Jorge Cunha Mendes Design, Implementation and Evaluation of Model-Driven Spreadsheets UMinho | 2017

Universidade do Minho
Escola de Engenharia

Jorge Cunha Mendes

Design, Implementation and Evaluation of Model-Driven Spreadsheets

Programa de Doutoramento em Informática (MAP-i)
das Universidades do Minho, de Aveiro e do Porto

Universidade do Minho

universidade de aveiro

U.PORTO

outubro de 2017

Universidade do Minho
Escola de Engenharia

Jorge Cunha Mendes

Design, Implementation and Evaluation of Model-Driven Spreadsheets

Programa de Doutoramento em Informática (MAP-i) das Universidades do Minho, de Aveiro e do Porto

Universidade do Minho

universidade de aveiro

U.PORTO

Trabalho realizado sob a orientação do
Professor Doutor João Alexandre Saraiva
e do
Professor Doutor Jácome Cunha

outubro de 2017

DECLARAÇÃO

Nome: Jorge Cunha Mendes

Endereço electrónico: jorgemendes@di.uminho.pt

Número do Bilhete de Identidade: 13313274

Título da tese: Design, Implementation and Evaluation of Model-Driven Spreadsheets

Orientador(es):

Professor Doutor João Alexandre Saraiva

Professor Doutor Jácome Cunha   Ano de conclusão: 2017

Designação do Doutoramento: Programa Doutoral em Ciências da Computação MAP-i

Universidade do Minho, 06/10/2017

Assinatura: Jorge Cunha Mendes

STATEMENT OF INTEGRITY

I hereby declare having conducted my thesis with integrity. I confirm that I have not used plagiarism or any form of falsification of results in the process of the thesis elaboration.
I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, October 6, 2017

Full name: Jorge Cunha Mendes

Signature: [illegible]

Acknowledgements

This work was not possible without the help of people that taught me and directed by in the right direction. In the first place, I would like to thank my supervisors João Saraiva and Jácome Cunha for their support and guidance. Without them, this work would not have seen the light of day.

Furthermore, I would like to thank my colleagues Nuno Macedo for his help with the bounded model checking and Hugo Pacheco for his help with the bidirectional transformation properties. I would like to thank also Zhenjiang Hu, the supervisor of my internship at the National Institute of Informatics (Japan), and his team that welcomed me into their group to work on bidirectional transformations. Moreover, I thank Martin Erwig that I first met as I started to work in this area and with whom I had interesting conversations about the topic of my work. Gregor Engels and Stefan Sauer also agreed to participate in a bilateral FCT-DAAD project, allowing to expand our work, and I am grateful to them for the opportunity. And a big thank you to my co-authors, namely João Paulo Fernandes who gave me some precious help at the beginning, Rui Pereira, and others already mentioned above.

Parts of this work required evaluation with people and I am thankful for their cooperation. I also thank the people at Banco Alimentar de Braga, Primavera BSS, and Francisco Duarte and João Roque from Bosch Car Multimedia, Portugal, for the discussion about problems they have and for giving me case studies. This work also benefited from interactions with other researchers that I crossed path with, namely in conferences and summer schools, but also within my own institution.

Finally, this work is partially funded by the ERDF through the Programme COMPETE and by the Portuguese Government through FCT - Foundation for Science and Technology, projects SSaaPP (ref. FCOMP-01-0124-FEDER-01004) and FATBIT (ref. FCOMP-01-0124-FEDER-020532), scholarships ref. BI2-2012 PTDC/EIA-CCO/120838/2012 UMINHO, ref. BI3-2012 PTDC/EIA-CCO/108613/2008 UMINHO, ref. BI1-2013 PTDC/EIA-CCO/120838/2012 UMINHO, ref. AE2015-0025 (INESC TEC) and ref. SFRH/BD/112651/2015.

Abstract

Design, Implementation and Evaluation of Model-Driven Spreadsheets

Spreadsheets are omnipresent tools to solve problems of all sorts. Their usage is simple and flexible, which attracts many of their users. The tabular format comes naturally to present data in many situations. This format simplifies the exchange of data between systems, but also provides simple visualizations of this data using some of the many features spreadsheet software provide nowadays. Moreover, users have advanced features available that make spreadsheets a powerful programming environment with a low-entry barrier.

However, spreadsheet users are usually not professional developers and thus lack knowledge and development methodologies to allow them to create error-free spreadsheets with minimal effort. This is even more significant with the freedom spreadsheets provide, where error-prevention and detection features are optional and lack emphasis.

A solution to improve spreadsheet development, both preventing errors and improving development performance, is defended in this work. This solution consists in using common software development methodologies and techniques to spreadsheet development, namely using Model-Driven Engineering, a methodology to specify a solution through abstraction.

This work brings a new modelling language to specify spreadsheets at a high level, abstracting the concrete data. It introduces an additional artefact in spreadsheet development that is connected to the actual spreadsheet through a conformance relation. With these two artefacts, spreadsheet development is divided into the definition of the layout and logic, and the input of concrete data. The former part defines constraints on the latter, preventing errors by design. Moreover, taking advantage of model-driven techniques, the whole life cycle of the spreadsheet is kept safe.

In addition to the theoretical definition of the modelling language, the conformance relation and other parts involved in the development of spreadsheets, an implementation of this work is provided as an extension to LibreOffice Calc. This implementation demonstrates the feasibility of the approach and allows to evaluate the work.

In order to evaluate this work, empirical studies were performed. These have shown the benefits of this work are significant, reducing error rates and improving spreadsheet development by end users.

Resumo

Desenho, Implementação e Avaliação de Folhas de Cálculo Orientadas por Modelos

Folhas de cálculo são ferramentas omnipresentes para resolver problemas de todos os tipos. O seu uso é simples e flexível, atraindo muitos dos seus utilizadores. Em muitas situações, o formato tabular é natural para a apresentação de dados. Este formato simplifica a troca de dados entre sistemas, mas também fornece uma visualização simples dos dados usando algumas das muitas funcionalidades que os programas de folhas de cálculo fornecem hoje em dia. Mais, os utilizadores têm disponível funcionalidades avançadas que tornam as folhas de cálculo um ambiente de programação poderoso e acessível.

