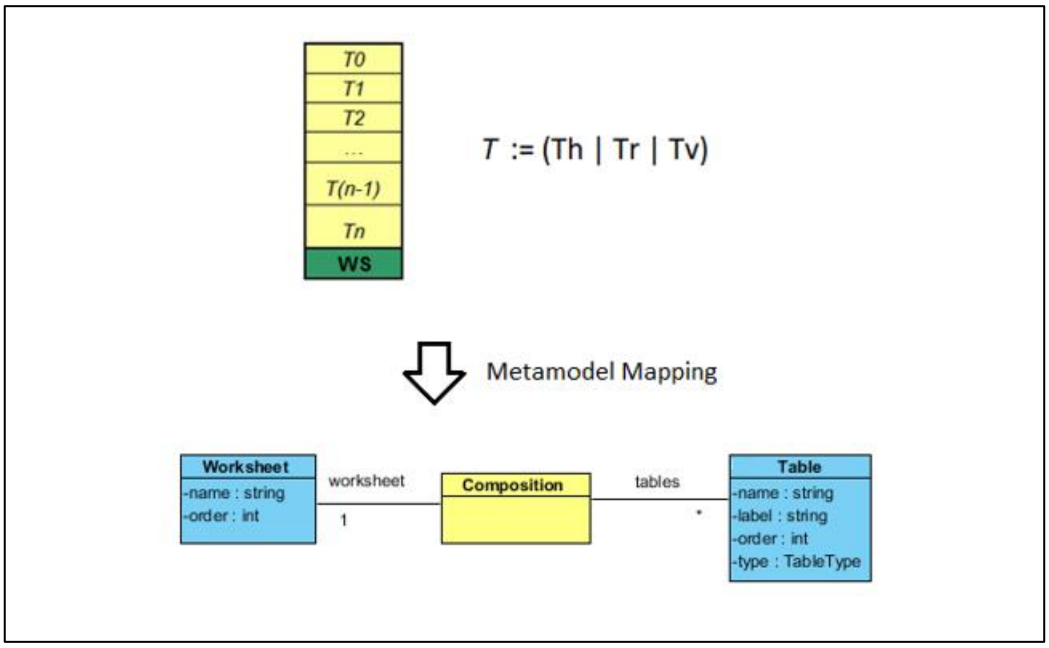


Figure 4.6: Worksheet's Metamodel

In respect to the three distinct table structures mentioned and described in Section 3.2.2 of Chapter 3, we can see in Figure 4.7 the part of the metamodel that concerns the data arrangements of these table structures. The *Table* entity is composed by multiple *Entry* entities, and has two associations through a *Composition* connection entity with *Header* and *Formula* entities. Also, the *Table* entity can have multiple *CellsAggregation* entities through an *Aggregation* connection. Moreover, a *Formula* entity can have multiple *Entry* entities through a *Usage* connection.

The value of the attribute *type* of the *Table* entity define how the mentioned entities' instances are physical arranged in the worksheet. If the *type* has the value *VERTICAL*, the table labels referring to entries, headers and formulas are placed horizontally next to each other, and the aggregation labels are displayed vertically at the bottom of the table, as we can see in the table structure in Figure 3.6; if the *type* has the value *HORIZONTAL*, all labels are displayed vertically as shows Figure 3.7; lastly, if the value is *RELATIONSHIP*, the *Header* instance that is associated with the *Entry* instance (has the same name) with the lesser *order* value attribute will be treated has a *Table* with the *type HORIZONTAL*, meanwhile the other entries are displayed horizontally, as we can see the table structure presented in Figure 3.7.

The Header's metamodel is presented in Figure 4.8. A *Header* entity has the same associations the *Table* entity has. The type of header structure is determined by the *type* of the *Table* instance associated with the root *Header* instance. If the

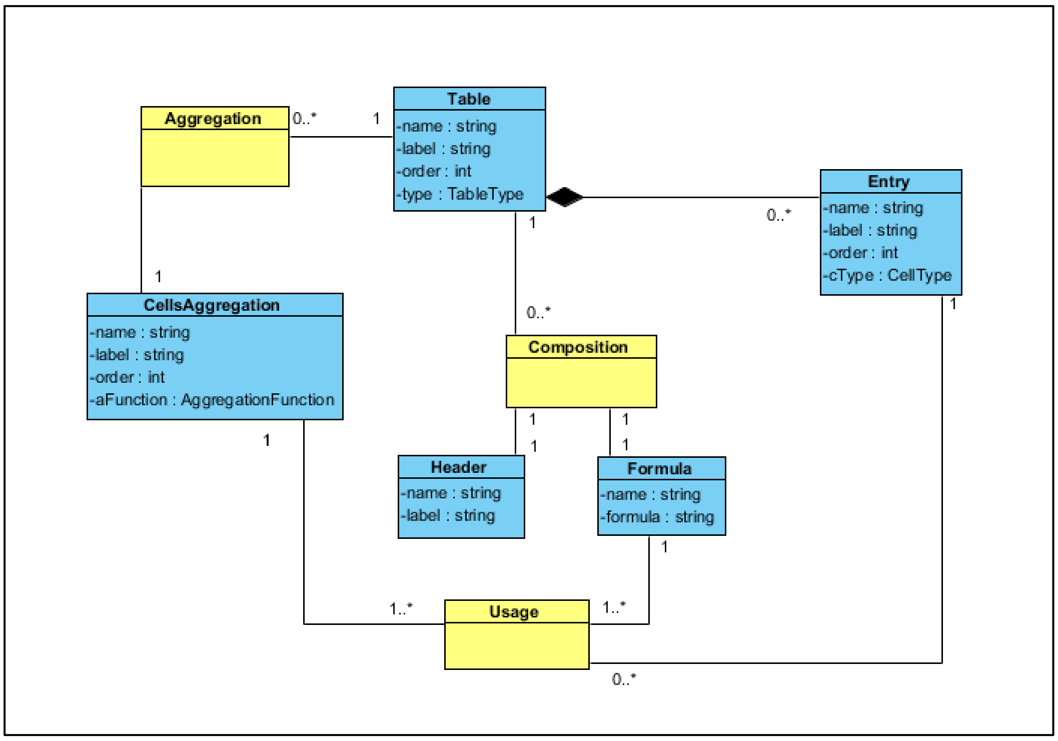


Figure 4.7: Table's Metamodel

root *Table* instance's type attribute has the value *VERTICAL*, then the header structure will be a *Header Composition* (Figure 3.12); if the value is *HORIZONTAL*, then a *Header Hierarchy* will be used (Figure 3.13); if the value is *RELATIONSHIP*, the *Header* instance related to the first *Entry* (the one with the lesser *order* value attribute) that composes the *Table* entity, and all the *Header* entity's "child headers" will be structured as a *Header Composition*, and the rest of the *Header* instances related to the rest of the *Entry* entities that compose the *Table* entity will be structured as a *Header Hierarchy*.

