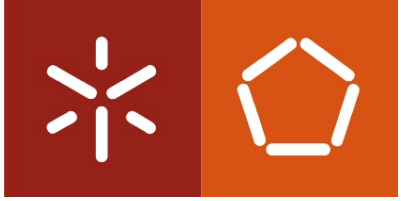


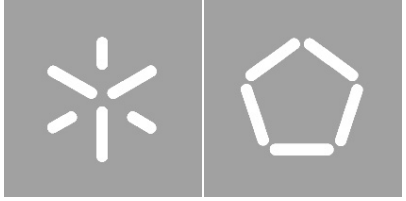
Universidade do Minho
Escola de Engenharia

Jorge Cunha Mendes

Evolution of Model-Driven Spreadsheets

Setembro de 2012

This work is funded by the ERDF through the Programme COMPETE and by the Portuguese Government through FCT - Foundation for Science and Technology, project ref. PTDC/EIA-CCO/108613/2008, grant ref. BI4-2011_PTDC/EIA-CCO/108613/2008.



Universidade do Minho
Escola de Engenharia

Jorge Cunha Mendes

Evolution of Model-Driven Spreadsheets

Tese de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de
Doutor João Alexandre Saraiva
Doutor Jácome Miguel Cunha

Setembro de 2012

DECLARAÇÃO

Nome

_____

Endereço electrónico: _____ Telefone: _____ / _____

Número do Bilhete de Identidade: _____

Título dissertação □ / tese □

_____

_____

_____

Orientador(es):

_____

_____ Ano de conclusão: _____

Designação do Mestrado ou do Ramo de Conhecimento do Doutoramento:

_____

Universidade do Minho, ____ / ____ / _____

Assinatura: _____

Acknowledgements

I would like to thank my supervisors Prof. Dr. João Saraiva and Dr. Jácome Cunha for the opportunity to develop interesting and relevant work which led to a great start for my scientific career, and for the availability and constant support during the development of this thesis. I would also like to thank Dr. João Paulo Fernandes for being like a supervisor to me and for providing me with insightful and supportive comments when needed.

Other people helped me during this work and they are not forgotten, namely Hugo Pacheco with insights on bidirectional transformations and my laboratory colleagues that distracted me enough so I would not go insane. But this work would not have started if it was not for the great support that I had from many teachers during my undergraduate and master's course, and I thank them for that.

Moreover, during this project I was awarded a Research Grant within the SSaaPP – Spreadsheets as a Programming Paradigm project funded by the ERDF through the Programme COMPETE and by the Portuguese Government through FCT – Foundation for Science and Technology (project ref.: PTDC/EIA-CCO/108613/2008, grant ref.: BI4-2011_PTDC/EIA-CCO/108613/2008).

Abstract

Evolution of Model-Driven Spreadsheets

Spreadsheets are likely to be one of the most widely used programming environments in the world. Spreadsheet popularity is due to characteristics such as their low entry barrier, their availability on almost any computer, their simple visual interface but mainly due to their flexibility. This flexibility, however, comes with a cost: spreadsheets are extremely error prone, as indicated by several studies.

The work presented in this thesis aims at tackling the problem of spreadsheet errors. The strategy described is based on a model-driven approach, and is achieved by embedding spreadsheet models within spreadsheets themselves. This embedding enables users to create models in the same environment that they use for spreadsheet development which is precisely the environment that they are familiar with.

Moreover, a set of evolution operations that can be performed on these models and respective instances is defined. In this setting, users interact and evolve both models and spreadsheets in the same coherent environment. This facilitates the establishment and maintenance of a consistency relationship between models and instances throughout the spreadsheet development life cycle, and by this it is expected the reduction of the number of errors that are made and the improvement of productivity in using spreadsheets.

Resulting from this work, a prototype was created and is also discussed in this dissertation. This tool can be used to validate the approach followed in this thesis and to provide a foundational framework for future developments.

Keywords: Spreadsheet; Model-Driven Engineering; Embedded Domain Specific Languages; Bidirectional Transformations; Prototype.

Resumo

Evolução de Folhas de Cálculo Orientadas por Modelos

Folhas de cálculo são provavelmente o ambiente de programação mais usado no mundo inteiro. A sua popularidade advém principalmente da facilidade com que se começa a usá-las, da sua disponibilidade em quase qualquer computador, da sua simples interface visual, mas principalmente da sua flexibilidade. Isto deve-se à falta de restrições impostas por este tipo de sistema, o que pode levar a numerosos erros na maioria das folhas de cálculo, como indicado por numerosos estudos.

O trabalho apresentado nesta tese visa combater o problema de erros em folhas de cálculo. A estratégia descrita baseia-se no uso de modelos e é alcançada embutindo modelos de folhas de cálculo dentro das folhas de cálculo em si. Esta embutidura possibilita aos utilizadores criar modelos no mesmo ambiente em que desenvolvem as suas folhas de cálculo, com o qual já estão habituados.

Mais, um conjunto de operações sobre esses modelos e respectivas instâncias também foi definido. Deste modo, utilizadores podem interagir com modelos e folhas de cálculo dentro do mesmo ambiente. Isto facilita o estabelecimento e manutenção de uma relação de consistência entre modelos e dados durante o ciclo de vida de folhas de cálculo, esperando-se que se reduza o número de erros cometidos e que se aumente a produtividade usando folhas de cálculo.

Um protótipo foi criado como resultado deste trabalho, e também é discutido nesta dissertação. Esta ferramenta pode ser usada para validar a abordagem escolhida nesta tese e também fornece uma base de trabalho para desenvolvimentos futuros.

Contents

Acronyms

API	Application Programming Interface
DSL	Domain-Specific Language
FFI	Foreign Function Interface
IDE	Integrated Development Environment
MDE	Model-Driven Engineering
MDSE	Model-Driven Spreadsheet Engineering
MDSD	Model-Driven Software Development
SSRB	Spreadsheet Standards Review Board
USA	United States of America
WYSIWYG	What You See Is What You Get

List of Figures

List of Tables

Chapter 1

Introduction

People recognize the importance of structuring data in a tabular-like interface using them for many years. The Plimpton 322 tablet (figure 1.1), dated from around 1800 BC [Robson, 2001], is a case of a table containing four columns and fifteen rows with numerical data. For each column there is a descriptive header, and the fourth column contains a numbering of the rows from one to fifteen. This tablet contains Pythagorean triples [Bruins, 1949], but was more likely built as a list of regular reciprocal pairs [Robson, 2001].

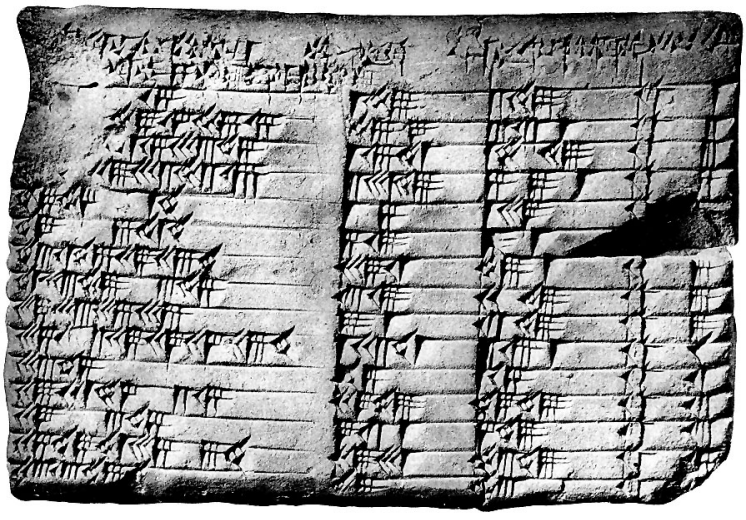


Figure 1.1: Plimpton 322 – a tablet from around 1800 BC.[1]

[1] Source: http://www.math.ubc.ca/~cass/courses/m446-03/pl322/pl322.html. (2012-09-01)

Tables provide an easy way to visualize and organize data, which can be unidimensional (figure 1.2a), two-dimensional (figure 1.2b) or multidimensional (figure 1.2c). The tabular layout allows a systematic analysis

Name	Age
John	43
Mary	36
Will	39

(a) Unidimensional data table with a list of persons and respective ages.

	2010	2011
John	1	3
Mary	3	2
Will	3	3

(b) Two-dimensional data table showing the number of publications for a pair of author/year (rows/columns, respectively).

		2010	2011
John	long	0	2
	short	1	1
Mary	long	2	1
	short	1	1
Will	long	1	3
	short	2	0

(c) Multidimensional data table (three dimensions in this case) similar to figure 1.2b but with additional information about the paper (long or short).

Figure 1.2: Representation of several table dimensions.

of the information displayed and helps to structure values in order to perform calculations. These benefits make tables applicable to a great variety of domains, e.g., mathematics, finance, and even games (see figure 1.3).

A widely used table-like structure is the spreadsheet. The term *spreadsheet* originated before computers from the use of tables that were spread across two pages (e.g., records in a ledger). This term came to mean *table of data arranged in columns and rows often used in business and financial applications* [Spreadsheet, 2012].

Even on paper, spreadsheets are useful for many purposes: student inquiries or exams, taxes submission, budgeting, gathering and analysis of sport statistics, or any purpose that requires input of data and/or performing calculations. Moreover, the use of spreadsheets can be for both personal and professional purposes.

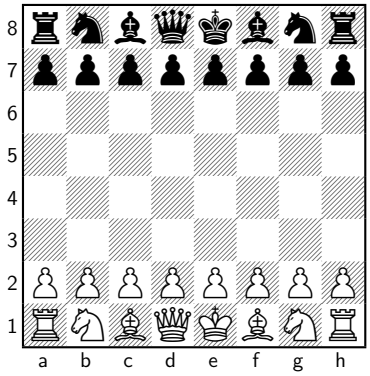


Figure 1.3: Chess boards have a tabular layout, with letters identifying columns and numbers identifying rows.

An example of a simple spreadsheet on a sheet of paper for students to have a multiplication table is displayed in figure 1.4. This spreadsheet has eleven columns and eleven rows, and the students should fill the empty cells (except the top-left one) with the result of multiplying the value of the topmost cell of that column and the value of the leftmost one of that row.

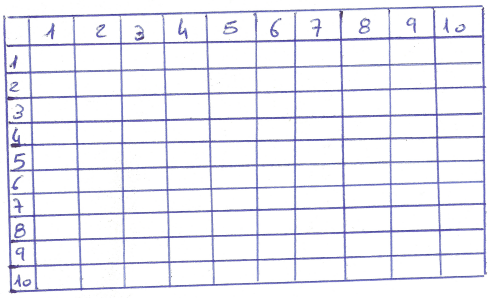


Figure 1.4: Paper spreadsheet for a multiplication table.

With the advent of personal computers, spreadsheets began to be used electronically. This allowed users to use a familiar environment to save data and to perform calculations automatically. Many spreadsheet host systems (programs to edit electronic spreadsheets) appeared, starting with batch spreadsheet report generators as a method to bring budgeting to computers [Mattessich, 1961]. Most of those systems were targeted to timesharing systems, which were too expensive for some companies and had not much use for individuals.

In 1979, VisiCal [Bricklin] was shown to the public targeting the Apple II microcomputer with a later version for IBM PC in 1981. This made spreadsheets available to a wider audience, but also led to make personal computers more popular by introducing them to the financial and business communities and others. VisiCal consisted of a column/row tabulation program with an What You See Is What You Get (WYSIWYG) interface. It provided cell references (format `A1`, `A3..A6`; very similar with the one used for chess boards as the one depicted in figure 1.3) and instant automatic recalculation of formulas, among other novel features still present in current spreadsheet host systems.

After VisiCal, many spreadsheet host systems were developed, but not many obtained huge success among competitors. One commercial spreadsheet system that is still successful is Microsoft Excel[2], first released in 1985 for Macintosh, but other commercial spreadsheets are available (e.g., Corel Quattro Pro and Apple Numbers). However, free open source alternatives can be used, namely Gnumeric[3], OpenOffice Calc[4] and derivatives like LibreOffice Calc[5]. Moreover, web-based spreadsheet host systems have recently been developed, e.g., Google Drive[6], Microsoft Office 365[7], and ZoHo Sheet[8]. These systems are not dependent on any particular operating system, allow to create and edit spreadsheets in an online collaborative environment, and provide import/export of spreadsheet files for offline use. There are many other spreadsheet

[2]Microsoft Excel: `http://office.microsoft.com/en-us/excel/`
[3]Gnumeric: `http://projects.gnome.org/gnumeric/`
[4]OpenOffice: `http://www.openoffice.org`
[5]LibreOffice: `http://www.libreoffice.org`
[6]Google Drive: `http://drive.google.com`
[7]Microsoft Office 365: `http://www.microsoft.com/en-us/office365/online-software.aspx`
[8]ZoHo Sheet: `http://sheet.zoho.com/`

host systems, either for desktop, mobile devices or online use, open source or proprietary, each one with their own particularities[9].

Spreadsheet systems have evolved into powerful systems. However, the basic features provided by spreadsheet host systems remain roughly the same:

- a spreadsheet is a tabular structure composed by cells, where the columns are referenced by letters and the rows by numbers;
- cells can contain either values or formulas;
- formulas can have references for other cells (e.g., `A1` for the individual cell in column `A` and row `1` or `A3:B5` for the range of cells starting in cell `A3` and ending in cell `B5`);
- instant automatic recalculation of formulas when cells are modified;
- ease to copy/paste values, with references being updated automatically.