Porém, os utilizadores de folhas de cálculo não são geralmente programadores profissionais, pelo que carecem do conhecimento e das metodologias de desenvolvimento que lhes permitiriam a criação de folhas de cálculo sem erros e com um esforço mínimo. Isto é ainda mais relevante com a liberdade que as folhas de cálculo disponibilizam, onde as funcionalidades de prevenção e de detecção de erros são opcionais e têm pouca ênfase.

Uma solução para melhorar o desenvolvimento de folhas de cálculo, tanto prevenindo erros como melhorando o desempenho no seu desenvolvimento, é defendida neste trabalho. Esta solução consiste no uso de metodologias e técnicas de desenvolvimento de software comuns, nomeadamente usando *Model-Driven Engineering* (engenharia orientada por modelos), uma metodologia para a especificação de uma solução através de abstracção.

Este trabalho traz uma nova linguagem de modelação para a especificação de folhas de cálculo a um alto nível, abstraindo os dados concretos. Ele introduz um artefacto adicional no desenvolvimento de folhas de cálculo que está relacionado com a folha de cálculo através de uma relação de conformidade. Este novo artefacto define restrições na folha de cálculo, prevenindo erros por construção. Aliás, tirando partido de técnicas de *Model-Driven Engineering*, todo o ciclo de vida da folha de cálculo é mantido correcto.

Em adição à definição teórica da linguagem, da relação de conformidade e de outra partes envolvidas no desenvolvimento de folhas de cálculo, resulta também deste trabalho uma implementação fornecida como uma extensão para o LibreOffice Calc. Esta implementação demonstra a viabilidade deste trabalho e também permite que ele seja avaliado.

Para validar este trabalho ser avaliado, foram desenhados e realizados estudos empíricos. Estes mostram que os benefícios deste trabalho são significativos, prevenindo erros e melhorando o desenvolvimento de folhas de cálculo pelos seus utilizadores.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the years, many solutions were proposed to allow people solve their problems through computer programs. One such solution, which appeared first in the late 1970s, is the electronic spreadsheet. Spreadsheets allow data to be arranged in a tabular fashion in columns and rows, but also to perform computation with such data using formulas.

Spreadsheets can be used to solve problems in many domains, for one-time problems or to be continuously used giving support to long processes. Spreadsheets can be developed by people with almost no knowledge on spreadsheet development or by experienced software developers. The solutions can be quite small, like a simple sum of a list of values, or they can be quite large, connecting to different sources of data, performing complex computations on that data and then providing an amplitude of views on the obtained results. These results can be used to provide a simple overview of the data or they can be used to take critical decisions with great potential of risk.

From their features, spreadsheets provide a way to create possibly complex software programs and thus can be seen as a programming paradigm. The development environment is quite simple and flexible, providing sheets where users can store their data in cells. Cells can have numeric or textual data, formulas to perform computation, and can be referenced using the coordinates of their respective column and row. In addition to these basic features, more complex ones are also available, providing data analysis and visualization tools for quick prototyping and development of software solutions. When these features are not enough, it is possible to implement new ones and automate further using macros, i.e., formulas or instructions developed in a programming language that can be used like native formulas. Therefore, spreadsheets provide simple features making it easy for new users to start using them, but they also can accommodate the needs of more experienced users.

The wide range of possible spreadsheet solutions coupled with the ample difference between basic spreadsheet users and the most experienced ones make spreadsheets very error prone. As such, both the academics and industry have proposed solutions to make spreadsheet development safer, but work is still required in order to empower end users with tools that indeed provide safer spreadsheets.

The work of this thesis, presented in this dissertation, intends to provide safer spreadsheets bringing common software development methodologies, techniques and tools to spreadsheet developers. This work is motivated in the next section, followed by an overview of the related state of the art and a summary of the approach taken in this thesis.

1.1 Motivation

The spreadsheet is an easy to use and flexible tool that allows people to solve their problems without requiring extensive programming knowledge. It permits to create programs to solve simple problems but also complex ones. Therefore, spreadsheets are the tool of choice in many situations. As such, many decisions are taken based on results of calculations performed using spreadsheets.

Panko and Ordway cite several reports showing the extensive usage of spreadsheets (Panko & Ordway, 2008), e.g., in 2004:

- financial intelligence firm CODA reported that 95% of U.S.A. firms use spreadsheets for financial reporting;
- the International Data Corporation found that 85% of the 118 interviewed business leaders were using spreadsheets in financial reporting and forecasting;

- A.R.C. Morgan interviewed 376 European individuals, with 80% of the respondents saying their firm use spreadsheets for both managing the control environment and financial reporting.

The usage of spreadsheets still remains significant nowadays. In PwC's 2014 Global Treasury Survey, 89% of the 110 survey respondents, from companies across the world, use spreadsheets (PwC, 2014). This trend is also present in the 2017 edition of the survey (PwC, 2017), influencing the productivity of an organisation:

> Treasury forecasting is still a cumbersome, manual and spreadsheet-based process involving many people from across the organisation, resulting in monthly or quarterly, rather than weekly, updates.
>
> (PwC, 2017)

Being such pervasive, mistakes or errors in spreadsheets become common, with a potentially serious impact in critical situations. News have been reported in some of these situations, e.g.:

- in the London 2012 Olympics, tickets to four synchronized swimming sessions were oversold by 10,000 due to the incorrect entry of the number of remaining tickets, having the organisation to make up to the people that were affected (Kelso, 2012);
- AstraZeneca, Britain's second largest drug maker, released confidential information that was inadvertently embedded in a spreadsheet template made available to analysts (Reuters, 2012);
- an influential economics paper (Reinhart & Rogoff, 2010) had, amongst other issues, a coding error in a spreadsheet leading to results favourable to the paper author's position (Konczal, 2013).