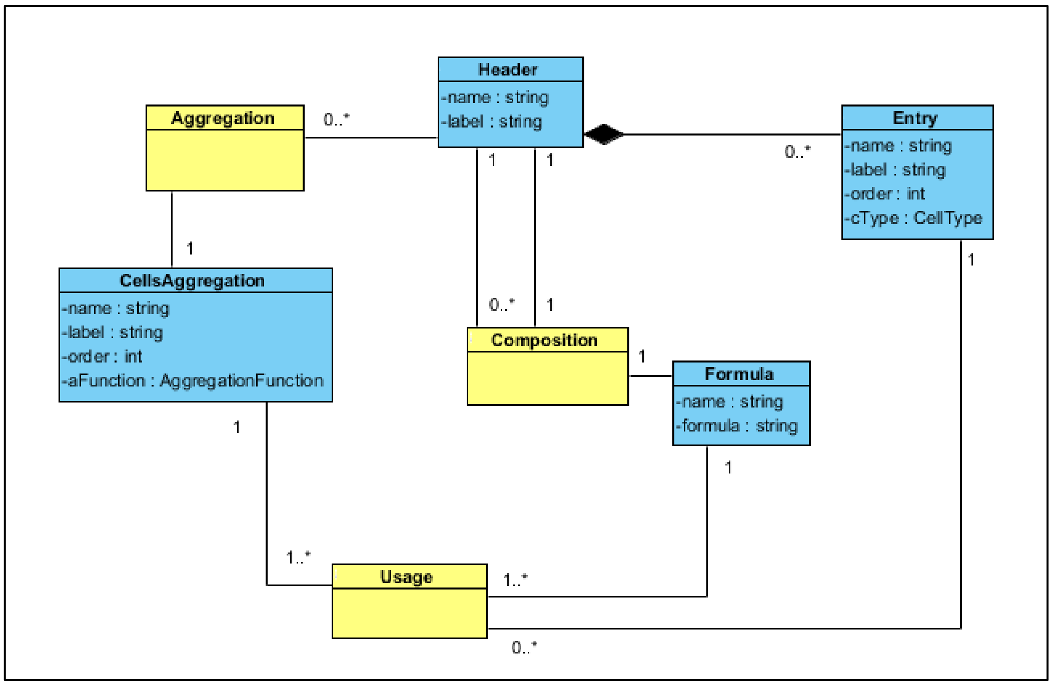


Figure 4.8: Header's Metamodel

4.2 Domain Constraints

4.2.1 Levels Addressed

Cell Level

In the *Sales* spreadsheet of Figure 4.1, for instance, it is possible to specify through our DSL model seen so far that only *TEXT* values are allowed to be entered in each cell of the *Pay Method* column (see *PayM* in Figure 4.3), however, it is not possible to specify the exact concrete *TEXT* values that are permitted, which, in this case, are *Check*, *Credit Card* and *PayPal*.

To address this issue, UML's *Enumeration* and *EnumerationLiteral* entities (Figure 4.9) are used to define the set of values allowed, with each one of them matching a literal of the enumeration as Figure 4.10 shows. Moreover, the enumeration is associated to the entry through a usage link, as Figure 4.10 shows.

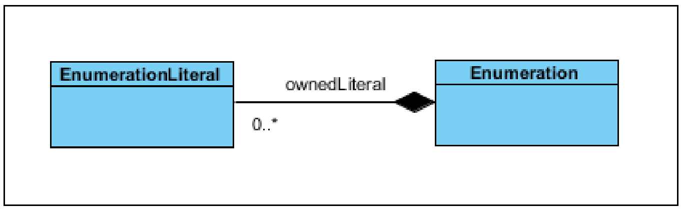


Figure 4.9: Enumeration's Metamodel

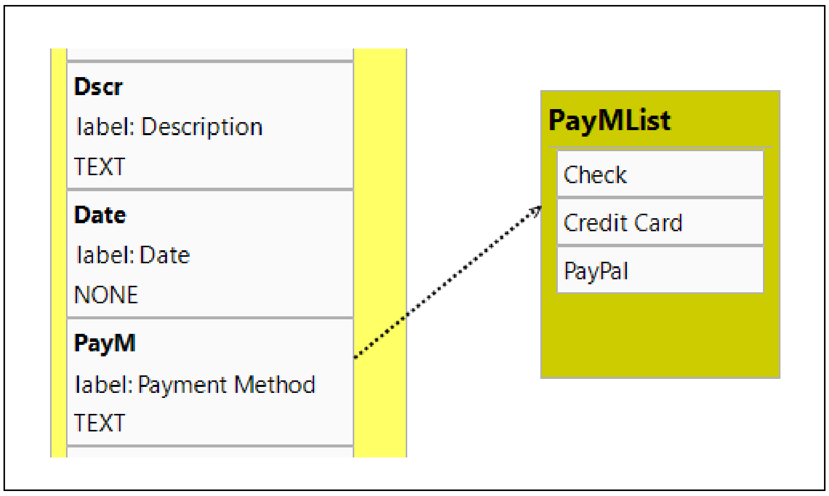
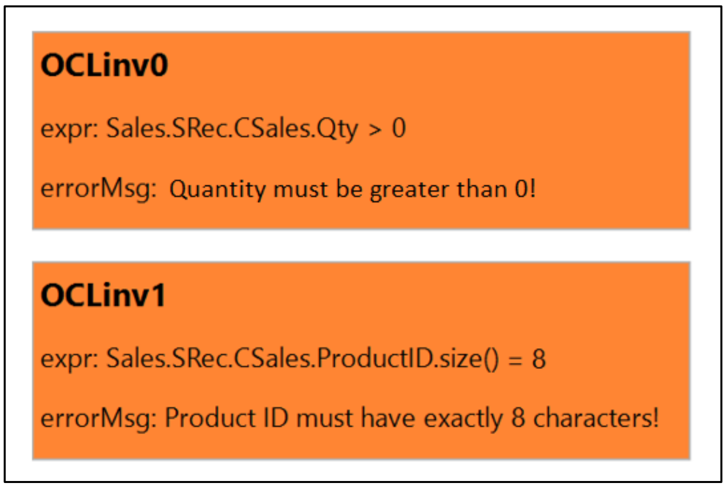


Figure 4.10: Enumeration Example

Nevertheless, we may be want to specify a larger set of values or define a range of values. For example, *Quantity* must certainly have a value above 0 and the *Product ID* must have exactly 8 characters. In these cases, the use of enumerations is obviously impracticable.

In order to achieve more expressiveness, a portion of OCL's invariants are used. Figure 4.11 shows the two OCL invariants used to address the respective value restrictions. To specify which entry is being referenced, the full path from the workbook to the entry is written down.



OCLinv0

expr: Sales.SRec.CSales.Qty > 0

errorMsg: Quantity must be greater than 0!