This last feature can help spreadsheet users if well used as demonstrated below. Further more advanced features are also available in many of those systems, but other features that are relevant for this work are discussed later.

Revisiting the multiplication spreadsheet example from figure 1.4, one can easily represent it in an electronic spreadsheet as shown in figure 1.5, where the cells to be filled in have a formula to compute automatically the result. The formula was created carefully in a way that copy/pasting it would refer to the correct cells in the column and row. To fill in the formulas, cell `B2` was set with formula `B$1*$A2` fixing the row for the first reference (using character `$`) and fixing the column for the second reference. This allowed to copy/paste the formula on the remaining empty cells, where the non-fixed elements of the references were set to the column and row of the pasted formula (e.g., formula in `D7` is `D$1*$A7` and the one in `I4` is `I$1*$A4`). Moreover, cell `D3` is selected displaying the formula for that cell (`D$1*$A3`) instead of the respective result (6).

When creating from scratch a filled spreadsheet for a multiplication table like the one mentioned earlier (figure 1.4), one can notice that the electronic version (figure 1.5) is easier, faster, and with a nicer and more versatile result than the paper one, but almost as intuitive.

[9] List of spreadsheet host systems: `http://en.wikipedia.org/wiki/List_of_spreadsheet_software`

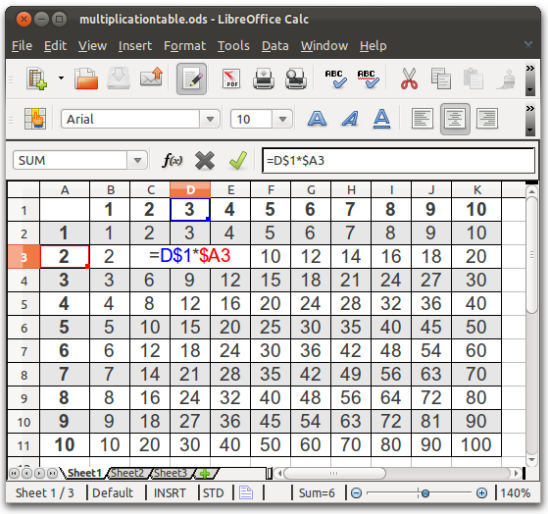


Figure 1.5: Electronic spreadsheet for a multiplication table.

1.1 Motivation

Electronic spreadsheets (just spreadsheets from now on) are easy to use for non-professional programmers, the so-called *end users* [Nardi, 1993], providing a tool which do not impose too restrictive barriers, and allowing a rapid development of large systems by single users.

The freedom that users find in spreadsheets is an advantage, but also a disadvantage: users can do almost everything, even make mistakes that can lead to errors with huge expenses. Several cases of spreadsheet errors have been reported, some leading to financial losses or staff dismissal (see below).

Many studies show that spreadsheets are heavily used by companies, and that they are used to make important decisions. For example, financial intelligence firm CODA reports that 95% of United States of America (USA) firms use spreadsheets for financial reporting according to its experience [Panko and Ordway, 2008]. Another example is brought by RevenueRecognition.com (now Softrax), that had the International Data Corporation interview 118 business leaders in 2004, finding that 85% were using spreadsheets in financial reporting and forecasting [RevenueRecognition.com, 2004]. Moreover, a survey to a company shows that 50% of all spreadsheets are the basis for decisions [Hermans et al., 2011].

From several audit studies to real-world operational spreadsheets, errors were found in more than 94% of the audited spreadsheets, with many errors found in the spreadsheets [Panko and Ordway, 2008]. Moreover, many cases of spreadsheet errors with impact on critical components of businesses and reputation damage have been reported, e.g.:

- A Florida construction company underbid a project by a quarter of a million dollars due to not updating range after items were added [Ditlea, 1987];
- A type error caused an operating fund of the Colorado Student Loan Program to be understated by $36,131 [U.S. Department of Education, 2003];
- A reference error caused a hospital to overclame its Medicare reimbursement by $38,240 [U.S. Department of Health and Human Services, 2003];
- An accounting error resulted in a firm expecting an income higher £4,300,000, which led to the resignation of its chief executive [Daily Express Reporter, 2011];
- A bad link caused a $6,000,000 reporting error, made by the Knox County Trustee's Office, costs taxpayers $12,500 [Donila, 2011];
- A bad formula led West Baraboo officials to calculate borrowing interest low by $400,000 [Bridgeford, 2011].

A large source of information about spreadsheets and risk management associated with them is available at the EuSpRIG website [EuSpRiG, a], where other cases of spreadsheet errors like the ones above can be found in their *horror stories* [EuSpRiG, b].

In the last years, the research community has been working to improve this scenario. In order to minimize spreadsheet errors, several techniques can be used:

- usage of best practices;
- spreadsheet testing; or,
- spreadsheet modeling.

The use of best practices in spreadsheet engineering helps to create spreadsheets that are less error prone, that are easier to understand and that make work more efficient. In 2002, Grossman presented guidelines for spreadsheet engineering, based on previous software engineering research and practice [Grossman, 2002]. Moreover, several entities have published standardized processes for spreadsheet engineering, e.g., FAST [FAST, 2010] and Spreadsheet Standards Review Board [SSRB, 2010]. A comparison of three of these spreadsheet engineering methodologies has been made by Grossman and Ozluk 2010, including the ones from FAST and SSRB. This last one has an accompanying tool (as a Microsoft Excel add-in) to help develop spreadsheets following it [BPM].

Best practices can help build a spreadsheet, but it applies mainly to the creation of new spreadsheets. To apply best practices to legacy spreadsheets, one needs to convert them, which can be toilsome and migration errors can be made. An alternative is to correct only parts of the spreadsheet, applying some best practices without any major changes (depending on the complexity and condition of the spreadsheet). This can be done by detecting spreadsheet smells [Hermans et al., 2012; Cunha et al., 2012b], a concept deriving from software smells [Fowler, 1999], which are a characteristic that may cause problems understanding, updating or evolving a software artifact. It is possible to remove some of these problems refactoring the spreadsheets to eliminate the smell that they contain.

Nevertheless, best practices do not ensure an error-free spreadsheet and other techniques can be used to check if the input and the result of the formulas is correct doing some testing. Spreadsheet testing [Rothermel et al., 2001; Pryor, 2008; Abraham and Erwig, 2008] provides a way to check the semantics of the spreadsheet, checking if the contents of the cells correspond to the expected values, either in input cells (e.g., check bounds of an integer number) or in the result of formulas (e.g., compare the result against the expected one for the given input). Tests for spreadsheet formatting can also be performed, but that only ensures that the spreadsheet follows a predetermined set of best practices.

Another verification that can be done is spreadsheet auditing, but it usually means to manually inspect the spreadsheets and their values and formulas. This technique does not verify results of the formulas per se, but the computation that they perform, which can be a laborious and tedious task.

Another approach that one can undertake is the use of concrete models, in contrary to the theoretical ones from best practices. Model-Driven Engineering (MDE) is a development methodology in software development that uses abstraction through modeling to specify a piece of software. Domain-Specific Languages (DSLs) for modeling can be used to formalize the application structure, behavior, and requirements within a particular domain, imposing domain-specific constraints and performing model checking that can detect and prevent many errors in early stages of the development process [Schmidt, 2006].

The same concepts have already been proposed for spreadsheet development, like spreadsheet templates [Abraham et al., 2005], ClassSheets [Engels and Erwig, 2005] and class diagrams for spreadsheet specification [Hermans et al., 2010].

MDE can be used to restrict spreadsheets in ways to prevent mistakes and also to provide better understanding of large spreadsheets in a small, precise and concise model. Using ClassSheets, the errors stated above could have been prevented since: ranges are automatically updated using expandable classes; a basic type system is provided; and, references are always named, preventing basic mistakes.

The spreadsheet modeling approach is taken as basis for this thesis, since it has solid principles in software engineering with some already applied to spreadsheets. Moreover, it has the potential to be very beneficial for end users, preventing them to make mistakes, improving their performance and documenting the spreadsheets.

1.2 Model-Driven Spreadsheet Engineering

Modeling of spreadsheets was already proposed by several researchers. Ireson-Paine introduced in 1997 *Model Master* [Ireson-Paine, 1997], a compiler and an object-oriented textual language to specify spreadsheets. The idea is that a textual representation of spreadsheets and the specification of formulas using named references instead of cell references (e.g., `$A$1`) is less error-prone than just editing the spreadsheet. Then, the compiler would generate a spreadsheet conforming to the given model. However, these models cover a limited kind of spreadsheets, namely database-like tables (a header and a set of rows).

Erwig et al. developed in 2005 a tool dubbed *Gencel* [Erwig et al., 2005]. This tool takes a template as input and generates a spreadsheet with machinery to restrict some user operations to only those that are logically and technically correct for that template. Along with the restrictions, new operations are also added to perform some repetitive tasks like the repetition of a set of columns with some default values.This method has the disadvantage that when a template is modified, a new spreadsheet is generated and the user has to migrate the data manually.

Abraham et al. introduced a visual specification language for spreadsheets called *ViTSL* [Abraham et al., 2005]. This language is visually similar to spreadsheets and it has also a formal textual representation. A tool was developed to design ViTSL templates (see figure 1.6) to be given as input to Gencel presented

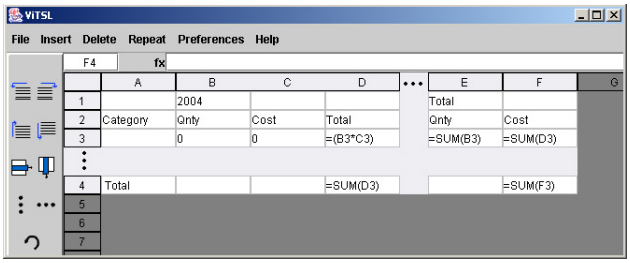


Figure 1.6: Screenshot of the ViTSL editor, taken from [Abraham et al., 2005].

above, but the architecture of this system (ViTSL template editor + Gencel + spreadsheet system, see

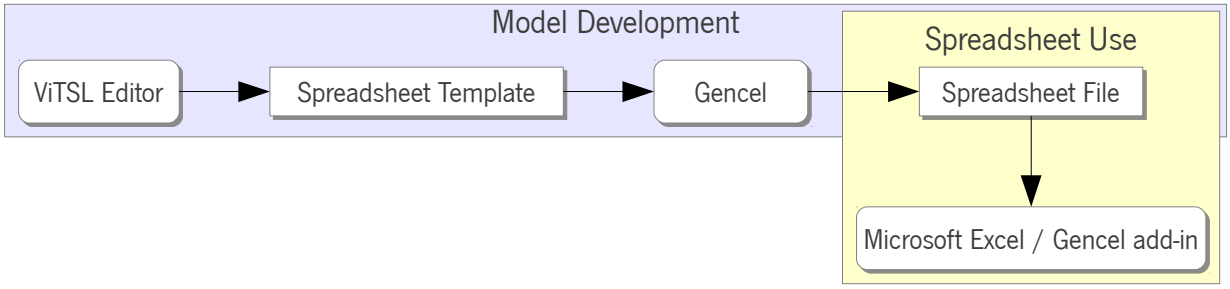


Figure 1.7: ViTSL-based environment for spreadsheet development.

figure 1.7) is not evolution friendly since after editing the template a new spreadsheet is created and the user needs to populate it again.

Based on ViTSL templates, Engels and Erwig introduced *ClassSheets* [Engels and Erwig, 2005], a higher-level object-oriented model. This model has both visual and (formal) textual representations. It allows end users to express the business logic and the structure of spreadsheets. However, it still has the same drawbacks as the system ViTSL-based replacing the ViTSL editor by a ClassSheet editor.

Hermans et al. also use MDE concepts, not to create spreadsheets from models, but to derive models from spreadsheets, using the model to extract knowledge about the spreadsheets [Hermans et al., 2010]. This work is based on patterns commonly available in spreadsheets, which are used to identify classes and then extract class diagrams. The techniques developed were implemented in a tool called *Gyro*.

Cunha adopted some techniques from MDE, using relational models to perform spreadsheet improvements, namely refactoring, migration and generation of edit assistance [Cunha, 2011]. The author applied reverse engineering techniques to derive various models from legacy spreadsheets and used functional dependencies as building blocks for those relational models.

As a result from Cunha's work, the HaExcel framework was developed. From a survey with several end users, the author obtained an indication that the models used can bring benefits to spreadsheet engineering, helping user to commit less errors and to work faster. Nevertheless, it has a severe drawback: it is mostly tailored to database-like spreadsheets.

Later, Cunha et al. developed a technique to solve the problem of spreadsheet evolution using ClassSheets [Cunha et al., 2011b]. The authors used the 2LT framework [Cunha et al., 2006; Cunha and Visser, 2007] to define a coupled transformation system that operates on spreadsheet models and respective instances. This work was implemented in the HaExcel framework.

A commercial solution named *ModelSheet Authoring* was created by ModelSheet Software LLC, providing some kind of Model-Driven Spreadsheet Engineering (MDSE). *ModelSheet Authoring* is a web-based tool

to build and maintain model-based spreadsheets, delivering then Microsoft Excel spreadsheets conforming to the developed models. A drawback of this tool is that one has to upload the spreadsheet so it can be coevolved. ModelSheet Software LLC also provides customized spreadsheet templates and custom consulting services, backed by model-based spreadsheets.