These and other news, some of them collected by the European Spreadsheet Risks Interest Group (EuSpRiG) in their *horror stories* page (EuSpRiG), show the importance of having correct spreadsheets and the impact an incorrect one can have.

Looking for the aspects of spreadsheet development that can lead to incorrect spreadsheets, two main aspects arise: the spreadsheet developer and the tools used to develop. The spreadsheet developer is usually also the *end user*, i.e., the one that uses the developed spreadsheet. Such user is usually a non-professional developer lacking in programming skills. The spreadsheet development environment is usually easy to use and flexible, giving much freedom to the user. The combination of both of these aspects result in the development of potentially incorrect spreadsheets, enabled by the lack of knowledge of the developer to create correct logic and the absence of restrictions from the development environment to create correct spreadsheets.

Many solutions implemented using spreadsheets, e.g., for budgeting and accounting, already have dedicated software solutions or could be implemented as one by professional developers. However, to move to one of those solutions would require additional costs, and spreadsheet users are not usually keen to switch their work tools. Therefore, diverse solutions were proposed by the industry and the academic community to assist spreadsheet users. These solutions include principles, guidelines and standards to mitigate errors and improve spreadsheet quality (e.g.: FAST, 2014; ICAEW, 2014; SSRB, 2010); tools in the form of add-ins and extensions to the development environment to enforce some (informal) specification or automate auditing tasks (e.g.: Modano[1]); and tools to help visualize, understand and locate potential sources of errors (e.g.: PerfectXL[2]; J. Cunha, Fernandes, Mendes, Martins, & Saraiva, 2012).

Another kind of solutions proposed to improve spreadsheet development consists in using a common software development methodology, namely Model-Driven Engineering (MDE). MDE consists in the creation of an abstraction of the problem that is then used to specify the solution. This methodology enables the use of further techniques and tools to support the development of software. Researchers have applied this methodology to spreadsheets, thus showing its applicability, but they did not take full advantage of what the methodology allows. The work from this thesis intends to improve the usage of MDE in spreadsheet development,

[1]`https://www.modano.com/`

[2]`https://www.perfectxl.com/`

further enhancing the development of spreadsheets

1.2 Background

The approach presented in this dissertation relies on existing methodologies, techniques and tools already known to software developers. Some essential ones are explained in this section.

Model-Driven Engineering (MDE). It is a methodology in software development that uses abstraction through modelling to specify a piece of software (Schmidt, 2006). Thus, a description of the domain of the problem (the model) is provided instead of stating the steps to solve such problem (the code instructions).

At the simplest level, MDE translates to a pair of artefacts (a model and an instance of that model) that are related by a conformance relation (fig. 1.1). More complex settings exist, where the model is itself modelled resulting in a meta-model that specifies a model which specifies an instance.

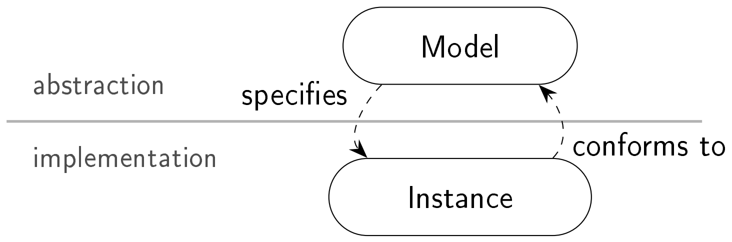


Figure 1.1: Overview of the model-driven engineering methodology.

In spreadsheets, the term *model* already exists: it is used to refer a spreadsheet solution that abstracts a concrete problem, but does not have any connection to model-driven software development.

In order to implement this methodology, other complementary concepts like *domain-specific modelling languages*, *model transformation* and *code generation* are used.

Domain-Specific (Modelling) Language (DSL/DSML). A DSL (van Deursen, Klint, & Visser, 2000; Voelter et al., 2013) is a language designed to be applied to a concrete domain, in contrast to a General-Purpose Language (GPL) that can be applied across several domains. It aims to provide domain-specific constructs to improve the expressivity of the language and reduce the complexity required when solving the same problem with a GPL.

Domain-specific languages for modelling can be used to formalize the application structure, behaviour, 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).

Model-instance synchronization. Using a model-driven approach can result in several artefacts for the same software object, but at distinct abstraction levels. When one of such artefacts is modified, or evolved, the others must be updated (co-evolved) to reflect the changes made and keep the conformance relation between instance and model.

Bidirectional transformations (BX) are a mechanism to maintain the consistency between related sources of information (Czarnecki et al., 2009; Diskin, 2008). They consist in a pair of transformations: one to update the instance after the model is modified, called the *to* transformation, and another to update the model after the instance is modified, called the *from* transformation (fig. 1.2). These transformations can work with the state of the artefacts (state-based BX), the differences between the original and the modified artefact (delta-based BX) and the operation used to modify the artefact (operation-based BX).

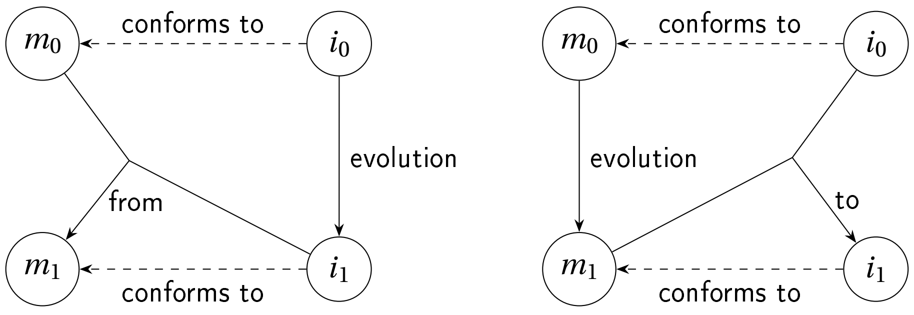


Figure 1.2: Synchronization of model (m_0) and instance (i_0) after an evolution step.