OCLinv1

expr: Sales.SRec.CSales.ProductID.size() = 8

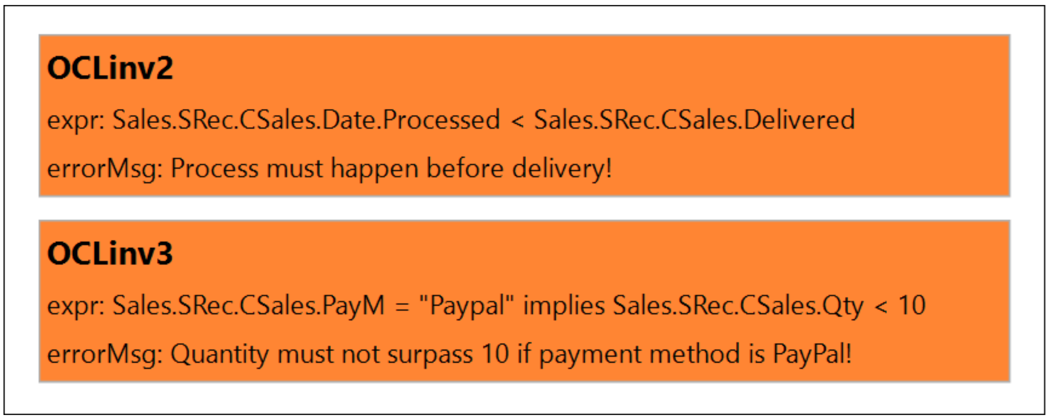
errorMsg: Product ID must have exactly 8 characters!

Figure 4.11: Cell Level Restrictions

Record Level

Besides the cell level, the entries may have value dependencies between each other on the record level. The *Processed DATE* must be certainly lower than the *Delivered DATE*.

Also, it may be wanted, for instance, a restriction to bound the *Quantity* to order if the *Payment Method* equals *PayPal*. Figure 4.12 presents the corresponding OCL invariants to address these two constraints.



OCLinv2

expr: Sales.SRec.CSales.Date.Processed < Sales.SRec.CSales.Delivered

errorMsg: Process must happen before delivery!

OCLinv3

expr: Sales.SRec.CSales.PayM = "Paypal" implies Sales.SRec.CSales.Qty < 10

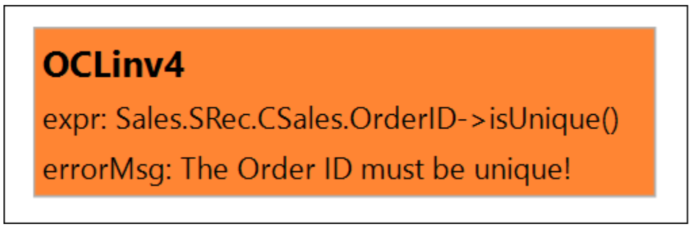
errorMsg: Quantity must not surpass 10 if payment method is PayPal!

Figure 4.7: Record Level Restrictions

Entry Level

We also may want to impose restrictions on the entry level. For instance, all table records must have a unique *Order ID* value. In order to guarantee this invariant, we call on OCL's collection operations as Figure 4.8 shows – a *isUnique* operation is used to check if all the *Order ID* values are different from each other.

Note, however, that the operation of this example is a simplification of the OCL's operation of the same name. In this function no expression is passed as argument to be applied to each value of the collection to then check if all the output values are unique (see all supported operations in Section 4.2.4).



OCLinv4

expr: Sales.SRec.CSales.OrderID->isUnique()

errorMsg: The Order ID must be unique!

Figure 4.8: Entry Level Restrictions

4.2.2 Operations

The operations that are used in the OCL's invariants of this DSL are described in the Tables 4.1, 4.2, 4.2, 4.3, and 4.4. The tabled tabulated operations are OCL Standard [31] operations, except the ones highlighted, which were added to the language to offer the expressiveness needed to cover a significant range of restrictions in the domain of spreadsheets.

In the first table (4.1) we have the *NUMBER* type operations, and in Table 4.2, we have the TEXT type operations. In addition to the standard string operations, we added the operation *toNum* that converts a *NUMBER* value to a *TEXT* value. Also, we added the operation *filter*. This operation returns either a TRUE value or a FALSE value, according if the *TEXT* value matches or not the "filter" – Excel's text search pattern – passed as argument. In Table 4.3, are described the *BOOLEAN* type operations, and in Table 4.4 are tabulated the *DATA* type operations that were all added to the language, so we can use restrictions on *DATA* types values. Lastly, Table 4.5 shows the *COLLECTION* operations. This operations, in the context of spreadsheets, represent operations applied to solo table entries, as showed in Section 4.2.3 example though the use of *isUnique* operation. Moreover, the *product*, *average*, *median*, *max* and *min* functions are analogous to the standard *sum* function.

Table 4.1: NUMBER Operations

NUMBER		
Operation	**Notation**	**Result type**
equals	a = b	BOOLEAN
not equals	a <> b	BOOLEAN
less	a < b	BOOLEAN
more	a > b	BOOLEAN
less or equal	a <= b	BOOLEAN
more or equal	a >= b	BOOLEAN
plus	a + b	NUMBER
minus	a - b	NUMBER
multiply	a * b	NUMBER
divide	a / b	NUMBER
modulus	a.mod(b)	NUMBER
integer division	a.div(b)	NUMBER
absolute value	a.abs()	NUMBER
maximum	a.max(b)	NUMBER
minimum	a.min(b)	NUMBER
round	a.round()	NUMBER
floor	a.floor()	NUMBER

Table 4.2: TEXT Operations

TEXT		
Operation	**Notation**	**Result type**
concatenation	s.concat(string)	TEXT
size	s.size()	NUMBER
to lower case	s.toLower()	TEXT
to upper case	s.toUpper()	TEXT
substring	s.substring(int, int)	TEXT
equals	s1 = s1	BOOLEAN
not equals	s1 <> s2	BOOLEAN
filter	*s.filter(string)*	*BOOLEAN*
to num	*s.toNum()*	*NUMBER*

Table 4.3: BOOLEAN Operations

BOOLEAN		
Operation	**Notation**	**Result type**
or	a or b	BOOLEAN
and	a and b	BOOLEAN
exclusive or	a xor b	BOOLEAN
negation	not a	BOOLEAN
equals	a = b	BOOLEAN
not equals	a <> b	BOOLEAN
implication	a implies b	BOOLEAN
if then else	if a then b1 else b2	type of b

Table 4.4: DATE Operations

DATE		
Operation	**Notation**	**Result type**
equals	*a = b*	*BOOLEAN*
not equals	*a <> b*	*BOOLEAN*
less	*a < b*	*BOOLEAN*
more	*a > b*	*BOOLEAN*
less or equal	*a <= b*	*BOOLEAN*
more or equal	*a >= b*	*BOOLEAN*
day	*a.day()*	*NUMBER*
month	*a.month()*	*NUMBER*
year	*a.year()*	*NUMBER*

Table 4.5: COLLECTION Operations

COLLECTION		
Operation	**Notation**	**Result type**
includes	a->includes(b)	BOOLEAN
is unique	*a->isUnique()*	*BOOLEAN*
sum	a->sum()	NUMBER
product	*a->product()*	*NUMBER*
average	*a->average()*	*NUMBER*
median	*a->median()*	*NUMBER*
max	*a->max()*	*NUMBER*
min	*a->min()*	*NUMBER*

4.3 Errors Addressed

This DSL prevents the occurrence of errors during the two spreadsheet life cycle stages: the development and the usage. Note that the error's risk taxonomy was not taken into consideration because, in contrast to other taxonomies, this is a taxonomy of spreadsheets rather than of errors in a spreadsheet, and the mentioned errors are transversal to any kind of spreadsheets.