Another commercial application is *Data Manager*, which was designed to manipulate spreadsheets providing a set of wizard-like dialogs to edit the data. This tool provides several methods to ensure that the data is inserted correctly, preventing end users to make mistakes restricting their input. However, this tool does not use any kind of MDSE and the spreadsheets are database like, i.e., they consist in data organized by records, where each record is stored in a row.

In spite of all this work to achieve MDSE, many issues are still unresolved. Unlike other programming languages, spreadsheets do not have an Integrated Development Environment (IDE)[10]. This kind of environment is widely used by professional programmers and they provide great features to improve programmers' performance. Creating an integrated environment for MDSE brings the benefits of MDE together with the ones provided by IDEs.

The work presented in this dissertation provides a similar integrated environment that was achieved by embedding ClassSheet models within spreadsheets themselves and providing operations to evolve both model and respective instances. This resulted in the contributions described in section 1.4.

1.3 Terminology

Throughout this dissertation, some terms are used which are explained in this section for clarity's sake. The goal is to dismiss ambiguous meanings that can arise since many of the terms are used every day, but without the accuracy needed in a scientific work.

spreadsheet — computer application, with an interactive interface, that provides sheets with tables. A spreadsheet is normally composed by a **spreadsheet host system** (e.g., Gnumeric, Microsoft Excel, OpenOffice Calc, etc.) and a **spreadsheet file**, where the data is stored along with the description of the embedded computations.

worksheet — sheet of a spreadsheet. A spreadsheet contains one or more worksheets. The words *worksheet* and *sheet* are both used interchangeably in this document to refer to a worksheet.

[10] Examples of IDEs: Eclipse (`http://www.eclipse.org/`) and Microsoft Visual Studio (`http://www.microsoft.com/visualstudio/en-us`).

range — set of one ore more cells.

cell — element of a table. A table contains several cells, displayed in a two-dimensional layout. Cells may contain values or formulas, and may reference other cells.

input cell — cell that should be filled by an end user, with the intent to be stored and/or to be used as input for a computation.

reference — link to another cell. It can be to a cell in the same worksheet, or a different worksheet in the same spreadsheet or another spreadsheet. In some contexts, to disambiguate terms, **cell reference** can be used to indicate a reference to a cell. Cell references are usually written like `A1`.

range reference — link to a range, like a cell reference is a link to another cell. They can be defined using the notation `A1:B2`.

value — content of a cell, i.e., a number, a date or time, or a string. In the case the value is in an input cell, it is dynamic (i.e., can be modified over time) and may be referred as **input value**; otherwise, it is static (i.e., it is not meant to be changed).

formula — expression that, when evaluated, returns a result that may depend on several inputs, which are usually set in other cells referred in the formula.

1.4 Main Contributions

This work builds upon the previous work on ClassSheets, focusing in providing a user-friendly environment for ClassSheet/spreadsheet coevolution, allowing end users to benefit from a model-driven environment.

To achieve the proposed goal, the following work was realized:

i. ClassSheets were embedded in spreadsheets, with the results presented at the 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2011) as a long paper and a research abstract at the Graduate Consortium:

   - **Embedding and Evolution of Spreadsheet Models in Spreadsheet Systems**, Jácome Cunha, Jorge Mendes, João Paulo Fernandes, and João Saraiva (VL/HCC 2011);

- **ClassSheet-driven Spreadsheet Environments**, Jorge Mendes (VL/HCC 2011 – Graduate Consortium);

and a prototype tool was also demonstrated.

ii. A set of transformations were defined forming a bidirectional spreadsheet evolution environment to allow a precise evolution of both spreadsheet models and respective instances, with the results presented at the 5th International Conference on Model Transformation (ICMT 2012) as a research paper, and at the 34th International Conference on Software Engineering (ICSE 2012) as a formal demonstration, a poster, a paper for the USER workshop and an extended abstract for the Student Research Competition:

   - **Bidirectional Transformation of Model-Driven Spreadsheets**, Jácome Cunha, João Paulo Fernandes, Jorge Mendes, Hugo Pacheco, and João Saraiva (ICMT 2012);
   - **A Bidirectional Model-Driven Spreadsheet Environment**, Jácome Cunha, João Paulo Fernandes, Jorge Mendes, and João Saraiva (ICSE 2012 – Poster);
   - **Coupled Evolution of Model-Driven Spreadsheets**, Jorge Mendes (ICSE 2012 – Student Research Competition).

iii. A prototype implementing the embedding of ClassSheet models and the bidirectional spreadsheet evolution environment was developed, resulting in an extension for OpenOffice/LibreOffice Calc, presented at ICSE 2012:

   - **MDSheet: A Framework for Model-Driven Spreadsheet Engineering**, Jácome Cunha, João Paulo Fernandes, Jorge Mendes, and João Saraiva (ICSE 2012);

iv. A comparison of the evolution of spreadsheets using the approach in this work with the evolution of plain spreadsheets was also made. A proposal for an empirical study with real-world users was presented in:

   - **Towards an Evaluation of Bidirectional Model-Driven Spreadsheets**, Jácome Cunha, João Paulo Fernandes, Jorge Mendes, and João Saraiva (ICSE 2012 – USER workshop);

1.5 Document Structure

This dissertation is organized as follows:

Chapter 2 contains the description of the embedding of ClassSheet models within spreadsheets and the effects that it has on spreadsheet data. It focuses contribution i.

Chapter 3 contains the list of operations that can be made on spreadsheet models and the ones that can be made on spreadsheet data. A relation between model operations and data ones is made, presenting the bidirectional spreadsheet evolution environment. This chapter focuses contribution ii.

Chapter 4 describes the prototype tool developed, dubbed MDSheet, which implements the techniques presented in chapters 2 and 3, focusing contributions iii and iv.

Chapter 5 concludes this dissertation with remarks on the work done and exposing directions for future work.

Chapter 2

Embedding Spreadsheet Models within Spreadsheets

Model-Driven Spreadsheet Engineering (MDSE) has so far been realized with the construction and use of stand alone tools to design an create spreadsheet models. This is also the case of the ViTSL editor [Abraham et al., 2005], a tool that has been described in chapter 1. This approach requires a significant effort to integrate the model creation with the spreadsheet development, since the spreadsheets that are generated by such tools are manipulated under a different environment, the traditional spreadsheet environment. Moreover, it has several other drawbacks:

- one has to create a new tool from scratch;
- having learned the modeling language, end users need to learn how to use the tool that enables model design;
- several steps are needed to generate a spreadsheet; and,
- when a model for which a spreadsheet has been derived needs to evolve over time, it is easy to transform it into one that does not respect the structure of the previously derived spreadsheet.

On the other end, the use of a tool specifically for the purpose of spreadsheet model design has some advantages:

- there is virtually no syntax limitation for the modeling language that one wants to use;

- there are no distractions from other features that are not directly related to modeling, as it happens with many generic purpose tools (e.g., a spreadsheet host system can provide statistical tools, which are not needed if one wants to create a calendar).

The use of a complex system with many computer programs with different purposes is not ideal to promote MDSE. In some cases, end users would have to intervene in operations not related to either the model evolution or the data edition (e.g., migrate the data after a some evolution step in the model).

A solution to overcome these difficulties allowing end users to focus only about the data (and possibly the model) is to include spreadsheet modeling inside the spreadsheet environment that end users are already used to. This approach eliminates the need to create a separate tool, the need for users to learn how to use a completely new tool, and the generation of a spreadsheet from the model and the coevolution of the data is facilitated since the model is in the same environment as the data.

To implement the proposed solution, one can:

- do major changes to the spreadsheet host system to include a model editor, which is almost like creating a new tool; or,
- include the model directly inside a sheet of the spreadsheet.

The second option is the one adopted in this work, since it permits to focus on the modeling language instead of the tool, but its implementation depends on the chosen modeling language.

The spreadsheet models that have received wider acceptance from the community have been ClassSheet models, which have a visual language that is very similar to spreadsheets themselves. The combination of these factors makes the ClassSheet language the ideal candidate to be embedded within spreadsheets. This language is described in the next section, presenting the motivation to use it as the modeling language in this thesis.

2.1 Spreadsheet Templates and ClassSheets

The spreadsheet templates defined in [Erwig et al., 2005; Abraham et al., 2005] are a formalism to define the structure and contents of a spreadsheet. Templates (t) are composed by blocks (b), which in turn can contain formulas (f). A template block can represent in its basic form a spreadsheet cell, or it can be a composition of other blocks, possibly forming columns (c). Moreover, template blocks (b or c) can be

expandable, i.e., their instances can be repeated either horizontally ($c^{\rightarrow}$) or vertically ($b^{\downarrow}$). When a block represents a cell, it contains a basic value (φ, e.g., a string or an integer), a reference (ρ), or an expression built by applying functions to a varying number of arguments given by a formula ($\varphi(f, \ldots, f)$).

Templates can be represented textually [Erwig et al., 2005] (see figure 2.1), or visually using the ViTSL language [Abraham et al., 2005].

$$
\begin{array}{lcll}
f \in Fml & ::= & \varphi \mid \rho \mid \varphi(f, \ldots, f) & (formulas) \\
b \in Block & ::= & f \mid b|b \mid b\hat{}b & (blocks, tables) \\
c \in Col & ::= & b \mid b^{\downarrow} \mid c\hat{}c & (columns) \\
t \in Template & ::= & c \mid c^{\rightarrow} \mid t|t & (templates)
\end{array}
$$

Figure 2.1: Syntax of the textual representation of spreadsheet templates [Abraham et al., 2005].

To generate a spreadsheet corresponding to a template, a spreadsheet of the size of the template is created, where the contents of the cells in the template indicate the default content and the type of that cell in the spreadsheet. If a block is in an expandable area (either horizontally, vertically, or both), then users can repeat it in the corresponding direction(s) (see figures 2.2 and 2.3). Moreover, one can impose expansion

	A
1	Values
2	0
⋮	
3	=SUM(A2)

(a) Template for a list of values.

	A
1	Values
2	0
3	0

(b) A spreadsheet generated from the template.

	A
1	Values
2	2
3	3
4	5

(c) A spreadsheet generated from the template, with some values.

Figure 2.2: Example of a template that expands vertically.

limits, identifying the number of columns or rows that should be repeated in each instance. This can be done removing the column or row separators as depicted in figure 2.4.

The construction of spreadsheet templates is restricted by a set of rules that are imposed by a type system. This type system has two distinguished sets of types: one for formulas and an other for templates. The former enforces the correct use of references within formulas, while the latter constraints the composition of template elements in addition to the ones imposed by the syntax of the language.

Simplistically, the formula typing ensures that range references are not used where a cell reference is expected. When using a reference in a template, the corresponding reference in the data can be either a

	A	B	···	C
1	Values	0		=SUM(B1)

(a) Template for a list of values.

	A	B	C
1	Values	0	0

(b) A spreadsheet generated from the template.

	A	B	C	D	E
1	Values	5	3	2	10

(c) A spreadsheet generated from the template, with some values.

Figure 2.3: Example of a template that expands horizontally.

	A	B	C	···	D
1	List	Item	0		

(a) Template for a list of values. Notice the missing column separator between columns B and C.

	A	B	C	D
1	List	Item	0	

(b) A spreadsheet generated from the template.

	A	B	C	D	E	F	G	H
1	List	Item	3	Item	6	Item	2	

(c) A spreadsheet generated from the template, with some values.

Figure 2.4: Example of a template that expands horizontally, with two columns being repeated.

Table 2.1: Types of references [Abraham et al., 2005].

Source cell	Target cell	Type of reference
non-repeating	non-repeating	cell
non-repeating	repeating	range
repeating	non-repeating	cell
repeating	repeating (same group)	cell
repeating	repeating (different group)	range

cell reference or a range one depending on the location of the source cell containing the formula with the reference and the target cell referenced in the formula (see table 2.1).

The template typing ensures that:

- when composing vertically two blocks, they have the same width;
- when composing horizontally two blocks, they have the same height;
- when composing vertically two columns, they have the same width; and,

- when composing horizontally two templates, they have the same height and the same block pattern (i.e., expandable blocks in one template match the position and size of the adjacent blocks in the other template).

With this information about spreadsheet templates, it is possible to create, for instance, a spreadsheet for budgeting (see figure 2.5), listing vertically the categories and horizontally the years. For each intersection

	A	B	C	D	E	⋯	F
1	Budget		Year				
2			year=2005				
3	Category	Name	Qnty	Cost	Total		Total
4			0	0	=(C3*D3)		=SUM(E3)
⋮							
5	Total				=SUM(E3)		=SUM(E5)

Figure 2.5: Template for budgeting purposes.

of the year columns with the category rows, there is the quantity, cost and sub-total for the corresponding category in that particular year. Moreover, sub-totals are also provided for each category (cell `F4`) and for each year (cell `E5`), and a grand total for all expenses is defined in cell `F5`. Also, there is no column separators between columns `C` and `D`, neither between columns `D` and `E`, which makes those three columns (`C`, `D` and `E`) to be repeated whenever one wants to add information about a new year. For the expansion used to add new categories, there is a separator between rows 3 and 4, implying that that expansion repeats only one row per category.