Object-Oriented Paradigm. (OOP) The Object-Oriented Paradigm (Meyer, 1997) is a programming paradigm where state and computation are grouped into objects. The data of the object is stored in attributes of the object, whilst computation is performed by invoking methods of the object.

Objects can be abstracted by their class. An object class, or just class, defines the attributes and methods of an object.

1.3 State of the Art

Industry and academics have both proposed solutions to improve spreadsheet development, be it in terms of preventing errors, finding errors, or following certain specifications. This section focuses on the state of the art of the specification of spreadsheets in order to prevent errors or to check their correctness against some concrete specification. An important aspect is that this covers specification of the spreadsheets themselves and not the data nor the computation that are represented in a spreadsheet, i.e., the work mentioned in this section is not an alternative to spreadsheet solutions.

Therefore, related work that uses models is presented, namely *Model Master*, *Excelsior*, *Gencel Templates* and *ClassSheets*, but first an informal approach to spreadsheet specification is provided.

Templates. Templates are not really spreadsheet models in the sense that they represent spreadsheets that originate from them, but rather a base to develop spreadsheets, where the initial layout and some labels and formulas may be present.

These templates are supported by some spreadsheet systems (e.g., LibreOffice Calc and Microsoft Office), and many templates are available on the Internet, e.g., at:

LibreOffice Templates: `http://templates.libreoffice.org/`

Microsoft Office Templates: `http://office.microsoft.com/templates/`

Model Master. Ireson-Paine introduced in 1997 *Model Master* (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., `A1`) is less error-prone than just editing the spreadsheet. Then, the compiler would generate a spreadsheet conforming to the given model.

For the compilation process, each attribute is mapped to a column in the spreadsheet. During this process, attribute names that are present within formulas are converted to cell references. This way, the models only cover a limited kind of spreadsheets, namely database-like tables (a header and a set of rows), but developers can improve the layout by specifying the worksheets and columns/rows where some or each attribute should be in.

Contrary to common models, the ones developed with Model Master already contain information about the values of the attributes, which can make of them a spreadsheet replacement for storage.

Model Master can provide some benefits:

Error reduction — Using named references, errors due to wrong references are reduced, and using a compiler to generate spreadsheets allows to check for errors before having the final spreadsheet.

Documentation — The textual format allows developers to insert comments with information about different parts of the spreadsheets, which contributes as some kind of documentation.

Modularity — Parts of the code can be reused, either by referencing different parts of the code (using object names), or by using object inheritance.

Portability — Models could be compiled to different kinds of spreadsheets, like LibreOffice Calc and Microsoft Excel, or even to another programming language such as Fortran and Pascal.

However, Model Master has also some limitations. After compiling the model to the spreadsheet, no kind of error prevention is made except for possible features present in the host spreadsheet system. Moreover, regarding the portability, the compiler itself may not be portable, due to being programmed in Prolog which makes it dependent on the targets supported by the Prolog compiler.

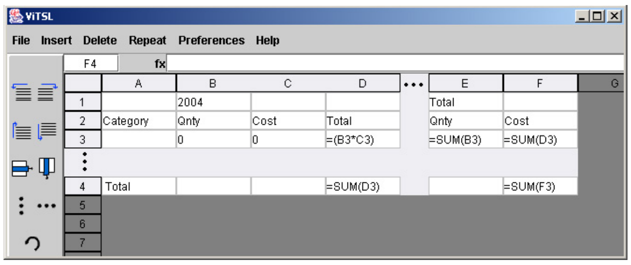
Later, Ireson-Paine improved on the original proposal adding a *decompilation* feature, allowing to convert spreadsheets to Model Master models (Ireson-Paine, 2001). Other features also present in this paper include multidimensional tables, and units and dimensional analysis when compiling.

Excelsior. Excelsior (Ireson-Paine, 2005) was developed by the same author of Model Master, both models sharing some characteristics. Excelsior, like Model Master, is a textual specification for spreadsheets, that is then compiled to a spreadsheet.

Unlike Model Master, the language of Excelsior has a different grammar with more features. A Excelsior specification is defined by a set of equations that can be related between themselves using primitives to modularize the specified spreadsheets. These primitives include ∪ (combination of sets of equations) and ⇒ (shifting of equations), amongst others.

The advantages and disadvantages of Excelsior are the same as Model Master's ones. In practice, Excelsior can generate Microsoft Excel and Google Sheets spreadsheets, and can be used to manipulate large spreadsheets in a programmatic way that provides more power than the usual features present in traditional spreadsheet systems.

Gencel Templates. In 2005, Abraham, Erwig, Kollmansberger, and Seifert presented ViTSL, a visual template specification language (Abraham et al., 2005; Erwig, Abraham, Cooperstein, & Kollmansberger, 2005). This language has both textual and visual representations, the latter being similar to spreadsheets themselves (see fig. 1.3).



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

Figure 1.3: Screenshot of the ViTSL editor, taken from Abraham et al. (2005).

The spreadsheet development process (fig. 1.4) using Gencel templates (or just templates) is similar to when using Excelsior, where the developer specifies the spreadsheet which is then compiled using Gencel to a spreadsheet that follows that specification. Unlike with Excelsior, spreadsheets generated with Gencel contain machinery to restrict some user operations to only those that are logically and technically correct for that template, preventing some kinds of errors when modifying the spreadsheet. 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. These are supported by an addon for Microsoft Excel.

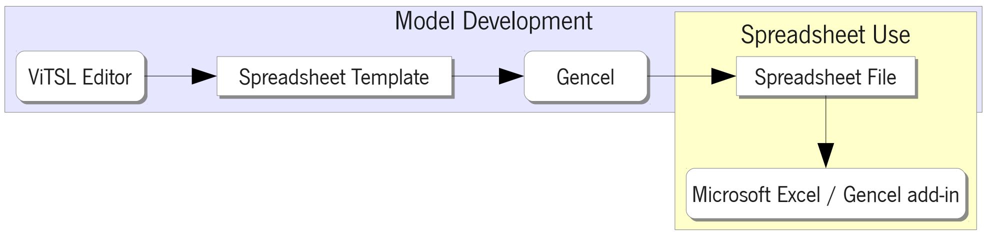


Figure 1.4: ViTSL-based environment for spreadsheet development.