Table 4.6: Development errors adressed

Development					
Level	Errors				
	Qualitative	Quantitative			
		Reasoning			Accidental
		Domain	Omission	Pure Logical	
Any	Adressed	Not Adressed	Indirecty. Adressed	Not Adressed	Adressed

In the development stage (Table 4.6) both qualitative and quantitative errors are addressed.

The first ones are dealt with in this stage given the fact that this types of errors are only introduced in the spreadsheet on the modeling phase. In respect to the quantitative errors, the accidental (mechanical) ones committed in this step are prevented, but the reasoning (logical) ones are not fully addressed: the domain errors are not treated because it is assumed that the developer is a domain expert. Moreover, other mechanisms can be built on top of this DSL to ensure correctness of domain implementation, for instance a DSL for the domain. The pure logical errors are also not addressed because spreadsheet systems already have mechanisms to detect some of these kind of errors (e.g., circular reference); lastly, the omission errors occurrence can be attenuated taking into account that the stated spreadsheet modeling offers a much better reasoning, but it cannot ensure the absence of this type of errors.

Table 4.7: Usage errors adressed

Usage					
Level	**Errors**				
	Qualitative	**Quantitative**			
		Reasoning			**Accidental**
		Domain	**Omission**	**Pure Logical**	
Cell	Not Applicable	Adressed	Not Applicable	Not Applicable	Adressed
Record		Adressed			
Entry		Adressed			
Spreadsheet		Not Adressed			

In the usage stage (Table 4.7) both qualitative, omission and pure logical errors are not addressed (they concern the development phase). On the other hand, accidental errors – such as inserting wrong type values or overwriting a formula – are prevented, and domain errors are also dealt with. Business logic restrictions are enforced on the cell, record and entry levels; on the spreadsheet level restrictions are not enforced, since the degree of coupling between worksheets in a spreadsheet is low, as we mentioned in Section 3.1 of Chapter 3.

5

GenSS - A Tool to Generate Spreadsheets

This chapter presents a prototype implementation of the approach described in Chapter 4. We describe the technologies used for the implementation (Section 5.1), the architecture (Section 5.2) of the prototype, along with details about its user interface (Section 5.3). The GenSS prototype's source code is available at: https://github.com/spreadsheetsunl/uml-spreadsheets/tree/master.

5.1 Technologies Used

For the DSL construction and design, we used the following technologies:

- **EMF (Eclipse Modeling Framework)** [32] – a modeling framework for describing models and runtime support for the models.
- **GMF (Graphical Modeling Framework)** [33] – a generative component and runtime infrastructure for developing graphical editors based on EMF.
- **Epsilon** [34] – a set of tools and task-specific languages for code generation, model-to-model transformation, model validation, model merging, among others, that works with EMF and GMF. The Epsilon task-specific languages used in the implementation are namely:

- **EVL (Epsilon Validation Language)** – a OCL alike validation language that supports dependencies between constraints and offers customizable error messages to be displayed to the user, as well as the specification of fixes which users can invoke to fix inconsistency errors.
- **EGL (Epsilon Generation Language)** – a template-based model-to-text language for generating code, documentation and other textual artefacts from models.

- **JavaCC (Java Compiler Compiler)** [35] – a parser generator that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar.

5.2 Architecture

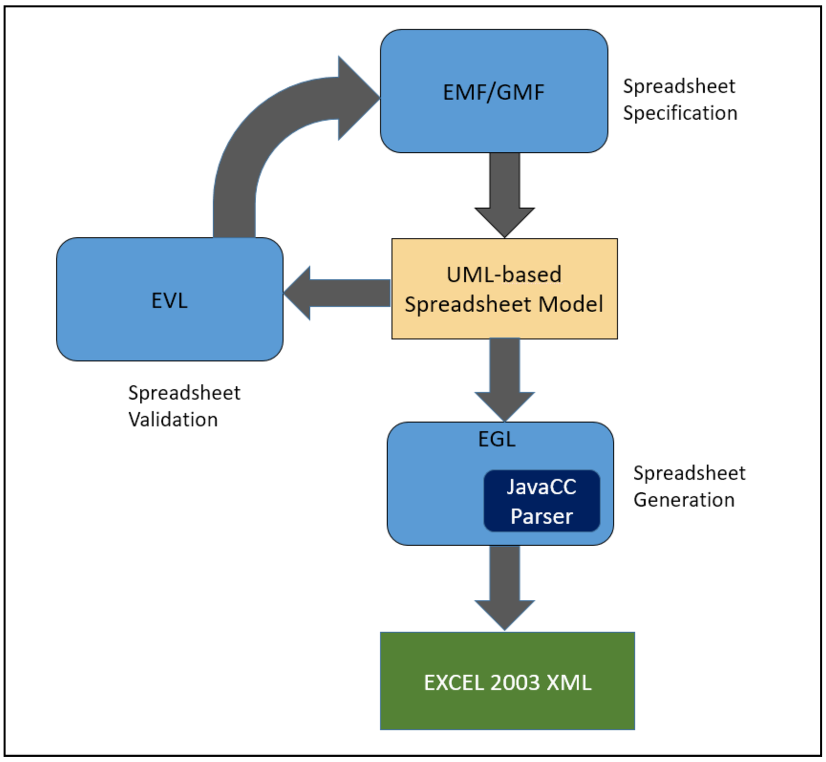


Figure 5.1: Tool prototype's architecture

Figure 5.1 shows the architecture of our tool. The first component concerns the spreadsheet specification using EMF/GMF, in which the spreadsheet's model is described using a visual editor. Then, the specified model is validated using EVL – if the model it is not valid, we go back to the specification step. Lastly, EGL is used to generate the spreadsheet from the defined model, and that includes the translation from the OCL-based invariant expressions to the spreadsheet formulas. The generated spreadsheet consists of an Excel 2003 XML formatted spreadsheet that can be processed by any other spreadsheet system that support that format.

5.3 User Interface

The user interface of our tool is shown in Figure 5.2. The interface consists of a visual editor that allows the user to model a spreadsheet using our DSL constructors. For this purpose, there is a "Palette" of entities divided into "Objects" and "Connections "that can be selected to instantiate the corresponding entity on the model.