Spreadsheet templates provide a way to ensure that spreadsheets follow a specific layout, and allow to fill in cells more easily, having the user only to insert input values since the formulas are automatically inserted with the correct references. However, spreadsheet templates still use non-user-friendly references with the format that is usual in spreadsheets, and the contents of the template have no semantics related to the business model associated with them. To remove some of disadvantages of spreadsheet templates, a new kind of models was developed, namely ClassSheet models [Engels and Erwig, 2005].

ClassSheet models are build upon spreadsheet templates, and are a high-level, object-oriented formalism to specify the business logic of spreadsheets [Engels and Erwig, 2005]. They are very similar to spreadsheet templates, but with more information about the classes used in the model, and with named attributes that can be used to reference cells by name, removing the need to use the usual format of references (i.e., references like `A1` and `B3:D6`).

Some improvements were made in the textual language to describe ClassSheets (see figure 2.6), namely the use of named references ($n.a$, where n is the name of the class and a the name of the formula), the naming of formulas ($a = f$), and the definition of classes (c) with the use of labels (l) to name them.

$$
\begin{array}{lcll}
f \in Fml & ::= & \varphi \mid n.a \mid \varphi(f,\ldots,f) & (formulas) \\
b \in Block & ::= & \varphi \mid a = f \mid b|b \mid b\hat{}b & (blocks) \\
l \in Lab & ::= & h \mid v \mid .n & (class\ labels) \\
h \in Hor & ::= & \underline{n} \mid \mathbf{|}\underline{n} & (horizontal) \\
v \in Ver & ::= & \mathbf{|}n \mid \mathbf{|}\underline{n} & (vertical) \\
c \in Class & ::= & l : b \mid l : b^{\downarrow} \mid c\hat{}c & (classes) \\
s \in Sheet & ::= & c \mid c^{\rightarrow} \mid s|s & (sheets)
\end{array}
$$

Figure 2.6: Syntax of the textual representation of ClassSheets [Engels and Erwig, 2005], with the differences from the template language in red.[1]

The visual representation of ClassSheets also has some improvements relatively to the one for spreadsheet templates. The content of the cells can now have named references and named formulas, borders with different colors are used to specify the area of the classes, and bold-formatted labels identify the name of the classes.

These alterations are presented in the ClassSheet that models the budget system (figure 2.7) described above as a template in figure 2.5. This budget system (represented by class **Budget**) is composed by three

	A	B	C	D	E	...	F
1	**Budget**		**Year**				
2			year=2005				
3	**Category**	Name	Qnty	Cost	Total		Total
4		name="abc"	qnty=0	cost=0	total=qnty*cost		total=SUM(total)
⋮							
5	Total				total=SUM(total)		total=SUM(**Year**.total)

Figure 2.7: ClassSheet modeling a budget spreadsheet.

other classes: **Year**, **Category**, and a class that relates **Year** and **Category** (similar to a relationship in a relational schema) that will be called **Year_Category** for the remaining of this dissertation. The budget system groups its data by year (class **Year**) and by category (class **Category**).

Year can be expanded horizontally to include several years and contains an attribute `year` that has default value 2005. The default value is used when a new instance of the class is added so it is populated automatically with some data.

Category can be expanded vertically to set several items using the attribute `name`, that has default value the string abc.

Year_Category 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. The information contained in this class is the quantity (`qnty`) of a category, its cost (`cost`) and a sub-total (`total`).

[1] It is assumed that the colors are visible through the use of the electronic version of this dissertation.

The attribute `total` in **Year_Category** is used to calculate sub-totals for each category (cell `G4` of the embedded model) and also for each year (cell `E6`). This last sub-total is in turn used to calculate the grand total of the budget (cell `G6`).

2.2 Embedding ClassSheets within Spreadsheets

The ClassSheet language is a DSL to represent the business model of spreadsheet data. The visual representation of ClassSheets very much resemble spreadsheets (see previous section). As a consequence, it is only natural to use the latter for creating ClassSheet models. Thus, the well-known techniques to embed DSLs in a host general purpose language [Swierstra et al., 1999] are adopted so that the visual ClassSheet is embedded in a spreadsheets host system. In this way, both the model and the spreadsheet can be stored in the same file, and model creation along with data edition can be handled in the same environment that end users are familiar with.

The embedding of ClassSheets within spreadsheets is not direct, since ClassSheets were not meant to be embedded inside spreadsheets. Their resemblance helps, but some limitations arise due to syntactic restrictions imposed by spreadsheet host systems. Several options are available to overcome the syntactic restrictions:

- write a new spreadsheet host system from start;
- modify an existing spreadsheet host system; or,
- adapt the ClassSheet visual language.

The two first options are not viable to distribute MDSE widely, since both require end users to switch their system, which can be inconvenient. Also, the first option takes too much work and removes some focus on the modeling part.

The solution adopted modifies lightly the ClassSheet visual language so it can be embedded in a worksheet without doing major changes on a spreadsheet host system (see figure 2.8). The modifications are:

i. draw an expansion limitation line in the spreadsheet instead of it being in the column/row indices (letters and numbers respectively);

ii. identify expansion using cells (in the ClassSheet language, this identification is between columns/rows); and,

iii. fill classes with a background color instead of using lines (which are used for the expansion).

The last change (iii) is not mandatory, but it is easier to identify the classes and, along with the first change (i), eases the identification of the classes' parts. This way, users do not need to think which role the line is playing (expansion limitation or class identification).

	A	B	C	D	E	⋯	F
1	**Budget**		**Year**				
2			year=2005				
3	**Category**	Name	Qnty	Cost	Total		Total
4		name="abc"	qnty=0	cost=0	total=qnty*cost		total=SUM(total)
⋮							
5	Total				total=SUM(total)		total=SUM(**Year**.total)

(a) Original ClassSheet.

	A	B	C	D	E	F	G
1	Budget		Year			⋯	
2			year=2005			⋯	
3	Category	Name	Qnty	Cost	Total	⋯	Total
4		name="abc"	qnty=0	cost=0	total=qnty*cost	⋯	total=SUM(total)
5	⋮	⋮	⋮	⋮	⋮	⋯	⋮
6	Total				total=SUM(total)	⋯	total=SUM(Year.total)

(b) Embedded ClassSheet.

Figure 2.8: Comparison between a plain ClassSheet to model a budget spreadsheet and an equivalent embedded one.

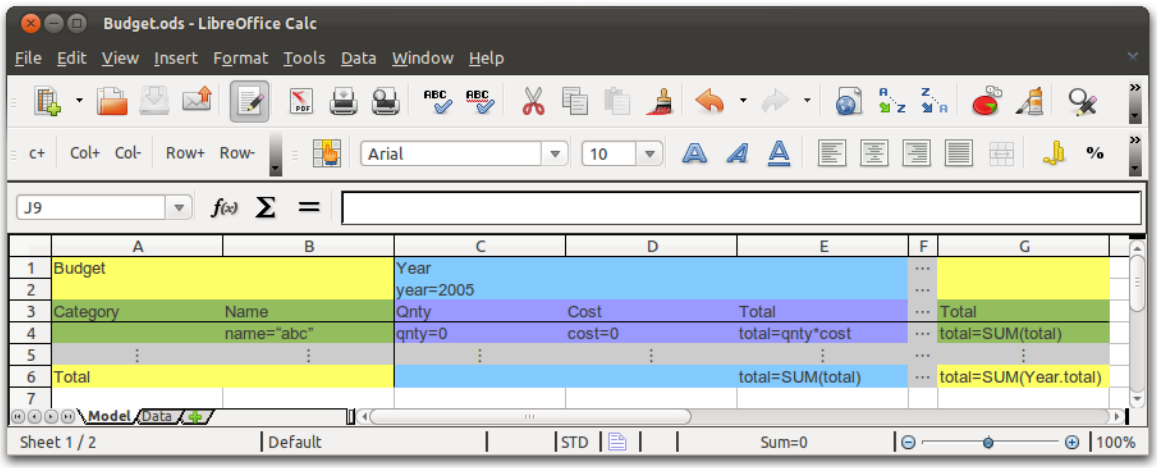
Using as example the budget ClassSheet from [Engels and Erwig, 2005] described previously and reproduced in figure 2.8a, one can see the differences between the original ClassSheet visual representation and the embedded one (figure 2.8b). The differences between the original ClassSheet and the embedded one are described next.

- In the original ClassSheet (figure 2.8a), there are two expansions denoted by the column between columns E and F for the horizontal expansion, and the row between rows 4 and 5 for the vertical one. Applying change ii to the original model, in the embedding (figure 2.8b) an extra column (F) and an extra row (5) are used to identify the expansions.

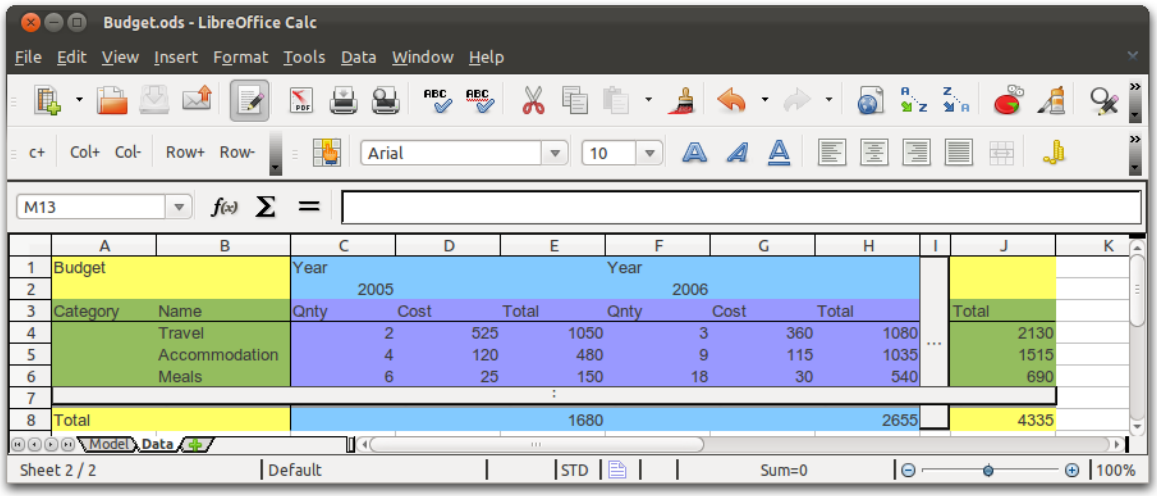
- To define the expansion limits in the original ClassSheet, there are no lines between the column headers of columns C, D and E which makes the horizontal expansion to use three columns and the vertical expansion only uses one row. This translates to a line between columns B and C and another line between rows 3 and 4 in the embedded ClassSheet as per change i.

- To identify the classes, background colors are used (change iii), so that the class **Budget** is identified by the yellow background, the class **Year** by the blue background, the class **Category** by the green background, and the class that relates the **Year** with the **Category** by the violet background.

To complete the embedding, the sheet containing the embedded ClassSheet should be alongside with the sheet containing its instance (i.e., the data), forming a spreadsheet with a sheet for the model and another one (or more) for the data (see figure 2.9). This way, users may evolve either model or data having the



(a) Model on the first sheet of the spreadsheet.



(b) Data on the second sheet of the spreadsheet.

Figure 2.9: Budget spreadsheet, with an embedded model and a conforming instance.

corresponding artifact automatically coevolved. Also, having the model near the data helps to document the latter, so that end users identify clearly the structure of the logic behind the spreadsheet.

To be noted that the data also is colored in the same manner as the model. This allows a correspondence between the data and the model to be made quickly, relating parts of the data to the respective parts in the model. This feature is not mandatory to implement the embedding, but can help the end users. One can

provide this coloring as an optional feature that could be activated on demand.

2.3 Instance Generation from Models

The envisioned MDSE environment is iterative, where when an operation is performed on the model, another operation is performed in the data (its instance) in order to coevolve the data accordingly with the model. This approach promotes an environment where the data is always conforming to the model.

ClassSheet models define a set of operations to evolve it, which can be used to create the models. Some basic operations available on ClassSheet models correspond to layout and cell content modifications: addition and deletion of columns and rows, and cell edition.

Moreover, since ClassSheets are an object-oriented formalism, one can also modify the classes within a model. A class is the description of an object which, within the context of this work, corresponds to a range of cells in the model instance. One specific operation is available to add a class to a model, but the deletion of a class is performed using basic spreadsheet operations, namely the deletion of either the columns or the rows corresponding to that class.

Like in the deletion operation, other more complex operations can be performed on models by using the operations previously described. A detailed description of the possible operations for model evolution is present in section 3.1:

To create, for example, the budget spreadsheet, one can:

1. add a class for the budget, selecting the range `A1:G6` and choosing the yellow color for its background;

2. add a class for the years, selecting the range `C1:F6`, choosing the blue color for its background, and setting the class to expand horizontally;

3. add a class for the categories, selecting the range `A3:G5`, choosing the green color for its background, and setting the class to expand vertically; and,

4. set the missing labels and formulas for the cells.

The addition of the relation class is not needed since it is added automatically when the environment detects superposing classes at the same level (**Year** is within **Budget**, as is **Category**, which leads to the automatic insertion of the relation class when the class **Category** is added after class **Year**).

After each step of the model creation, it is possible to visualize the result of the data coevolution, so the user creating the model has a precise idea of the effects resulting from is actions.