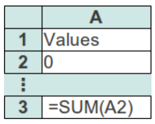
Gencel templates are a formalism to define the structure and contents of a spreadsheet. Templates (t) are composed by blocks (b), which in turn can contain formulas (f). A template block can represent in its basic form a spreadsheet cell, or it can be a composition of other blocks, possibly forming columns (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 fig. 1.5), or visually using the ViTSL language (Abraham et al., 2005).

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

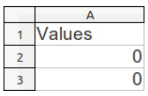
Figure 1.5: Syntax of the textual representation of spreadsheet templates (Abraham et al., 2005).

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 figs. 1.6 and 1.7). Moreover, one can impose expansion limits, identifying the



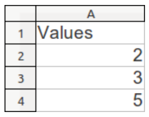
	A
1	Values
2	0
⋮	
3	=SUM(A2)

(a) Template for a list of values.



	A
1	Values
2	0
3	0

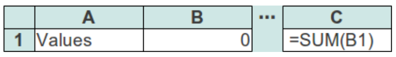
(b) A spreadsheet generated from the template.



	A
1	Values
2	2
3	3
4	5

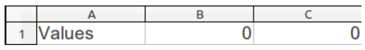
(c) A spreadsheet generated from the template, with some values.

Figure 1.6: Example of a template that expands vertically.



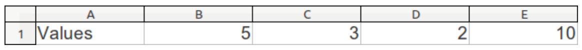
	A	B	⋯	C
1	Values	0		=SUM(B1)

(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

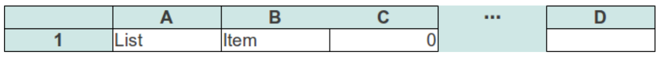
(c) A spreadsheet generated from the template, with some values.

Figure 1.7: Example of a template that expands horizontally.

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 fig. 1.8.

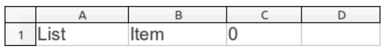
The construction of spreadsheet templates is restricted by a set of rules that are imposed by a type system. This type system has two distinguished sets of types: one for formulas and another for templates. The former enforces the correct use of references within formulas, while the latter constraints 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 cell reference or a range one depending on the location of the source



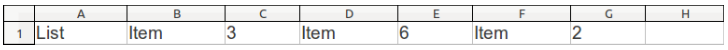
	A	B	C	···	D
1	List	Item	0		

(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 1.8: Example of a template that expands horizontally, with two columns being repeated.

cell containing the formula with the reference and the target cell referenced in the formula (see table 1.1).

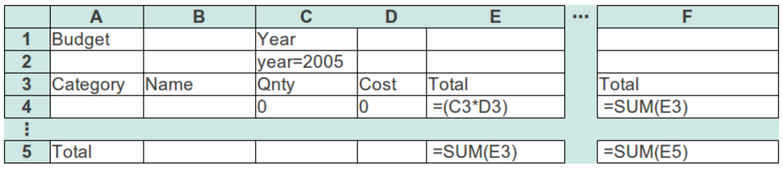
Table 1.1: Types of references in Gencel Templates (Abraham et al., 2005).

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

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 fig. 1.9), listing vertically the categories and horizontally the years. For each intersection of the year columns with the category rows, there is the quantity,



	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 1.9: Template for budgeting purposes.

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.

ClassSheets. ClassSheet models are built upon spreadsheet templates, and are a high-level, object-oriented formalism to specify the business logic of spreadsheets (Engels & 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 fig. 1.10), 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.

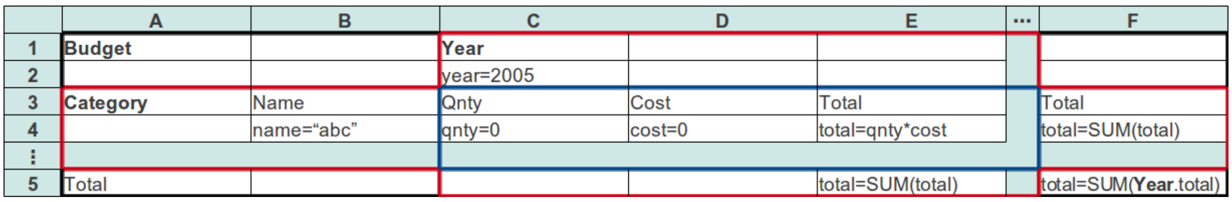
[3]It is assumed that the colours are visible through the use of the electronic version of this document.

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

Figure 1.10: Syntax of the textual representation of ClassSheets (Engels & Erwig, 2005), with the differences from the template language in red.[3]

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 colours are used to specify the area of the classes, and bold-formatted labels identify the name of the classes.

These modifications are presented in the ClassSheet that models the budget system (fig. 1.11) described above as a template in fig. 1.9. This budget system (represented by



	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 1.11: ClassSheet modelling a budget spreadsheet (Engels & Erwig, 2005).

class **Budget**) is composed by three 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 document. 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`).

The attribute `total` in **Year_Category** is used to calculate sub-totals for each category and also for each year. This last sub-total is in turn used to calculate the grand total of the budget.

Model Comparison. Each of these modelling languages have their own characteristics to help spreadsheet development. Thus, a simple high-level comparison using boolean attributes to denote the presence of some features was performed using the following items:

Textual — The language has a textual representation.

Visual — The language has a visual representation.

Comments — The language supports comments.

Object Oriented — The language is based on concepts from the object-oriented paradigm.

Named References — The model can make use of named references in formulas.

Compiled — The model is compiled to spreadsheets.

Model Enforcement — The model specification is enforced on the spreadsheet.