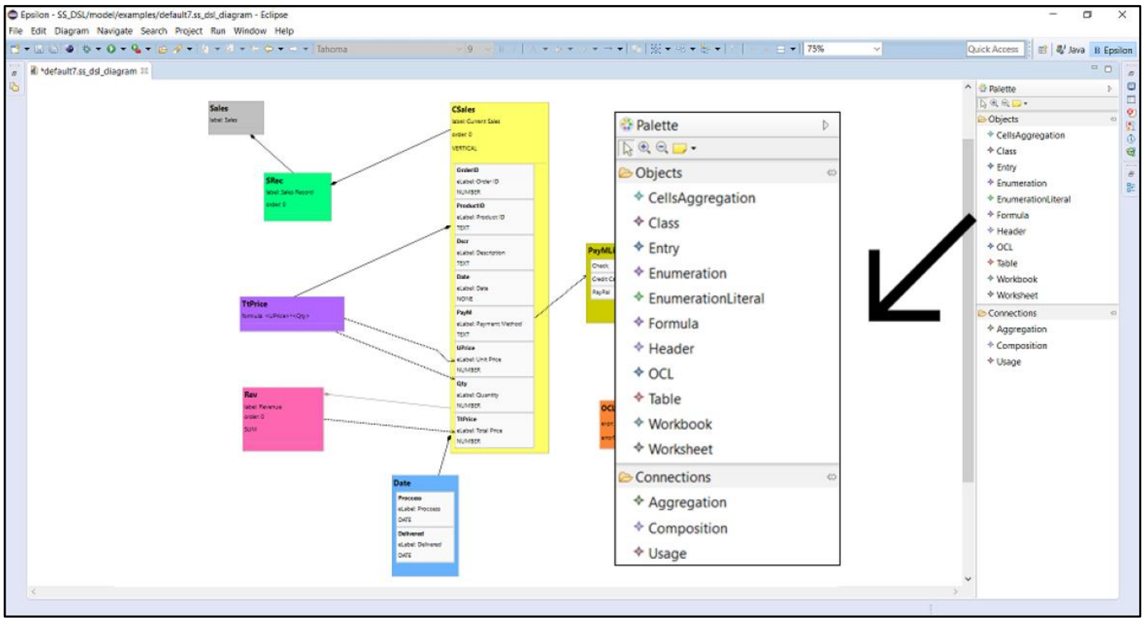


Figure 5.2: Tool's user interface

6

Case Study

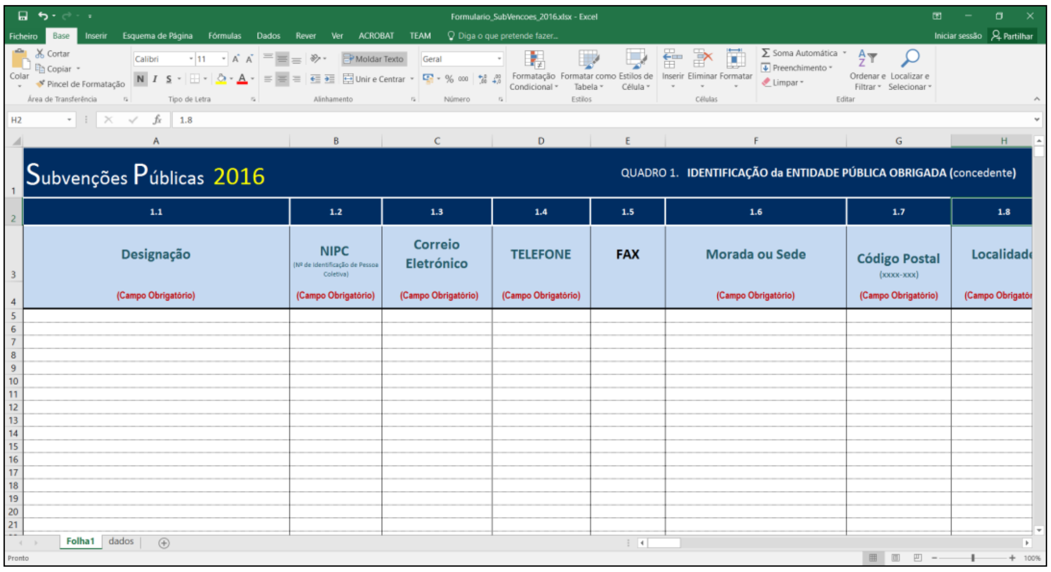
The case study for this dissertation consists of a real-world spreadsheet built by a Portuguese financial public entity – IGF (Inspeção-Geral de Finanças) – responsible for the evaluation and strategic control of the financial administration of the State and the specialized technical support to the Ministry of Finances. The spreadsheet under study serves as a collector of the data regarding the beneficiaries of the annual public subsidies in 2016. The beneficiary entities are responsible for filling the spreadsheet with their data, and then to send it back to the financial entity that validates the received data and imports it to a database.

During the meetings with IGF, they reported that is quite common the introduction of errors in the spreadsheet during the data insertion by the users. Our interest is to show how we can increase the robustness of the spreadsheet through the use of this work's DSL and tool.

The spreadsheet is a workbook composed by two worksheets. Figure 6.1 shows part of the first worksheet - the main one. This worksheet has a solo vertical table with multiple header hierarchies – as the pattern mentioned in Section 3.2.2 -, with a total number of 56 table entries.

Each entry is one of three types: *DATE*, *NUMBER* or *TEXT*. For the last two types there are data validations that ensure that the values entered correspond to the respective defined type. Also, list values are used to restrict the possible entering values; those list values are defined in the second worksheet (see Figure 6.2).

Lastly, there are also comments that inform the user about the format of the values that are expected and the existing dependencies between them.

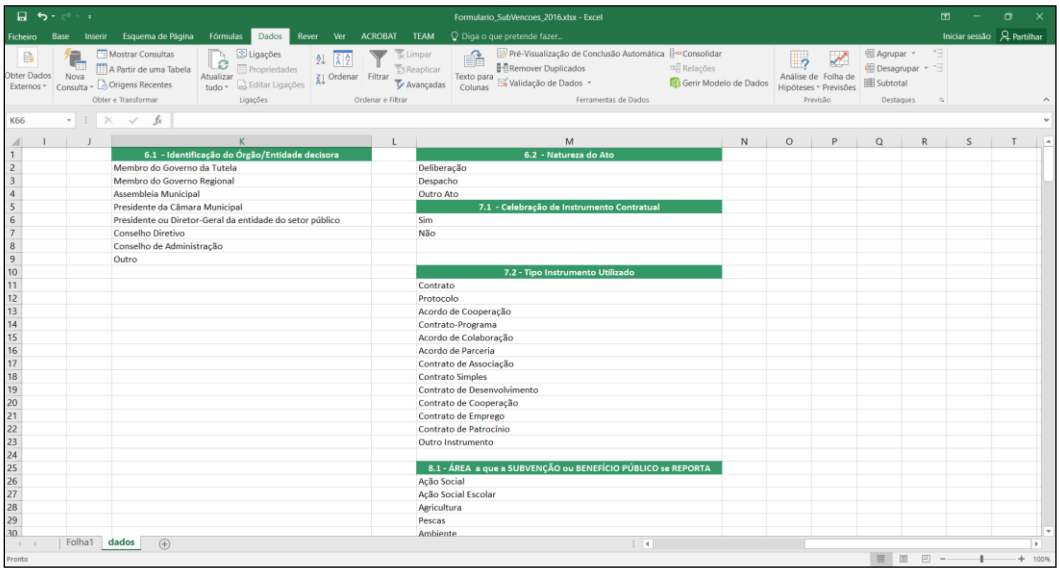


Subvenções Públicas 2016

QUADRO 1. IDENTIFICAÇÃO da ENTIDADE PÚBLICA OBRIGADA (concedente)