2.4 Summary

In this chapter, the advantages of using an integrated environment for MDSE are presented. The approach taken involves embedding ClassSheet models within spreadsheets, using a spreadsheet host system as the only tool that users need to develop spreadsheets. This work is also presented in [Cunha et al., 2011a; Mendes, 2011].

Only the embedding of ClassSheets is detailed in this chapter, which is not enough for spreadsheets that rely on a business logic that needs evolution over time. To address this concern, a set of model and data operations is presented in the next chapter.

Chapter 3

Model-Driven Spreadsheet Evolution

Creating a spreadsheet can take several steps, starting with an empty spreadsheet and modifying it until the wanted one is obtained. Moreover, spreadsheets, like many software artifacts, need to be updated over the time to meet the needs of the latest characteristics of businesses (e.g., billings and taxes).

Current spreadsheet host systems already provide several kinds of operations used to evolve spreadsheets: edit cell and add or remove columns or rows. Also, several other features help to create or modify spreadsheets: copying and pasting cell values, filters, cell sorting, etc., along with other more advanced features like the support for macros to perform repetitive tasks. However, these operations do not take into account the business logic behind the spreadsheet.

In order to evolve spreadsheets correctly regarding their business logic, a set of operations was created on ClassSheets, defining generic transformations over the models and the behavior of their modification on the data.

The usual process to evolve a system with some model and a corresponding instance is to evolve the model and then coevolve the instance accordingly in an unidirectional approach. However, it is possible to provide the ability to modify an instance in a way that it does not conform to the model anymore and then find a model to which the evolved instance conforms to. This can be done by inferring a new model or by modifying the original model.

Merging both model and instance evolution approaches we obtain a bidirectional coevolution environment where, when one artifact is evolved (model or instance), the respective artifact (instance or model) is automatically coevolved. This environment is especially directed to users which are responsible of the model.

It allows the user to edit the model having a clear idea of the changes performed on the data. Also, some operations are easier and faster performed using this kind of environment.

An example of an operation easier done in the data than in the model is the addition of a column in only one instance of the class **Year** of the ClassSheet introduced in chapter 2 (figure 2.8). This new column could store the information of a tax for that particular year, or be used to add some observation about a category. This operation can be realized either in the model or in the data:

In the model (figure 3.1): Replicate class **Year**, or add a new class with four columns, since the change is to be applied to one year only. Choosing the replication option, one has to replicate the class and then add one column to the new class. After having the new class with the wanted specification, the data of the year to be modified needs to be migrated manually to the new class.

In the data (figure 3.2): Select the column in the instance of the year to be modified and add a column.

From this overall instructions, it is clearly simpler to do the operation in the data rather than in the model. Also, some sorting problems may arise if the operation is done in the model, so more steps would be needed to complete the operation. The operation in the data creates automatically the necessary classes and there is no need to migrate any data.

	A	B	C	D	E	F	G
1	Budget		Year			⋯	
2			year=2005			⋯	
3	Category	Name	Qnty	Cost	Total	⋯	Total
4		name="abc"	qnty=0	cost=0	total=qnty*cost	⋯	total=SUM(total)
5	⋮	⋮	⋮	⋮	⋮	⋯	⋮
6	Total				total=SUM(total)	⋯	total=SUM(Year.total)

(a) Original budget spreadsheet model.

	A	B	C	D	E	F	G	H	I	J
1	Budget		Year			Year				
2			2005			2006				
3	Category	Name	Qnty	Cost	Total	Qnty	Cost	Total		Total
4		Travel	2	525	1050	3	360	1080	⋯	2130
5		Accommodation	4	120	480	9	115	1035		1515
6		Meals	6	25	150	18	30	540		690
7	⋮									
8	Total		1680			2655				4335

(b) Original budget spreadsheet data, conforming to model in figure a.

⇓ **Model evolution step:** replicate class **Year**

	A	B	C	D	E	F	G	H	I	J	K
1	Budget		Year			⋯	Year			⋯	
2			year=2005			⋯	year=2005			⋯	
3	Category	Name	Qnty	Cost	Total	⋯	Qnty	Cost	Total	⋯	Total
4		name="abc"	qnty=0	cost=0	total=qnty*cost	⋯	qnty=0	cost=0	total=qnty*cost	⋯	total=SUM(total)
5	⋮	⋮	⋮	⋮	⋮	⋯	⋮	⋮	⋮	⋯	⋮
6	Total				total=SUM(total)	⋯			total=SUM(total)	⋯	total=SUM(Year_1.total;Year_2.total)

(c) Evolved budget spreadsheet model.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Budget		Year				Year			Year				
2			2005				2006			2005				
3	Category	Name	Qnty	Cost	Total	Qnty	Cost	Total		Qnty	Cost	Total		Total
4		Travel	2	525	1050	3	360	1080	⋯	0	0	0	⋯	2130
5		Accommodation	4	120	480	9	115	1035		0	0	0		1515
6		Meals	6	25	150	18	30	540		0	0	0		690
7							⋮							
8	Total				1680			2655				0		4335

(d) Coevolved budget spreadsheet data, conforming to model in figure c.

⇓ **Model evolution step:** add column to the new class, before column `I`

	A	B	C	D	E	F	G	H	I	J	K	L
1	Budget		Year			⋯	Year				⋯	
2			year=2005			⋯	year=2005				⋯	
3	Category	Name	Qnty	Cost	Total	⋯	Qnty	Cost		Total	⋯	Total
4		name="abc"	qnty=0	cost=0	total=qnty*cost	⋯	qnty=0	cost=0		total=qnty*cost	⋯	total=SUM(total)
5	⋮	⋮	⋮	⋮	⋮	⋯	⋮	⋮	⋮	⋮	⋯	⋮
6	Total				total=SUM(total)	⋯				total=SUM(total)	⋯	total=SUM(Year_1.total;Year_2.total)

(e) Evolved budget spreadsheet model, with a new column at index `I`.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	Budget		Year			Year				Year					
2			2005			2006				2005					
3	Category	Name	Qnty	Cost	Total	Qnty	Cost	Total		Qnty	Cost		Total		Total
4		Travel	2	525	1050	3	360	1080	⋯	0	0		0	⋯	2130
5		Accommodation	4	120	480	9	115	1035		0	0		0		1515
6		Meals	6	25	150	18	30	540		0	0		0		690
7							⋮								
8	Total				1680			2655					0		4335

(f) Coevolved budget spreadsheet data, with a new column at index `L`, conforming to model in figure g.

⇓ **Data evolution step:** migrate **Year** instance for year 2006, copying that instance's data to the new class and removing the old instance.

	A	B	C	D	E	F	G	H	I	J	K	L
1	Budget		Year			⋯	Year				⋯	
2			year=2005			⋯	year=2005				⋯	
3	Category	Name	Qnty	Cost	Total	⋯	Qnty	Cost		Total	⋯	Total
4		name="abc"	qnty=0	cost=0	total=qnty*cost	⋯	qnty=0	cost=0		total=qnty*cost	⋯	total=SUM(total)
5	⋮	⋮	⋮	⋮	⋮	⋯	⋮	⋮	⋮	⋮	⋯	⋮
6	Total				total=SUM(total)	⋯				total=SUM(total)	⋯	total=SUM(Year_1.total;Year_2.total)

*(g) Budget spreadsheet model, where a new class similar to **Year** has an additional column.*

	A	B	C	D	E	F	G	H	I	J	K	L
1	Budget		Year				Year					
2			2005				2006					
3	Category	Name	Qnty	Cost	Total		Qnty	Cost		Total		Total
4		Travel	2	525	1050	⋯	3	360		1080	⋯	2130
5		Accommodation	4	120	480		9	115		1035		1515
6		Meals	6	25	150		18	30		540		690
7												
8	Total				1680					2655		4335

*(h) Budget spreadsheet data, where **Year** instance for 2006 has a new column, conforming to model in figure g.*

Figure 3.1: Adding a column to only one class instance, evolving the model and coevolving the data.

	A	B	C	D	E	F	G
1	Budget		Year			⋯	
2			year=2005			⋯	
3	Category	Name	Qnty	Cost	Total	⋯	Total
4		name="abc"	qnty=0	cost=0	total=qnty*cost	⋯	total=SUM(total)
5	⋮	⋮	⋮	⋮	⋮	⋯	⋮
6	Total				total=SUM(total)	⋯	total=SUM(Year.total)

(a) Original budget spreadsheet model.

	A	B	C	D	E	F	G	H	I	J
1	Budget		Year			Year				
2			2005			2006				
3	Category	Name	Qnty	Cost	Total	Qnty	Cost	Total		Total
4		Travel	2	525	1050	3	360	1080	⋯	2130
5		Accommodation	4	120	480	9	115	1035		1515
6		Meals	6	25	150	18	30	540		690
7	⋮									
8	Total				1680			2655		4335

(b) Original budget spreadsheet data, conforming to model in figure .

⇓ **Data evolution step:** add a new column before column `M`

	A	B	C	D	E	F	G	H	I	J	K	L
1	Budget		Year			⋯	Year				⋯	
2			year=2005			⋯	year=2005				⋯	
3	Category	Name	Qnty	Cost	Total	⋯	Qnty	Cost		Total	⋯	Total
4		name="abc"	qnty=0	cost=0	total=qnty*cost	⋯	qnty=0	cost=0		total=qnty*cost	⋯	total=SUM(total)
5	⋮	⋮	⋮	⋮	⋮	⋯	⋮	⋮	⋮	⋮	⋯	⋮
6	Total				total=SUM(total)	⋯				total=SUM(total)	⋯	total=SUM(Year_1.total;Year_2.total)

*(c) Coevolved budget spreadsheet model, with a new class similar to **Year** but with an additional column at index I.*

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	Budget		Year			Year				Year					
2			2005			2006				2005					
3	Category	Name	Qnty	Cost	Total	Qnty	Cost	Total		Qnty	Cost		Total		Total
4		Travel	2	525	1050	3	360	1080	⋯	0	0		0	⋯	2130
5		Accommodation	4	120	480	9	115	1035		0	0		0		1515
6		Meals	6	25	150	18	30	540		0	0		0		690
7	⋮														
8	Total				1680			2655					0		4335

*(d) Evolved budget spreadsheet data, where **Year** instance for 2006 has an additional column at index L.*

Figure 3.2: Adding a column to only one year, evolving the data and coevolving the model.

The model/data coevolution is accomplished defining a set of transformations on ClassSheets (Op_M, see section 3.1) and another set of transformations on the data (Op_D, see section 3.2). Then, both sets of transformations are related so that any transformation on either artifact has a sequence of one or more transformations on the other artifact (see section 3.3). The relation between the two sets is designed so that a valid transformation in one artifact corresponds to a valid sequence of transformations on the other artifact and that the instance conforms to the model (figure 3.3).

An operation is a sequence of one or more transformations that are made available for the users to evolve model and data. Operations for model and data are presented in the next section. Model operations are indexed with an M and data ones by a D.

The specification of the parameters of the operations are given by their types. The ones used are:

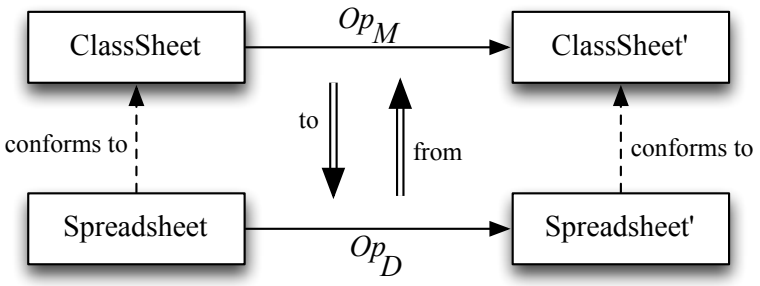


Figure 3.3: Bidirectional evolution diagram: operations on each artifact maintain conformity. The relation of model operations and data ones are represented by the functions to and $from$.

$Model$ — type of a model;
$Data$ — type of a model instance;
$Index$ — integer that represents a zero-based index of a column or row;
$Point$ — pair of integers used to identify a point using two indices, with $Point \stackrel{\text{def}}{=} (Index, Index)$;
$Range$ — pair of points that for a rectangular range, with $Range \stackrel{\text{def}}{=} (Point, Point)$;
$Where$ — position relative to an index, with the possible values being *Before* or *After*;
$Color$ — type of a color;
$String$ — type of a string;
$ClassName$ — type of a class name, which is basically a string but with values that follow a specific grammar, being $ClassName \stackrel{\text{def}}{=} String$.

Moreover, to simplify the type signatures of the operations,

$$Op_M :: Model \to Model$$

is used to represent an operation that has no parameters but the model, and

$$Op_D :: Data \to Data$$

is used to represent an operation that has no parameters but the data.

3.1 Model Operations

To evolve models, some basic operations are needed. A simple description of these operations follows, with their respective types and examples where the operation is applied to the budget spreadsheet (figure 2.8b),

unless specified otherwise.

$addColumn_M$ — Adds a column to the model. To perform this operation, one has to select a column in the model and choose to add the new column before or after that column.

$$addColumn_M :: Where \rightarrow Index \rightarrow Op_M$$

	A	B	C	D	E	F	G	H
1	Budget		Year				⋯	
2			year=2005				⋯	
3	Category	Name	Qnty	Cost		Total	⋯	Total
4		name="abc"	qnty=0	cost=0		total=qnty*cost	⋯	total=SUM(total)
5	⋮	⋮	⋮	⋮	⋮	⋮	⋯	⋮
6	Total					total=SUM(total)	⋯	total=SUM(Year.total)

Figure 3.4: $addColumn_M$ Before 4, resulting in the new column E.