Using these attributes, it is possible to compare the previously mentioned models as displayed in table 1.2. For the templates (from section 1.3), the characteristics available are the ones already present in spreadsheets, and can vary from host to host.

One characteristic that most kind of models has is the possibility to use of named references within formulas (excepting Gencel Templates). Actually, this is a feature in some spreadsheet host systems that allows to name cells (or range of cells) and use that name to refer to it (or them). However, this is not frequently found in common spreadsheets, and this feature is not known by many spreadsheet users.

Although Model Master and Excelsior prevent some errors, they only are prevented through the compilation process of the specification to the spreadsheet without any verification afterwards. Using any of these tools, users have to decompile the spreadsheet to be sure that they

Table 1.2: Comparison of the different spreadsheet models.

	Textual	Visual	Comments	Object Oriented	Named References	Compiled	Model Enforcement
Templates		✓	✓		✓		
Model Master	✓		✓	✓	✓	✓	
Excelsior	✓		✓	✓	✓	✓	
Gencel Templates	✓	✓				✓	✓
ClassSheets	✓	✓		✓	✓	✓	✓

are correct. On the other hand, Gencel Templates and ClassSheets impose some restrictions on the data that are active during "runtime", continuously preventing the input of some kinds of errors.

1.4 Thesis

This work brings common software development methodologies, techniques and tools to spreadsheets. It has some focus on end-user development, without neglecting automated processing.

The approach taken in this work relies on MDE to specify spreadsheets and provides tools to develop spreadsheets. With this approach, the following is expected:

- correct spreadsheets by construction;
- safe evolution of specification and corresponding solutions;
- support for a large set of real-world spreadsheets;
- low knowledge overhead.

Correct by construction. Many kinds of errors can be present in spreadsheets. The specification language defined in this work prevents by itself a class of common errors in spreadsheets. Thus, when creating a new spreadsheet, it should be correct as per the specification.

Safe evolution. Using a bidirectional transformation engine, evolution of the specification is applied also to the spreadsheet, i.e., the spreadsheet is co-evolved, but also structural changes to the spreadsheet are applied to the spreadsheet's specification. Thus, the conformance relation is always kept intact.

Real-world spreadsheets. Many languages to model spreadsheets lack in expressiveness, resulting in a relatively small set of spreadsheets that can specified. The language presented in this work intends to be flexible enough to not restrict too much the spreadsheets that it allows to specify.

Low knowledge overhead. Few additional concepts are required. Most of the features that are specifiable have a direct translation to concrete spreadsheets, making most of the use of spreadsheet features already available in common spreadsheet systems.

1.5 Contributions

This work has a strong relation to previous work on model-driven spreadsheets, namely ClassSheets (Engels & Erwig, 2005), and it was developed alongside other projects close to the concepts used in this thesis. The distinction of the author's work to other's work is made whenever appropriate. Nevertheless, the main contributions are:

- A language to model spreadsheet tables.
- An embedding of the language within spreadsheets themselves.
- A technique to keep spreadsheets in conformance to their models.
- A prototype that implements a subset of the language's features.

- An evaluation of the model-driven approach to spreadsheet development.

Language. The language resulting from this work is named Tabula, it is described in chapter 3, and was also published in:

- “Tabula: A Language to Model Spreadsheet Tables”, Jorge Mendes and João Saraiva. In proceedings of the *4th Workshop on Software Engineering Methods in Spreadsheets* (SEMS), 2017.

Conformant spreadsheets. The conformance of spreadsheets is obtained using multiple concepts, including the embedding of models in spreadsheets themselves and the use of bidirectional transformations for safe evolution (chapter 4), which were also published in:

- “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.

- “MDSheet – Model-Driven Spreadsheets”, Jácome Cunha, João Paulo Fernandes, Jorge Mendes, Rui Pereira and João Saraiva. In proceedings of the *1st Workshop on Software Engineering Methods in Spreadsheets* (SEMS), 2014.

- “Embedding, Evolution, and Validation of Model-Driven Spreadsheets”, Jácome Cunha, João Paulo Fernandes, Jorge Mendes and João Saraiva. In *IEEE Transactions on Software Engineering*, vol. 41, no. 3, 2015.

Prototype. The prototype (chapter 5) is used to show the feasibility of the approach, to evaluate it and to perform additional experiments. All publications refer to this support tool, but the following one has a focus on it:

- “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.

Evaluation. Specific parts of this work were evaluated through empirical studies (chapter 6), with one of these studies being published in:

- "Towards an Evaluation of Bidirectional Model-Driven Spreadsheets", Jácome Cunha, João Paulo Fernandes, Jorge Mendes and João Saraiva. In proceedings of the *User evaluation for Software Engineering Researchers* (USER) workshop, 2012.
- "Embedding, Evolution, and Validation of Model-Driven Spreadsheets", Jácome Cunha, João Paulo Fernandes, Jorge Mendes and João Saraiva. In *IEEE Transactions on Software Engineering*, vol. 41, no. 3, 2015.

Further related work using model-driven spreadsheets developed during this thesis was published, including:

- "Complexity Metrics for Spreadsheet Models", Jácome Cunha, João Paulo Fernandes, Jorge Mendes and João Saraiva. In proceedings of the *13th International Conference on Computational Science and its Applications* (ICCSA), LNCS 7972, 2013.
- "Towards Systematic Spreadsheet Construction Processes" (abstract/poster), Jorge Mendes, Jácome Cunha, Francisco Duarte, Gregor Engels, João Saraiva and Stefan Sauer. In proceedings of the *39th International Conference on Software Engineering* (ICSE), 2017.
- "Systematic Spreadsheet Construction Processes", Jorge Mendes, Jácome Cunha, Francisco Duarte, Gregor Engels, João Saraiva and Stefan Sauer. In proceedings of the *2017 IEEE Symposium on Visual Languages and Human-Centric Computing* (VL/HCC), 2017.