1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8
Designação (Campo Obrigatório)	NIPC (Nº de Identificação de Pessoa Coletiva) (Campo Obrigatório)	Correio Eletrónico (Campo Obrigatório)	TELEFONE (Campo Obrigatório)	FAX	Morada ou Sede (Campo Obrigatório)	Código Postal (xxxx-xxx) (Campo Obrigatório)	Localidade (Campo Obrigatório)

Figure 6.1: Part of Worksheet 1 of the IGF Spreadsheet



6.1 - Identificação do Órgão/Entidade decisora	6.2 - Natureza do Ato
Membro do Governo da Tutela	Deliberação
Membro do Governo Regional	Despacho
Assembleia Municipal	Outro Ato
Presidente da Câmara Municipal	7.1 - Celebração de Instrumento Contratual
Presidente ou Diretor-Geral da entidade do setor público	Sim
Conselho Diretivo	Não
Conselho de Administração	
Outro	
	7.2 - Tipo Instrumento Utilizado
	Contrato
	Protocolo
	Acordo de Cooperação
	Contrato-Programa
	Acordo de Colaboração
	Acordo de Parceria
	Contrato de Associação
	Contrato Simples
	Contrato de Desenvolvimento
	Contrato de Cooperação
	Contrato de Emprego
	Contrato de Patrocínio
	Outro Instrumento
	8.1 - ÁREA a que a SUBVENÇÃO ou BENEFÍCIO PÚBLICO se REPORTA
	Ação Social
	Ação Social Escolar
	Agricultura
	Pescas
	Ambiente

Figure 6.2: Part of Worksheet 2 of the IGF Spreadsheet

We assume that there were no development errors and the spreadsheet is correctly modulated for the intended purpose. In respect to the usage errors, although the spreadsheet already avoids some usage errors directly (type validations) and use strategies to mitigate them (comments), a larger majority of the errors are still unaddressed, specifically:

- Accidental errors are not prevented (e.g., the user can modify the list values defined in the second spreadsheet).
- Domain constraints on the cell level are not addressed (e.g., an entry of type *TEXT* for a ZIP Code has a specific text formatting, but is not possible to ensure that only values that meet the condition are entered by the user; also, there is no validation on the values of entries with type *DATE*).
- Domain constraints on the record level are not addressed (e.g., entries of type *DATE* have certain conditional dependencies between them such as one being lesser than a certain other, nevertheless the user can insert values that violate that restriction.).

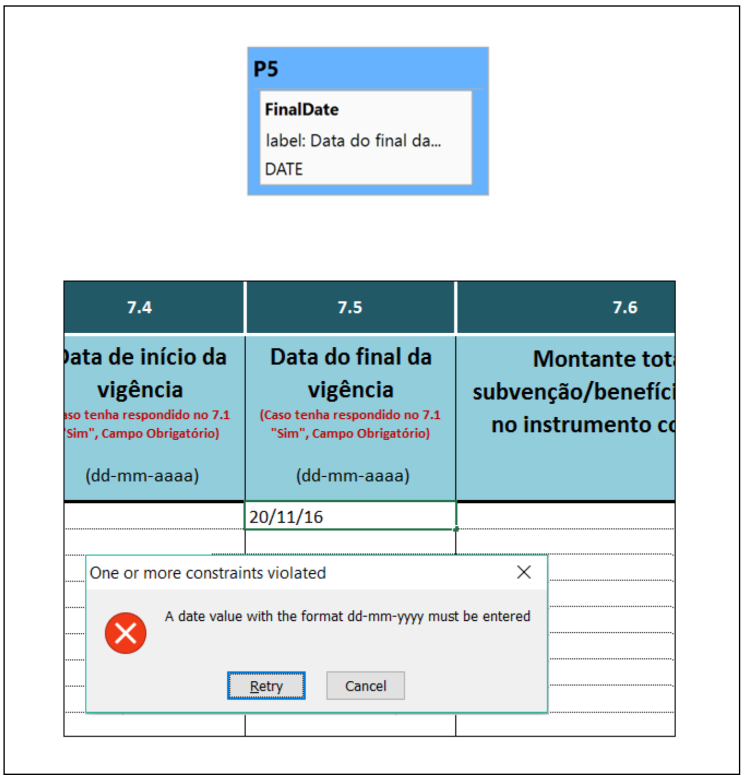
Through the use of this work's DSL and tool it was possible to address the listed usage errors up to a certain level. Firstly, accidental errors are prevented, since the list values used are protected and hidden from the user. In fact, all the cells that are not supposed to be modified by the user, e.g., cells associated with formulas or labels, are also protected. The cells to be protected are implicitly determined by the model, since the cells derived from certain model entities, e.g., Formula or Header, are generated with a "protected" flag.

Table 6.1: Number of Domain Constraints not addressed

	Domain Constraints Not Addressed	
	Cell Level	**Record Level**
Original	16/56	0/54
Using GenSS	2/56	34/54

In respect of the domain constraints, Table 6.1 shows the existing domain constraints to be addressed (both at the cell and record levels) in the original spreadsheet versus the generated spreadsheet by our tool.

Of the 56 entries of the spreadsheet's vertical table (Figure 6.1), the original spreadsheet still had 16 non-error free entries, while the spreadsheet generated from our tool reduced that number to 2.


P5

FinalDate

label: Data do final da...

DATE

7.4	7.5	7.6
)ata de início da vigência (aso tenha respondido no 7.1 'Sim", Campo Obrigatório) (dd-mm-aaaa)	Data do final da vigência (Caso tenha respondido no 7.1 "Sim", Campo Obrigatório) (dd-mm-aaaa)	Montante tot subvenção/benefíci no instrumento co
	20/11/16	

One or more constraints violated

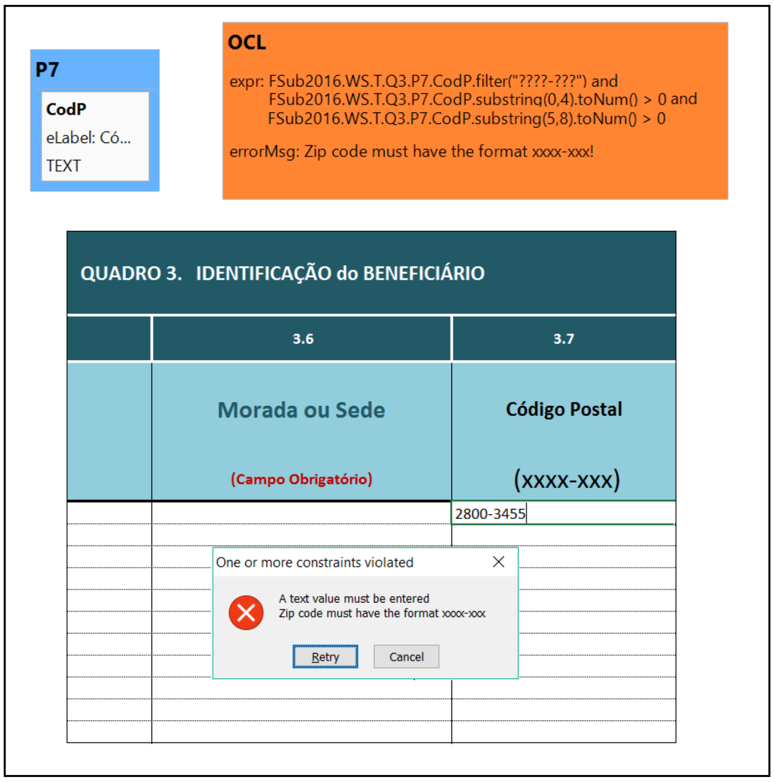
A date value with the format dd-mm-yyyy must be entered

Figure 6.3: DATE type validation

For instance, we can ensure that the user only enters valid values of type *DATE*. In Figure 6.3 it is shown the type definition of the entry, defining it as *DATE*, and the corresponding error prompt when a wrong value is entered.