$delColumn_M$ — Removes a column from the model. One just needs to select the column to remove to perform this operation.

$$delColumn_M :: Index \rightarrow Op_M$$

	A	B	C	D	E	F
1	Budget	Year			⋯	
2		year=2005			⋯	
3	Category	Qnty	Cost	Total	⋯	Total
4		qnty=0	cost=0	total=qnty*cost	⋯	total=SUM(total)
5	⋮	⋮	⋮	⋮	⋯	⋮
6	Total			total=SUM(total)	⋯	total=SUM(Year.total)

Figure 3.5: $delColumn_M$ 1, resulting in the deletion of column B from the original model.

$addRow_M$ — Adds a row to the model. To perform this operation, one has to select a row in the model and choose to add the new row before or after that row.

$$addRow_M :: Where \rightarrow Index \rightarrow Op_M$$

	A	B	C	D	E	F	G
1	Budget		Year			⋯	
2			year=2005			⋯	
3	Category	Name	Qnty	Cost	Total	⋯	Total
4		name="abc"	qnty=0	cost=0	total=qnty*cost	⋯	total=SUM(total)
5						⋯	
6	⋮	⋮	⋮	⋮	⋮	⋯	⋮
7	Total				total=SUM(total)	⋯	total=SUM(Year.total)

Figure 3.6: $addRow_M$ After 3, resulting in the new row 5.

$delRow_M$ — Removes a row from the model. One just needs to select the row to remove to perform this operation.

$$delRow_M :: Index \rightarrow Op_M$$

	A	B	C	D	E	F	G
1	Budget		Year			⋯	
2	Category	Name	Qnty	Cost	Total	⋯	Total
3		name="abc"	qnty=0	cost=0	total=qnty*cost	⋯	total=SUM(total)
4	⋮	⋮	⋮	⋮	⋮	⋯	⋮
5	Total				total=SUM(total)	⋯	total=SUM(Year.total)

Figure 3.7: $delRow_M\ 1$, resulting in the deletion of row 2 from the original model.

$setCell_M$ — Sets the content of a cell. To perform this operation, one has to select the cell to modify and give a new value (label or formula).

$$setCell_M :: Point \rightarrow String \rightarrow Op_M$$

	A	B	C	D	E	F	G
1	Budget		Year			⋯	
2			year=2005			⋯	
3	Category	Name	Qnty	Cost	Total	⋯	Total
4	id=0	name="abc"	qnty=0	cost=0	total=qnty*cost	⋯	total=SUM(total)
5	⋮	⋮	⋮	⋮	⋮	⋯	⋮
6	Total				total=SUM(total)	⋯	total=SUM(Year.total)

Figure 3.8: $setCell_M\ (0,3)$ "id=0", resulting in a new content for cell A4.

$addClass_M$, $addClassExp_M$ — Adds a class to the model. The class can either be static (i.e., not expandable; $addClass_M$) or expandable ($addClassExp_M$). To perform this operation, one has to select a range in the model, a name and a color for the new class. For the expandable form, one has also to select the direction of the expansion (horizontal or vertical).

$$addClass_M :: ClassName \rightarrow Color \rightarrow Range \rightarrow Op_M$$
$$addClassExp_M :: ClassName \rightarrow Color \rightarrow Range \rightarrow Direction \rightarrow Op_M$$

	A	B	C	D	E	F	G
1	Budget		Year			⋯	
2			year=2005			⋯	
3						⋯	
4						⋯	
5						⋯	
6						⋯	

(e) Before the operation.

	A	B	C	D	E	F	G
1	Budget		Year			⋯	
2			year=2005			⋯	
3						⋯	
4						⋯	
5	⋮	⋮	⋮	⋮	⋮	⋯	⋮
6						⋯	

(f) After the operation.

*Figure 3.9: $addClassExp_M$ "Category" Green $((0,2),(6,6))$, resulting in the addition of the class **Category** and the relation class **Year_Category**.*

$replicate^{\rightarrow}_M$, $replicate^{\downarrow}_M$ — Replicates a class, either horizontally ($replicate^{\rightarrow}_M$) or vertically ($replicate^{\downarrow}_M$). This operation is useful to create a new class with the same content of the one selected. The new class is created just after the class to be replicated.

$$replicate^{\rightarrow}_M :: ClassName \rightarrow Op_M$$
$$replicate^{\downarrow}_M :: ClassName \rightarrow Op_M$$

	A	B	C	D	E	F	G
1	Budget		Year			⋯	
2			year=2005			⋯	
3	Category	Name	Qnty	Cost	Total	⋯	Total
4		name="abc"	qnty=0	cost=0	total=qnty*cost	⋯	total=SUM(total)
5	⋮	⋮	⋮	⋮	⋮	⋯	⋮
6	Category	Name	Qnty	Cost	Total	⋯	Total
7		name="abc"	qnty=0	cost=0	total=qnty*cost	⋯	total=SUM(total)
8	⋮	⋮	⋮	⋮	⋮	⋯	⋮
9	Total				total=SUM(Category_1.total;Category_2.total)	⋯	total=SUM(Year.total)

Figure 3.10: $replicate^{\downarrow}_{M}$ *"Category", resulting in the duplication of class* ***Category*** *in* ***Category_1*** *and* ***Category_2****, with the formula in cell E9 updated accordingly.*

In section 3.3, more model operations are presented. However, those operations consist in a sequence of the operations described in this section.

3.2 Data Operations

Data operations are also available, consisting in operations already available for spreadsheet evolution (e.g., add a new column – $addColumn_D$), but other operations where created to reflect some model operations (e.g., add a new column to each instance of a class – $AddColumn_D$). For consistency, only operations that reflect model ones start with an uppercase letter. A simple description of these operations follows, with their respective types and examples where the operation is applied to the budget spreadsheet (figure 2.9b).

$addColumn_D$, $AddColumn_D$ — Adds a column to the data. To perform this operation, one has to select a column in the data and choose to add the new column before or after that column. The first operation ($addColumn_D$) adds only one column, unlike the second operation ($AddColumn_D$) which adds a column to each instance of the class that contains that column.

$$addColumn_D :: Where \rightarrow Index \rightarrow Op_D$$
$$AddColumn_D :: Where \rightarrow Index \rightarrow Op_D$$

	A	B	C	D	E	F	G	H	I	J	K	L
1	Budget		Year				Year					
2			2005				2006					
3	Category	Name	Qnty	Cost		Total	Qnty	Cost		Total		Total
4		Travel	2	525		1050	3	360		1080	⋯	2130
5		Accommodation	4	120		480	9	115		1035		1515
6		Meals	6	25		150	18	30		540		690
7	⋮											
8	Total					1680				2655		4335

Figure 3.11: $AddColumn_D$ *Before* 4*, resulting in the two news columns* `E` *and* `I`*.*

$delColumn_D$, $DelColumn_D$ — Removes a column from the data. One just needs to select the column to remove to perform this operation. The first operation ($delColumn_D$) removes only one column,

unlike the second operation ($DelColumn_D$) which removes a column from each instance of the class that contains that column.

$$delColumn_D :: Index \rightarrow Op_D$$
$$DelColumn_D :: Index \rightarrow Op_D$$

	A	B	C	D	E	F	G	H	I
1	Budget	Year			Year				
2		2005			2006				
3	Category	Qnty	Cost	Total	Qnty	Cost	Total		Total
4		2	525	1050	3	360	1080	...	2130
5		4	120	480	9	115	1035		1515
6		6	25	150	18	30	540		690
7				:					
8	Total			1680			2655		4335

Figure 3.12: $DelColumn_D$ 1, resulting in the data without column B from the original model.

$addRow_D$, $AddRow_D$ — Adds a row to the data. To perform this operation, one has to select a row in the data and choose to add the new row before or after that row. The first operation ($addRow_D$) adds only one row, unlike the second operation ($AddRow_D$) which adds a row to each instance of the class that contains that row.

$$addRow_D :: Where \rightarrow Index \rightarrow Op_D$$
$$AddRow_D :: Where \rightarrow Index \rightarrow Op_D$$

	A	B	C	D	E	F	G	H	I	J
1	Budget		Year			Year				
2			2005			2006				
3	Category	Name	Qnty	Cost	Total	Qnty	Cost	Total		Total
4		Travel	2	525	1050	3	360	1080		2130
5										
6		Accommodation	4	120	480	9	115	1035	...	1515
7										
8		Meals	6	25	150	18	30	540		690
9										
10					:					
11	Total				1680			2655		4335

Figure 3.13: $AddRow_D$ After 3, resulting in three new rows at lines 5, 7, and 9, since this is the upper case function to add a row.

$delRow_D$, $DelRow_D$ — Removes a row from the data. One just needs to select the row to remove to perform this operation. The first operation ($delRow_D$) removes only one row, unlike the second

operation ($DelRow_D$) which removes a row from each instance of the class that contains that row.

$$delRow_D :: Index \to Op_D$$
$$DelRow_D :: Index \to Op_D$$

	A	B	C	D	E	F	G	H	I	J
1	Budget		Year			Year				
2	Category	Name	Qnty	Cost	Total	Qnty	Cost	Total		Total
3		Travel	2	525	1050	3	360	1080		2130
4		Accommodation	4	120	480	9	115	1035	...	1515
5		Meals	6	25	150	18	30	540		690
6					⋮					
7	Total				1680			2655		4335

Figure 3.14: $delRow_D\ 1$ or $DelRow_D\ 1$, having both the same result, which is the data without row 2 from the original data.

$setCell_D$, $SetCell_D$ — Sets the content of a cell. To perform this operation, one has to select the cell to modify and give a new value (label or some input value). The first operation ($setCell_D$) sets the content of only one cell, unlike the second operation ($SetCell_D$) which sets the content of the cell from each instance of the class that contains that cell.

$$setCell_D :: Point \to String \to Op_D$$
$$SetCell_D :: Point \to String \to Op_D$$

	A	B	C	D	E	F	G	H	I	J
1	Budget		Year			Year				
2			2005			2006				
3	Category	Name	Qnty	Cost	Total	Qnty	Cost	Total		Total
4	0	Travel	2	525	1050	3	360	1080	...	2130
5	0	Accommodation	4	120	480	9	115	1035		1515
6	0	Meals	6	25	150	18	30	540		690
7					⋮					
8	Total				1680			2655		4335

Figure 3.15: $SetCell_D\ (0, 3)$ "$id = 0$", resulting in having the default value 0 inserted in cells A4, A5 and A6

$addInstance_D$ — Adds an instance of a class. It is used to add more instances of an expandable class, setting the cells of the new class with the default values defined in the model.

$$addInstance_D :: ClassName \to Model \to Op_D$$

	A	B	C	D	E	F	G	H	I	J
1	Budget		Year			Year				
2			2005			2006				
3	Category	Name	Qnty	Cost	Total	Qnty	Cost	Total		Total
4		Travel	2	525	1050	3	360	1080		2130
5		Accommodation	4	120	480	9	115	1035	···	1515
6		Meals	6	25	150	18	30	540		690
7		abc	0	0	0	0	0	0		0
8					:					
9	Total				1680			2655		4335

Figure 3.16: $addInstance_D$ "$Category$" m, where m corresponds to the original ClassSheet model, resulting in the new row 7 set with the default values defined in the model.

$replicate^{\rightarrow}_D$, $replicate^{\downarrow}_D$ — Replicates a class, either horizontally ($replicate^{\rightarrow}_D$) or vertically ($replicate^{\downarrow}_D$). It is different from $addInstance_D$ since this operation is equivalent to creating a new class in the model. If a class is expandable, then the current expand button remains unchanged and a new class instance with its own expand button is added.

$$replicate^{\rightarrow}_D :: Index \rightarrow Op_D$$
$$replicate^{\downarrow}_D :: Index \rightarrow Op_D$$

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Budget		Year			Year				Year				
2			2005			2006				2005				
3	Category	Name	Qnty	Cost	Total	Qnty	Cost	Total		Qnty	Cost	Total		Total
4		Travel	2	525	1050	3	360	1080		0	0	0		2130
5		Accommodation	4	120	480	9	115	1035	···	0	0	0	···	1515
6		Meals	6	25	150	18	30	540		0	0	0		690
7					:									
8	Total				1680			2655				0		4335

Figure 3.17: $replicate^{\rightarrow}_D$ 4, resulting in the coevolution corresponding to the operation $replicate^{\rightarrow}_M$ "$Year$"

3.3 Bidirectional Transformation Functions

To create the bidirectional evolution environment, model operations where related to data ones. Having a model and data conforming to that model, when a model operation is performed, the related data operation must be ran to coevolve the data so that it keeps conforming to the model.

Some operations may relate to a sequence of operations in the corresponding artifact. A sequence operations is defined by a sequence of one or more individual operations separated by a semicolon (;) as per the regular expression: $Op_0(;Op_i)^*$, where Op represents an operation. Also, some operations in an artifact may not have any effect on the related artifact. This is represented by the empty set ($\emptyset$).

Model operations have at most one corresponding individual data operation (see table 3.1). Moreover, the relation is an equivalence, where the data operation in the right-hand side corresponds to the model operation in the left-hand side, except for the empty ones.