Some more work on spreadsheet engineering, but not related to model-driven engineering, with the cooperation of other authors, was published in:

- "Towards the Design and Implementation of Aspect-Oriented Programming for Spreadsheets", Pedro Maia, Jorge Mendes, Jácome Cunha, Henrique Rebêlo and João Saraiva. In proceedings of the *2nd Workshop on Software Engineering Methods in Spreadsheets* (SEMS), 2015.

- "Towards an Automated Classification of Spreadsheets", Jorge Mendes, Kha N. Do and João Saraiva. In proceedings of the *3rd Workshop on Software Engineering Methods in Spreadsheets* (SEMS), 2016.

Moreover, this work enabled other researchers to build upon model-driven spreadsheet development resulting in the following publications with contributions from the author of this thesis:

- "QuerySheet: A Bidirectional Query Environment for Model-Driven Spreadsheets", Orlando Belo, Jácome Cunha, João Paulo Fernandes, Jorge Mendes, Rui Pereira and João Saraiva. In proceedings of the *2013 IEEE Symposium on Visual Languages and Human-Centric Computing* (VL/HCC), IEEE, 2013.

- "Querying Model-Driven Spreadsheets", Jácome Cunha, João Fernandes, Jorge Mendes, Rui Pereira, and João Saraiva. In proceedings of the *2013 IEEE Symposium on Visual Languages and Human-Centric Computing* (VL/HCC), IEEE, 2013.

- "Embedding Model-Driven Spreadsheet Queries in Spreadsheet Systems", Jácome Cunha, João Paulo Fernandes, Jorge Mendes, Rui Pereira and João Saraiva. In proceedings of the *2014 IEEE Symposium on Visual Languages and Human-Centric Computing* (VL/HCC), IEEE, 2014.

- "ES-SQL: Visually Querying Spreadsheets", Jácome Cunha, João Paulo Fernandes, Jorge Mendes, Rui Pereira and João Saraiva. In proceedings of the *2014 IEEE Symposium on Visual Languages and Human-Centric Computing* (VL/HCC), IEEE, 2014.

- "Design and Implementation of Queries for Model-Driven Spreadsheets", Jácome Cunha, João Paulo Fernandes, Jorge Mendes, Rui Pereira and João Saraiva, In *Central European Functional Programming – Summer School on Domain-Specific Languages* (CEFP/DSL 2013), Springer, 2014.

- "Evaluating Refactorings for Spreadsheet Models", Jácome Cunha, João Paulo Fernandes, Pedro Martins, Jorge Mendes, Rui Pereira, João Saraiva. In *Journal of Systems and Software*, vol. 118, 2016.

The author, while working on this thesis, also contributed to these additional publications:

- "SmellSheet Detective: A Tool for Detecting Bad Smells in Spreadsheets" (tool demo), Jácome Cunha, João Paulo Fernandes, Pedro Martins, Jorge Mendes and João Saraiva. In proceedings of the *2012 IEEE Symposium on Visual Languages and Human-Centric Computing* (VL/HCC), 2012.

- "Model Inference for Spreadsheets", Jácome Cunha, Martin Erwig, Jorge Mendes and João Saraiva. In *Automated Software Engineering*, Springer US, 2014.

- "Model-based Programming Environments for Spreadsheets", Jácome Cunha, Jorge Mendes, João Saraiva and Joost Visser. In *Science of Computer Programming* (SCP), Elsevier, 2014.

- "Spreadsheet Engineering" (invited tutorial), Jácome Cunha, João Paulo Fernandes, Jorge Mendes and João Saraiva. In *Central European Functional Programming – Summer School on Domain-Specific Languages* (CEFP/DSL 2013), Springer, 2014.

1.6 Document Structure

This dissertation describes the work accomplished while developing this thesis. It includes a summary of the relevant state of the art, some background on spreadsheets, a description of the proposed solution and an evaluation of the such solution. The contents in this dissertation are organised as follows:

Chapter 1 A description of what this thesis is about, a presentation of the concepts exposed in this dissertation and the results that where obtained.

Chapter 2 Definition of spreadsheets and their contents as used in this thesis.

Chapter 3 Definition of the Tabula model to abstract and specify spreadsheets.

Chapter 4 Tools to enable evolution of the specification of the spreadsheets to be propagated to the respective spreadsheets.

Chapter 5 The implementation of the theory developed in the previous chapters.

Chapter 6 Evaluation of the approach and results of using the implementation.

Chapter 7 Final thoughts, results, and possible future lines of work based on the contributions from this thesis.

Chapter 2

Spreadsheets

Spreadsheets are a versatile tool that can be decomposed into three main components:

- **language —** a two-dimensional language to store data and define computations;
- **program —** a concrete computer program implementation and execution, using the spreadsheet language; and as a
- **host —** a program that reads, writes and executes spreadsheet programs.

The spreadsheet language is the main point of focus of this work. A description of such language is presented below. This language allows to create spreadsheet programs that contain both logic and data, which are stored in spreadsheet files. The program is developed and presented to the user using a spreadsheet host system. Examples of spreadsheet hosts include *LibreOffice Calc*[1], *Microsoft Excel*[2], and *Google Sheets*[3].

Depending on the spreadsheet host and the file format used to store the spreadsheet, spreadsheet languages and programs can have distinct features not compatible between them. Therefore, the language described further on does not represent all kinds of spreadsheets, but it includes the foundational features shared by most of the formats. A specification to model this language, which makes the core of this work, is then provided in chapter 3.

[1]About LibreOffice Calc: `https://www.libreoffice.org/discover/calc/`
[2]About Microsoft Excel: `https://products.office.com/en/excel`
[3]About Google Sheets: `https://www.google.com/sheets/about/`

2.1 Foundational Features

Some spreadsheet features are transversal to most spreadsheet systems and spreadsheet file formats. The features described in the next subsections were gathered based on experience using the spreadsheet host systems mentioned above. The list of features is by no means meant to be exhaustive nor to formalize the core of the mentioned spreadsheet systems. Instead, it describes essential features that make spreadsheets a valuable tool.