Figure 6.4 shows another example of a restriction applied on the cell level. In this case, the local ZIP Code is validated.

The 2 non-error free entries not addressed by our approach are entries for the insertion of the Tax Identification Number. The checksum algorithm representation in a spreadsheet formula surpasses the number of 255 characters allowed for data validation in the spreadsheet system.



```
expr: FSub2016.WS.T.Q3.P7.CodP.filter("????-???") and
      FSub2016.WS.T.Q3.P7.CodP.substring(0,4).toNum() > 0 and
      FSub2016.WS.T.Q3.P7.CodP.substring(5,8).toNum() > 0

errorMsg: Zip code must have the format xxxx-xxx!
```

A text value must be entered
Zip code must have the format xxxx-xxx

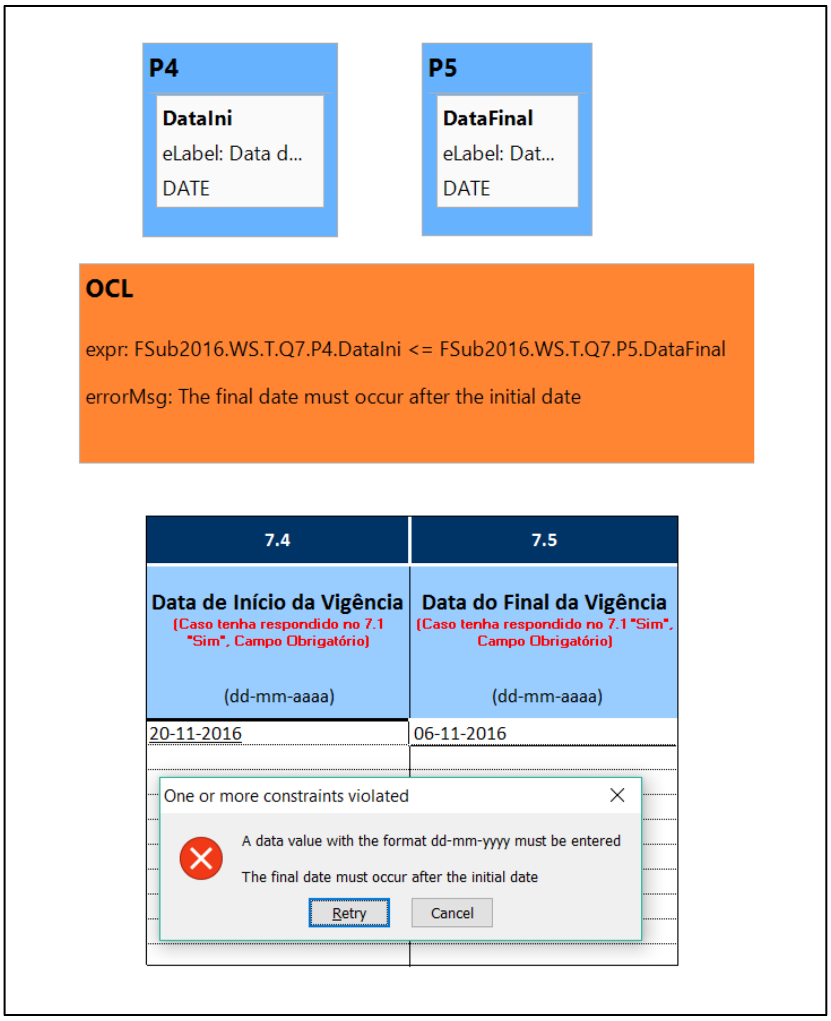
Figure 6.4: Domain restriction at the cell level example

Regarding the domain restrictions at the record level, the original spreadsheet does not address any of them, with a total of 54 dependencies between entries not prevented from possible violation, while the spreadsheet generated from our tool addresses 20 dependencies, thus reducing the number of dependencies not addressed to 34. For example, in Figure 6.5, we see a validation of a dependency between two data entries, where it is guaranteed that the start date is never higher than the end date.

However, the limitations of the spreadsheet system make some errors not addressable by our approach - cells associated with lists of values cannot have associated data validation formulas too.

A possible way to overcome these limitations of the system (maximum number of characters and unique type of validation per cell) in future versions of our tool, is to use specific script languages for each specific spreadsheet system (e.g, VB in MS Excel) to address constraints regardless of the mentioned system limitations.

Nevertheless, in the current version of our tool we already have a substantial level of effectiveness, as we saw in Table 6.1.



expr: FSub2016.WS.T.Q7.P4.DataIni <= FSub2016.WS.T.Q7.P5.DataFinal

errorMsg: The final date must occur after the initial date

7.4	7.5
Data de Início da Vigência (Caso tenha respondido no 7.1 "Sim", Campo Obrigatório) (dd-mm-aaaa)	Data do Final da Vigência (Caso tenha respondido no 7.1 "Sim", Campo Obrigatório) (dd-mm-aaaa)
20-11-2016	06-11-2016

One or more constraints violated

A data value with the format dd-mm-yyyy must be entered

The final date must occur after the initial date

Figure 6.5: Domain restriction at the record level example

6.1 State of the Art Techniques

Although some of the approaches mentioned in Section 2.4 of Chapter 2 prevent accidental errors during the usage stage (e.g., Embedded ClassSheets), none of them can derive completely the original spreadsheet of this case study - header hierarchy structures and the list values are not possible to model.

Moreover, none of those approaches can derive the domain constraints needed to prevent domain errors.

This demonstrates the clear advantage of our approach in comparison with other existing techniques in terms of expressiveness.

7 Conclusion

Several types of research have proposed approaches to offer a better understanding of spreadsheets, improving work performance and preventing errors. Nevertheless, none of them adequately addresses real-word spreadsheet's patterns, being specific to a single pattern. Moreover, none of those techniques includes mechanisms to express arbitrary constraints on top of the model in order to address domain errors, and, consequently, granting a larger cover of spreadsheet errors.

Adopting a Model-Driven approach, the work in this dissertation increases the expressiveness of a model specification of a spreadsheet. It allows the generation of spreadsheets that are more close to the user's needs regarding the domain of the problem (and not regarding the solution domain), and allowing him/her to specify domain restrictions, and thus helping the enforcement of the spreadsheet's business logic.

To achieve this, we proposed a UML-based class diagram DSL that allows the specification of a spreadsheet with a higher level of expressiveness than a standard UML model, and, in addition, the use of some OCL invariant language constructors that allows a solid sub-set of arbitrary constraints over the defined model.