Table 3.1: Model transformations and corresponding transformations on the respective instances.

Model Operation	Data Operation
$addColumn_M$	$AddColumn_D$
$delColumn_M$	$DelColumn_D$
$addRow_M$	$AddRow_D$
$delRow_M$	$DelRow_D$
$setCell_M$	$SetCell_D$
$replicate^{\rightarrow}_M$	$replicate^{\rightarrow}_D$
$replicate^{\downarrow}_M$	$replicate^{\downarrow}_D$
$addClass_M$	$\emptyset$
$addClassExp_M$	$\emptyset$

In turn, data operations relate to sequences of operations with more than one individual operation in the model, except for $addInstance_D$, which has no correspondence. Also, $setCell_D$ has no effect on the model when an input cell is edited. This relation is represented in table 3.2.

Table 3.2: Instance transformations and corresponding transformations on the respective model.

Data Operation	Model Operation
$addColumn_D$	$replicate^{\rightarrow}_M \ ; \ addColumn_M$
$delColumn_D$	$replicate^{\rightarrow}_M \ ; \ delColumn_M$
$addRow_D$	$replicate^{\downarrow}_M \ ; \ addRow_M$
$delRow_D$	$replicate^{\downarrow}_M \ ; \ delRow_M$
$setCell_D$	$(replicate^{\rightarrow}_M \ ; \ replicate^{\downarrow}_M \ ; \ setLabel_M)$ if the cell is empty or it contains a label; $\emptyset$ otherwise, i.e., it is an input cell.
$addInstance_D$	$\emptyset$

Note: When the operation is performed in a middle instance of an expandable class, each replication is performed twice. These operations correspond to non-expandable classes, or to the first or last instance of an expandable one.

3.4 Summary

In this chapter, the evolution of the embedded models was introduced, presenting a set of operations that users can perform on them.

Also, a set of data operations was described. Some of these operations are common spreadsheet operations while others has as goal to perform transformations on the data that correspond to model operations.

Moreover, a relation between model operations and data ones is presented. With this, a bidirectional evolution environment is defined. Having a valid model and some conforming data, when performing a valid evolution of the model and after the coevolution of the data, the latter is conform to the evolved model. Also, having a valid model and some conforming data, when performing a valid evolution of the data and after the coevolution of the model, the latter models the evolved data.

It is ensured by construction that the result of the coevolution is a valid artifact.

Chapter 4

Model-Driven Spreadsheet Engineering with MDSheet

The work described in this dissertation is implemented as an extension for OpenOffice/LibreOffice Calc, dubbed **MDSheet**. This implementation is a prototype to demonstrate the feasibility of the approach herein presented. Also, it permits to evaluate the approach.

Moreover, it allows to test new theories and encourages the application of new technologies to spreadsheet development, providing a base framework for spreadsheet analysis and transformation.

4.1 Architecture

The MDSheet framework was developed with modularity in mind. This modularity allows to adapt MDSheet to more spreadsheet host systems, but also to easily extend MDSheet with more features.

MDSheet has three main components (see figure 4.1):

- user interface (≈670 lines of OpenOffice Basic code);
- transformation system (≈2700 lines of Haskell code); and,
- integration code (≈4480 lines of C/C++ and Haskell code).

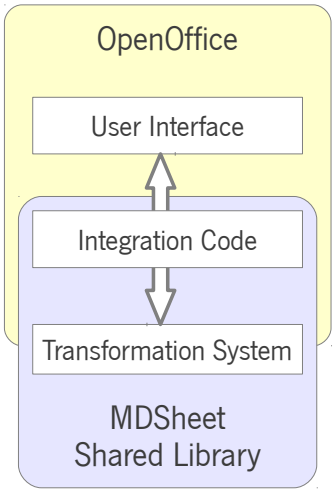


Figure 4.1: MDSheet architecture.

4.1.1 User Interface

The user interface consists of several controls arranged in a toolbar (see figure 4.2), a set of OpenOffice Basic macros and some dialogs (see figure 4.3). Pressing any control in the toolbar, an event is triggered that executes the macro defined for that event. Buttons are also present in the data to perform expansions. Moreover, an event listener is added to the model to monitor cell changes. This executes a macro that performs operation $setCell_M$ and the corresponding coevolution of the data.

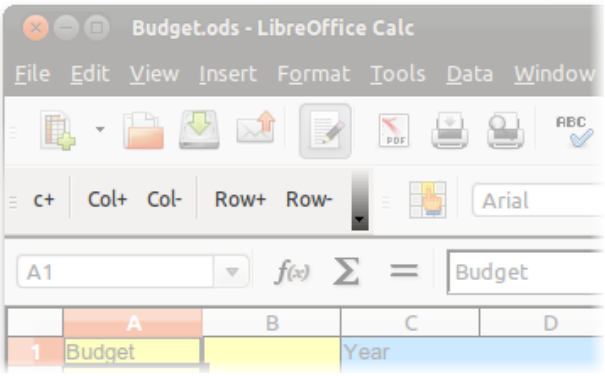
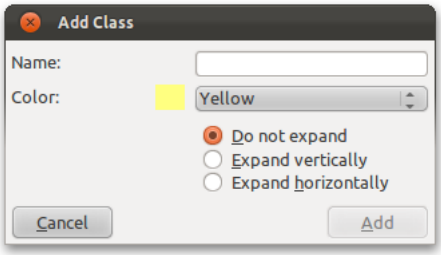
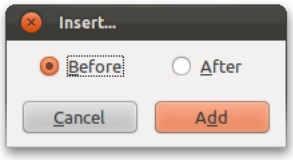


Figure 4.2: MDSheet toolbar in the LibreOffice Calc user interface.

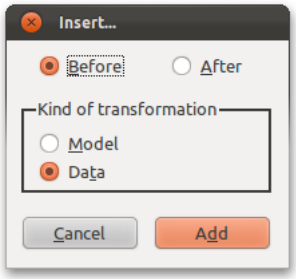
The control c+ in the toolbar is used to add a new class to the model. It uses a dialog to query the user about the name of the class, its color and if it expands horizontally, vertically or if it does not expand (figure 4.3a).



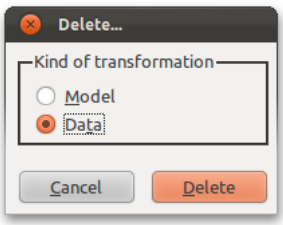
(a) Parameters to add a new class to the model.



(b) Parameters to add a column or a row to the model.



(c) Parameters to add a column or a row to the data.



(d) Parameters to delete a column or a row from the data.

Figure 4.3: MDSheet dialogs.

Controls `Col+`, `Col-`, `Row+` and `Row-` are used to add a new column, remove a column, add a new row

and remove a row, respectively. When in the model, the macros for `Col+` and `Row+` use a dialog to query the user if it is to add the new item (column or row) either before or after the selected one (see figure 4.3b). When in the data, it also queries about the kind of operation to execute, i.e., if it is only for that instance or for all instances of the class that contains the item (see figure 4.3c). The macros for `Col-` and `Row-` only use a dialog when in the data, to query about the kind of operation to perform: remove only one item (column or row) or remove the item in all instances of the class that contains it (see figure 4.3d).

4.1.2 Transformation System

The transformations system consists of a set of Haskell data structures and functions on those structures to manipulate ClassSheet models and spreadsheet data, including functions to perform the operations described in chapter 3.

The Haskell language is used because it is a high-level language, useful to specify programs, and calculations can be performed on the source code [Cunha and Pinto, 2005]. Calculations on the source code can be used to formally verify the correctness of the operations, or test properties of the framework like the ones described in [Cunha et al., 2012a].

Two main data structures are used: one for ClassSheet models and another one for spreadsheet data. These structures are similar to each other. They are composed by a list of classes and a matrix that contains the cells forming the model (for the model structure) or the data (for the spreadsheet data structure). In Haskell, these data structures are described as

```
data Model = Model [ModelClass] Grid
```

for the model, and

```
data Data = Data [DataClass] Grid
```

for the data, where the classes are represented by `ModelClass` and `DataClass`, and the matrix of cells is represented by `Grid`. The difference between model classes (`ModelClass`) and data classes (`DataClass`) is that the data class contains more information about the class, namely the number of instances.

The structure for the model is very different from the ClassSheet textual representation grammar defined in [Engels and Erwig, 2005] and reproduced in previously in figure 2.6, where the cells are described by blocks inside the specification of the classes. The structure chosen to implement ClassSheets is easier to manipulate than if it was similar to the ClassSheet grammar.

Despite many functions being defined over model, instance and auxiliary data structures, only the ones that provide the operations are needed to mentioned. All the operations over models and data have a respective function, but they are not invoked by themselves. Another set of functions is defined, where each function from that set invokes a model operation and a data operation, receiving a model/data pair and other arguments (e.g., an index when removing a column) and performing the necessary conversion between the arguments of the functions. The conversion is needed for some arguments, like indices, where the position of a cell in the data may not be specified by the model cell at the same position (e.g., cell `G6` from the spreadsheet in figure 2.9b is specified by cell `J8` in the respective model).

4.1.3 Integration Code

The integration code is used to connect the user interface to the transformation system. It is important to refer this code, since it is a great part of the MDSheet framework.

Unfortunately there is no Haskell bindings of the OpenOffice Application Programming Interface (API), which would deprecate most of the integration code. C/C++ is used to fill this gap, connecting to Haskell using its Foreign Function Interface (FFI) [Marlow, 2010, Chapter I.8].

Difficulties arise from doing this connection, like the conversion of the data structures, from the simple ones (e.g., integers of several sizes [8, 16, 32 or 64 bits]) to complex ones (e.g., OpenOffice Unicode strings to Haskell ones). Also, the Haskell strong type system and lazy evaluation need to be dealt with very carefully, or errors related to wrong types or to not completely evaluated values might appear. Haskell concurrency features are not used in this thesis, but they could be used to improve performance in future work to deal with very large models/spreadsheets.

4.2 Usage

In order to use the extension, users have available the toolbar (see figure 4.2) for model evolution, and also buttons in the data sheets to add instances of expandable classes (see figure 4.4).

In the MDSheet prototype, it is assumed that the first sheet contains the model and the second one the data.

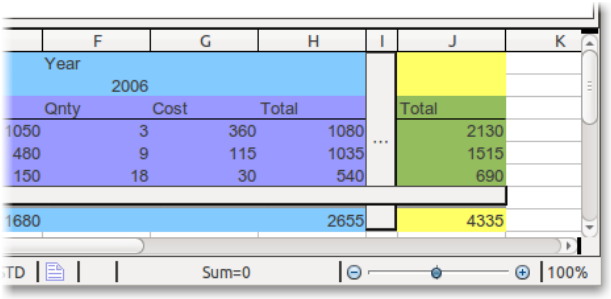


Figure 4.4: Spreadsheet data with focus on an expansion button (column `I`).

Starting with an empty spreadsheet, users can start creating their model. For that, an easy to use interface is provided: the user just needs to select the area that the main class will use and press control `c+` from the toolbar. After that, the *Add Class* dialog (figure 4.3a) is shown so that users can provide more information about the class. Then, the new class is added, and the top-left cell of that class is set with a label corresponding to the name provided in the dialog. At the same time, the data is coevolved and the user can see it going to the second sheet.

From this point on, users can add more classes inside the existing one, or perform other kinds of evolution. Also, labels and formulas can be added to the classes' cells.

4.3 Advantage in the Use of MDSheet

Creating a spreadsheet like the one displayed in figure 2.9, one can see the advantages from using a MDE approach. In that example, adding a new year to the spreadsheet is just as far as pressing a button. Without a model, it would require users to insert three new columns, setting the labels, adding the default values and inserting the formulas. With this many steps, users take longer, and may introduce errors while setting the contents of the cells or writing formulas. Copy/pasting can help the process, but errors can still arise (e.g., wrong references in the formulas).

Continuing with this example, having the data containing information about dozen years and about twenty category entries, to add a column to each year, the user can go to the model, select a column from **Year** and press `Col+` (or execute the corresponding operation from the data side). This indeed adds a column for each year, updating all the formulas to correct any cell reference. Without a model, the user needs to add manually a new column to each year (i.e., add ten times a new column at the right place), and then update the formulas. Some kind of formulas can be updated automatically and correctly by the spreadsheet host

system, but others not, depending on the kind of references used (direct cell reference like `A1;A2;B1;B2` or range reference like `A1:B2`). Once more, without a model, users need more steps to obtain the wanted result and errors can be introduced performing those steps.

The previous examples resulted in many changes on the data. But if the user only wants to add a column to only one instance of the class **Year**? Without a model, the user just adds that column and update the formulas if needed. But, with the bidirectional evolution environment, it is even simpler when having a model: the user just needs to select where to add the column, and formulas are updated automatically, as is the model!

Experienced users can use advanced features provided by spreadsheet host systems, like *named cells*, which can help prevent some errors, but it requires more steps in the user interface to use those features.

From a technical point of view, this approach has many benefits over plain spreadsheets, and can improve the performance of users while preventing errors. However, this method might not be intuitive enough for most of the end users.

4.4 Summary

In this chapter, a prototype implementation of the techniques described in chapters 2 and 3 is presented.

The overall architecture of the prototype is described, with details about its user interface. A basic usage of the prototype is shown and a example-based evaluation is made, presenting the advantages that the work resulting from this thesis brings.