Lastly, we used this DSL to overcome the deficiencies of a real-world spreadsheet regarding its error-proneness, reducing it and, thus, showing the effectiveness of this work's approach.

7.1 Future Work

The currently modelling language requires an expert to create models. We intend to create a family of DSL's, so spreadsheet end users can build these models themselves with a more natural way. From this, we will generate our models. The second direction for future research could be the generation of these models from the database schema itself, since many spreadsheets are created to be interfaces to databases.

References

[1] Power, D. J., A Brief History of Spreadsheets, DSSResources.COM, http://dssresources.com/history/sshistory.html, 2004 (visited: 2016-06-22)

[2] Scaffidi, C., Shaw, M., and Myers, B.A., Estimating the numbers of end users and end user programmers. In Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp. 207–214, 2005.

[3] Panko, R., and Ordway, N., Sarbanes-Oxley: What About all the Spreadsheets? Controlling for Errors and Fraud in Financial Reporting. EuSpRIG 2005 Proceedings, pp. 2-3, London, 2005.

[4] Durfee, Don. Spreadsheet Hell. CFOs are interested in the many new technologies being pitched to them, but are they really trapped in spreadsheet hell?, CFO.COM, http://www.cfo.com/printable/article.cfm/3014451, 2004 (visited: 2016-01-22)

[5] Panko R., Facing the problem of spreadsheet errors, Decision Line,. vol. 37, no. 5, 2006.

[6] EuSpRIG. European Spreadsheet Risks Interest Group. http://www.eusprig.org/ (visited: 2016-01-22).

[7] EuSpRIG. EuSpRIG Horror Stories. http://www.eusprig.org/horror-stories.htm/ (visited: 2015-01-22).

[8] Kent, S., Model Driven Engineering, in IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods, pp. 286-298, London, 2002.

[9] Cunha, J., Fernandes, J. P., Mendes, J., and Saraiva, J., Embedding and Evolution of Spreadsheet Models in Spreadsheet Systems. In Proceedings of the 2011 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC '11, pp. 179-186, 2011

[10] Antunes, L., Corrêa, A., and Barros, M., Automatic Spreadsheet Generation from Conceptual Models. In 29th Brazilian Symposium on Software Engineering, pp. 140 - 149, Belo Horizonte, 2015

[11] Hermans, F., and Murphy-Hill, E., Enron's Spreadsheets and Related Emails: A Dataset and Analysis, In ICSE' 15: 37th International Conference on Software Engineering, Florence, 2015

[12] Jansen, B., Enron versus EUSES: A comparison of two spreadsheet Corpora, In SEMS'15: Second Workshop on Software Engineering Methods in Spreadsheets, pp. 41-46, Florence, 2015

[13] De Lucia, A., Gravino, C., Oliveto, R., and Tortora, G., An experimental comparison of ER and UML class diagrams for data modelling, In Empirical Software Engineering Volume 15, Issue 5, pp. 455-492, 2010

[14] Kühne, T., What is a Model?, In: Bézivin, J., Heckel, R. (eds.) Procs. Dagstuhl Seminar 04101, Language Engineer-ing for Model-Driven Software Development, 2004

[15] Selic, B., The Pragmatics of Model-Driven Development, IEEE Softw., vol. 20, no. 5, pp.19-25, 2003

[16] Deursen, Arie van, Klint, Paul and Visser, Joost. Domain-specific languages: An annotated bibliography. ACM SIGPLAN Notices, vol. 35, no. 6, pp. 26-36, 2000.

[17] Völter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., L. C. L. Kats, L. C. L., Visser, E., and Wachsmuth, G., DSL Engineering - Designing, Implementing and Using Domain-Specific Languages, dslbook.org, 2013.

[18] Powell, S. G., Baker, K. R., Lawson, B., A critical review of the literature on spreadsheet errors, Decision Support Systems, vol. 46, no. 1, pp. 128-138, 2008

[19] Galletta, D. F., et al, An Empirical Study of Spreadsheet Error – Finding Performance, Accounting, Management, and Information Technology, vol. 3, no. 2, pp. 79-95, 1993

[20] Panko, R., and Halverson, R. P., Jr., Spreadsheets on trial: A survey of research on spreadsheet risks, Proceedings of the 29th Hawaii International Conference on System Sciences, pp. 326-335, 1996

[21] Teo, T. S. H., and Tan, M., Quantitative and Qualitative Errors in Spreadsheet Development, Proceedings of the Thirtieth Hawaii International Conference on System Sciences, Maui, Hawaii, 1997

[22] Rajalingham, K., Chadwick, D. & Knight, B., Classification of spreadsheet errors, Proceedings of the EuSpRIG 2000 Conference, pp. 23-24, University of Greenwich, London, 2000

[23] Madahar, M., Cleary, P. and Ball, D., Categorisation of Spreadsheet Use within Organisations Incorporating Risk: A Progress Report, Proceedings of the European Spreadsheet Risks Interest Group 8th Annual Conference, University of Greenwich, London, pp. 37-45, 2007

[24] Panko, R. R. and Aurigemma, S., Revising the Panko-Halverson taxonomy of spreadsheet errors. Decision Support Systems, vol. 49, pp. 235-244, 2010

[25] Erwig, M., Abraham, R., Cooperstein, I., and Kollmansberger, S., Gencel: A Program Generator for Correct Spreadsheets, Journal of Functional Programming, vol. 16, no. 3, pp. 293-325, 2006

[26] Engels, G., and Erwig, M., ClassSheets: Automatic Generation of Spreadsheet Applications from Object-Oriented Specifications, 20th IEEE/ACM Int. Conf. on Automated Software Engineering, pp. 124-133, 2005

[27] Cunha, J., Fernandes, J. P., and Saraiva, J., From Relational ClassSheets to UML+OCL, In: Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC 2012), pp. 1151–1158, Trento, Italy, 2012

[28] Hermans, F., Pinzger, M., and Deursen, A. van, Atomatically extracting class diagrams from spreadsheets. In Proc. of the 24th European Conference on Object-Oriented Programming, pp. 52-75, Berlin, 2010

[29] Abraham, R., and Erwig, M., Header and Unit Inference for Spreadsheets Through Spatial Analyses. In Proceedings of the 2004 IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 165-172, Rome, 2004

[30] OMG Unified Modeling Language, Version 2.5, Object Management Group, 2015

[31] Object Constraint Language, Version 2.4, Object Management Group, 2014

[32] ECLIPSE, ECLIPSE MODELLING FRAMEWORK, HTTPS://ECLIPSE.ORG/MODEING/EMF/ (VISITED: 2016-01-22)

[33] ECLIPSE, GMF TOLLING, HTTP://WWW.ECLIPSE.ORG/GMF-TOOLING/ (VISITED: 2016-01-22)

[34] ECLIPSE, EPSILON, HTTP://WWW.ECLIPSE.ORG/EPSILON/ (VISITED: 2016-01-22)

[35] JAVA, JAVACC, HTTPS://JAVACC.JAVA.NET/ (VISITED: 2016-01-22)