Chapter 5

Conclusion

Researchers are being more and more aware of the use of spreadsheets and the amount of errors that they contain. Some work has already been done to overcome some spreadsheet deficiencies, and to prevent or correct errors, but none has obtained a great response from end users.

MDE can be advantageous in the context of spreadsheets, improving spreadsheet end-users' work performance [Cunha, 2011] and preventing errors, while providing documentation capabilities.

The work in this thesis tries to fill the gap between MDE methodologies and spreadsheet development providing an integrated environment for MDSE. The approach taken consists in the embedding of spreadsheet models, namely ClassSheets, within spreadsheets themselves in order to use many features that spreadsheet host systems provide. For end users, this approach provides them with an environment that they are already familiar with and, for the implementation of model support tools, it removes the necessity to create a complex full-featured software program, focusing only on adding features related to the model.

Bringing closer ClassSheets and spreadsheets, like the embedding of the former in spreadsheets themselves, originates new possibilities as the easy and fluid evolution environment detailed in this dissertation. Based on a restricted set of operations for model and respective instances, it is possible to provide a bidirectional evolution environment that is built so that model/data coevolution is correct by construction, always providing a pair of model and data, where the data conforms to the model.

The work resulting from this thesis is based on ClassSheets, which are used to model spreadsheets and which are already accepted by the scientific community. This latter point helped to divulge the work herein presented, and several publications resulted from it:

- **Embedding and Evolution of Spreadsheet Models in Spreadsheet Systems**, Jácome Cunha, Jorge Mendes, João Paulo Fernandes, and João Saraiva. In proceedings of the 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2011), Pittsburgh, PA, USA, pages 179–186, IEEE Computer Society, September 2011.

- **ClassSheet-driven Spreadsheet Environments**, Jorge Mendes. In proceedings of the 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2011) (Graduate Consortium), Pittsburgh, PA, USA, pages 235-236, September 2011.

- **Bidirectional Transformation of Model-Driven Spreadsheets**, Jácome Cunha, João Paulo Fernandes, Jorge Mendes, Hugo Pacheco, and João Saraiva. In proceedings of the 5th International Conference on Model Transformation (ICMT 2012), Prague, Czech Republic, May 2012.

- **Coupled Evolution of Model-Driven Spreadsheets**, Jorge Mendes. In proceedings of the 34th International Conference on Software Engineering (ICSE 2012) (Extended Abstract for the Student Research Competition), Zurich, Switzerland, pages 1616–1618, June 2012.

- **MDSheet: A Framework for Model-Driven Spreadsheet Engineering**, Jácome Cunha, João Paulo Fernandes, Jorge Mendes, and João Saraiva. In proceedings of the 34th International Conference on Software Engineering (ICSE 2012), Zurich, Switzerland, pages 1395–1398, June 2012.

- **A Bidirectional Model-Driven Spreadsheet Environment**, Jácome Cunha, João Paulo Fernandes, Jorge Mendes, and João Saraiva. In proceedings of the 34th International Conference on Software Engineering (ICSE 2012) (Poster), Zurich, Switzerland, pages 1443–1444, June 2012.

- **Towards an Evaluation of Bidirectional Model-Driven Spreadsheets**, Jácome Cunha, João Paulo Fernandes, Jorge Mendes, and João Saraiva. In proceedings of User evaluation for Software Engineering Researchers (USER 2012), an ICSE 2012 Workshop, Zurich, Switzerland, pages 25–28, June 2012.

- **Model-Driven Spreadsheets in a Multi-User Environment**, Jorge Mendes. In proceedings of the 2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2012) (Graduate Consortium), Innsbruck, Austria, September/October 2012. (to appear)

- **Extension and Implementation of ClassSheet Models**, Jácome Cunha, João Paulo Fernandes, Jorge Mendes, and João Saraiva. In proceedings of the 2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2012), Innsbruck, Austria, September/October 2012. (to appear)

Apart from the publications, a prototype named MDSheet was also developed, and that can be used to put in practice MDSE. It is also useful to compare the proposed spreadsheet development process with the current techniques used and, having an implemented tool, it is possible to receive feedback from end users and direct the development of model-driven spreadsheets to accommodate features and techniques that are more beneficial for them.

For that, it is planed, as future work, to assess the usefulness of the approach taken and the developed techniques with regard to common end users in an empirical study.

The work presented in this dissertation is only the beginning for a IDE for MDSE and more features are planned to be studied, namely:

- the interaction of a ClassSheet and several complying instances, with the result from the effects derived from their evolution;
- the control of spreadsheet versions, which is derived from the previous item;
- the concurrent use of the same spreadsheet in an online environment (e.g., Google Drive) or with synchronization in an offline setting, using techniques that can result from the previous item; and
- the notion of *views* derived from databases, that can be used to implement privacy control features.

Moreover, more work needs to be done to integrate the transformation environment with OpenOffice, improving the user interface in order to provide a seamless integration of MDSheet controls with OpenOffice's interface.

The MDSheet prototype is available for download at the author's institutional page:

`http://www.di.uminho.pt/~jorgemendes`

More informations about spreadsheet development is available at the SSaaPP – Spreadsheets as a Programming Paradigm project web page:

`http://ssaapp.di.uminho.pt`

References

Abraham, R. and Erwig, M. (2008). Test-Driven Goal-Directed Debugging in Spreadsheets. In *Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC '08, pages 131–138. IEEE Computer Society. 7

Abraham, R., Erwig, M., Kollmansberger, S., and Seifert, E. (2005). Visual Specifications of Correct Spreadsheets. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC '05, pages 189–196. IEEE Computer Society. 7, 8, 15, 16, 17, 18

BPM. bpmToolbox. http://www.bestpracticemodelling.com/software/bpmtoolbox (last retrieved: 2012-09-01). 6

Bricklin, D. VisiCalc: Information from its creators, Dan Bricklin and Bob Frankston. http://www.bricklin.com/visicalc.htm (last retrieved: 2012-09-01). 3

Bridgeford, B. D. (2011). W. Baraboo to pay more for borrowed money than believed. *News Republic*. http://www.wiscnews.com/baraboonewsrepublic/news/local/article_7672b6c6-22d5-11e1-8398-001871e3ce6c.html (last retrieved: 2012-09-01). 6

Bruins, E. (1949). On Plimpton 322. Pythagorean Numbers in Babylonian Mathematics. *Koninklijke Nederlandse Akademie van Wetenschappen*, 52:629–632. 1

Cunha, A., Oliveira, J. N., and Visser, J. (2006). Type-Safe Two-Level Data Transformation. In Misra, J., Nipkow, T., and Sekerinski, E., editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 284–299. Springer-Verlag. 9

Cunha, A. and Pinto, J. S. (2005). Point-free Program Transformation. *Fundamenta Informaticae*, 66(4):315–352. IOS Press. 43

Cunha, A. and Visser, J. (2007). Strongly Typed Rewriting for Coupled Software Transformation. *Electronic Notes in Theoretical Computer Science*, 174(1):17–34. Elsevier Science. 9

Cunha, J. (2011). *Model-based Spreadsheet Engineering*. PhD thesis, University of Minho. 9, 47

Cunha, J., Fernandes, J. P., Mendes, J., Pacheco, H., and Saraiva, J. (2012a). Bidirectional Transformation of Model-Driven Spreadsheets. In Hu, Z. and de Lara, J., editors, *Theory and Practice of Model Transformations – ICMT 2012*, volume 7307 of *Lecture Notes in Computer Science*, pages 105–120. Springer-Verlag. 43

Cunha, J., Fernandes, J. P., Mendes, J., and Saraiva, J. (2011a). 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, pages 179–186. IEEE Computer Society. 25

Cunha, J., Fernandes, J. P., Ribeiro, H., and Saraiva, J. (2012b). Towards a Catalog of Spreadsheet Smells. In Murgante, B., Gervasi, O., Misra, S., Nedjah, N., Rocha, A., Taniar, D., and Apduhan, B., editors, *Computational Science and Its Applications – ICCSA 2012*, volume 7336 of *Lecture Notes in Computer Science*, pages 202–216. Springer-Verlag. 7

Cunha, J., Visser, J., Alves, T., and Saraiva, J. (2011b). Type-Safe Evolution of Spreadsheets. In Giannakopoulou, D. and Orejas, F., editors, *Fundamental Approaches to Software Engineering – FASE '11/ETAPS '11*, volume 6603 of *Lecture Notes in Computer Science*, pages 186–201. Springer-Verlag. 9

Daily Express Reporter (2011). Mouchel Profits Blow. http://www.express.co.uk/posts/view/276053/Mouchel-profits-blow (last retrieved: 2012-09-01). 6

Ditlea, S. (1987). Spreadsheets can be hazardous to your health. *Personal Computing*, 11(1):60–69. 6

Donila, M. (2011). Trustee's Office mistake to cost taxpayers $12,500. www.knoxnews.com/news/2011/dec/03/trustees-office-mistake-to-cost-taxpayers-12500/ (last retrieved: 2012-09-01). 6

Engels, G. and Erwig, M. (2005). Classsheets: Automatic Generation of Spreadsheet Applications from Object-Oriented Specifications. In *Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering*, ASE '05, pages 124–133. ACM. 7, 9, 19, 20, 22, 43

Erwig, M., Abraham, R., Cooperstein, I., and Kollmansberger, S. (2005). Automatic Generation and Maintenance of Correct Spreadsheets. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 136–145. ACM. 8, 16, 17

EuSpRiG. European spreadsheet risks interest group. http://www.eusprig.org/ (last retrieved: 2012-09-01). 6

EuSpRiG. Spreadsheet mistakes - news stories collated by the European Spreadsheet Risks Interest Group. http://www.eusprig.org/horror-stories.htm (last retrieved: 2012-09-01). 6

FAST (2010). The FAST Standard. http://www.fast-standard.org/the-standard/ (last retrieved: 2012-09-01). 6

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley. 7

Grossman, T. A. (2002). Spreadsheet Engineering: A Research Framework. In *European Spreadsheet Risks Interest Group 3rd Annual Symposium*, pages 21–34. 6

Grossman, T. A. and Ozluk, O. (2010). Spreadsheets Grow Up: Three Spreadsheet Engineering Methodologies for Large Financial Planning Models. *CoRR*, abs/1008.4174. 6

Hermans, F., Pinzger, M., and van Deursen, A. (2010). Automatically Extracting Class Diagrams from Spreadsheets. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, ECOOP '10, pages 52–75. Springer-Verlag. 7, 9

Hermans, F., Pinzger, M., and van Deursen, A. (2011). Supporting Professional Spreadsheet Users by Generating Leveled Dataflow Diagrams. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 451–460. ACM. 5

Hermans, F., Pinzger, M., and van Deursen, A. (2012). Detecting and Visualizing Inter-Worksheet Smells in Spreadsheets. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 441–451. IEEE Press. 7

Ireson-Paine, J. (1997). Model Master: an Object-Oriented Spreadsheet Front-End. In *Computer-Aided Learning using Technology in Economies and Business Education*, CALECO '97. 8

Marlow, S., editor (2010). *Haskell 2010 Language Report*. http://haskell.org/definition/haskell2010.pdf (last retrieved: 2012-09-01). 44

Mattessich, R. (1961). Budgeting Models and System Simulation. *The Accounting Review*, 36(3):382–397. 3

Mendes, J. (2011). Classsheet-driven Spreadsheet Environments. In *Proceedings of the 2011 IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC '11, pages 235–236. 25

Nardi, B. A. (1993). *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press. 5

Panko, R. R. and Ordway, N. (2008). Sarbanes-Oxley: What About all the Spreadsheets? *CoRR*, abs/0804.0797. 5

Pryor, L. (2008). When, why and how to test spreadsheets. *CoRR*, abs/0807.3187. 7

RevenueRecognition.com (2004). The Impact of Compliance on Finance Operations. *Financial Executive Benchmarking Survey: Compliance Edition*. http://www.softrax.com. 5

Robson, E. (2001). Neither Sherlock Holmes nor Babylon: A Reassessment of Plimpton 322. *Historia Mathematica*, 28(3):167–206. 1

Rothermel, G., Burnett, M., Li, L., Dupuis, C., and Sheretov, A. (2001). A Methodology for Testing Spreadsheets. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(1):110–147. 7

Schmidt, D. C. (2006). Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31. IEEE Computer Society. 7

Spreadsheet (2012). *The American Heritage® New Dictionary of Cultural Literacy*. Houghton Mifflin Company, third edition. 2

SSRB (2010). Best Practice Spreadsheet Modelling Standards (version 6.1). http://www.ssrb.org/best_practice_spreadsheet_modelling_standards.html (last retrieved: 2012-09-01). 6

Swierstra, S. D., Azero Alcocer, P. R., and Saraiva, J. (1999). Designing and Implementing Combinator Languages. In Swierstra, S. D., Oliveira, J. N., and Henriques, P. R., editors, *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 150–206. Springer-Verlag. 21

U.S. Department of Education (2003). Audit of the Colorado Student Loan Program's Establishment and Use of Federal and Operating Funds for the Federal Family Education Loan Program. Technical Report ED-OIG/A07-C0009. 6

U.S. Department of Health and Human Services (2003). Review of Medicare Bad Debts at Pitt County Memorial Hospital. Technical Report A-04-02-02016. 6