2.1.1 Spreadsheet Structure

A *spreadsheet* or *workbook*, in its basic form, consists of a set of *worksheets* (or simply *sheets*) that are two-dimensional grids of *cells*. Cells can have a value, either text or a number, which can be directly set or that can be computed from a formula. Cells in a sheet are referenced by coordinates, indicating the column and row of the referenced cell and that can be in the form $(0,0)$, $A1$, or $R1C1$. In a top-down approach, the following definitions describing the spreadsheet language are obtained:

Definition 1. *A* spreadsheet *(or* workbook*) Sp is a pair* $Sp = (N, S^{\star})$ *where:*

- *N is the name of the spreadsheet;*
- $S^{\star}$ *is a list of worksheets.*

When stored in a file, the name of the spreadsheet is its file name.

Definition 2. *A* worksheet *(or* sheet *for short) S is a pair* $S = (N, G)$ *where:*

- *N is the name of the sheet;*
- $G = C^{\star\star}$ *is a two-dimensional grid of cells C.*

Definition 3. *A* grid $G = E^{\star\star}$ *is a two-dimensional rectangular structure with elements of E. A grid* $G_{m\times n}$ *has m columns and n rows with elements* $G_{(x,y)} = E$ *at column* $x \in [0..m-1]$ *and row* $y \in [0..n-1]$*:*

$$G_{m\times n} = \begin{array}{|c|c|c|c|}\hline G_{(0,0)} & G_{(1,0)} & \cdots & G_{(m-1,0)} \\ \hline G_{(0,1)} & G_{(1,1)} & \cdots & G_{(m-1,1)} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline G_{(0,n-1)} & G_{(1,n-1)} & \cdots & G_{(m-1,n-1)} \\ \hline \end{array}$$

Each cell in a sheet grid has a position that is given by its column and row. However, the usual notation used in spreadsheets is different from the one used in definition 3. Columns are identified by an equivalent bijective base-26 numeration system using the letters of the Latin alphabet. Therefore, column references range from `A` to `Z` continued with `AA`, `AB`, until `ZZ`, followed by `AAA`, `AAB`, and so on. Rows are identified by natural numbers, starting at `1`. Thus, the first (top-left) cell is at position `A1`, the one to its right is at position `B1`, and the one bellow the former is at position `A2`. More information on cell addressing is provided in section 2.1.2.

Definition 4. *A* cell *is a container that can store values and formulas to compute values. The contents of a cell are represented textually. Values in cells are of two types, with the contents of a cell being of the following kinds:*

- *Number — a value representable as a number;*
- *Text — any value that is not a number;*
- *Formula — a formula to compute a numerical or textual value.*

The type system of cell values is simple, with only two types:

Number. Any value that can be represented by a number, e.g., integer numbers, real numbers, currency amounts, and dates. Its value is represented using a real number allowing to display any of the other kinds of supported numerical values.

Text. Any value that cannot be represented by a number, and thus is represented as text, i.e., a sequence of characters. Moreover, numbers can also be represented as text, which can be specified when inserting a value into a cell.

Moreover, when a value cannot be evaluated (e.g., due to a division by zero) its type becomes of a special type *error* indicating the reason why the value could not be correctly computed.

The main function of cells is to store data and computations. The type of a cell is the type of the inserted value or the one computed by its formula. However, the type of the cell can also be influenced by the format applied to it, defining a specific view of the data and, in addition, its type. Moreover, the format applied to a cell can provide further information using a secondary notation (Petre, 1995).

The contents of a cell in a spreadsheet host is defined inserting a textual representation of the contents that is then interpreted and its type is inferred. Grammar 1 presents a simplification of the possible contents spreadsheet cells can have, covering all features addressed in this work.

Grammar 1 (Spreadsheet cell content)**.** The content of a cell should match the following grammar, with $\langle\text{cell}\rangle_s$ being the entry point:

$\langle\text{cell}\rangle_S$	$::=$	NUMBER	(numeric value)
	$\mid$	TEXT	(textual value)
	$\mid$	‘ = ’ $\langle\text{expr}\rangle_S$	(formula)
$\langle\text{expr}\rangle_S$	$::=$	FUN ‘(’ $\big(\langle\text{args}\rangle_S \mid \epsilon\big)$ ‘)’	(function call)
	$\mid$	(‘ + ’ $\mid$ ‘ − ’)$\langle\text{expr}\rangle_S$	(expression sign)
	$\mid$	$\langle\text{expr}\rangle_S$‘%’	(expression percentage)
	$\mid$	$\langle\text{expr}\rangle_S\langle\text{op}\rangle_S\langle\text{expr}\rangle_S$	(binary operation)
	$\mid$	$\langle\text{ref}\rangle_S$	(cell and range reference)
	$\mid$	$\langle\text{value}\rangle_S$	
$\langle\text{args}\rangle_S$	$::=$	$\langle\text{expr}\rangle_S$	(argument)
	$\mid$	$\langle\text{expr}\rangle_S$‘ , ’$\langle\text{args}\rangle_S$	(list of arguments)
$\langle\text{op}\rangle_S$	$::=$	‘ + ’ $\mid$ ‘ − ’ $\mid$ ‘ × ’ $\mid$ ‘ ÷ ’	(arithmetic operators)
	$\mid$	‘ < ’ $\mid$ ‘ ≤ ’ $\mid$ ‘ ≥ ’ $\mid$ ‘ > ’ $\mid$ ‘ = ’ $\mid$ ‘ ≠ ’	(logical operators)
	$\mid$	‘∧’	(exponentiation)
	$\mid$	‘&’	(text concatenation)
$\langle\text{ref}\rangle_S$	$::=$	REF	(cell reference)
	$\mid$	REF ‘:’ REF	(range)
	$\mid$	$\langle\text{ref}\rangle_S \cup \langle\text{ref}\rangle_S$	(range union)
	$\mid$	$\langle\text{ref}\rangle_S \cap \langle\text{ref}\rangle_S$	(range intersection)
$\langle\text{value}\rangle_S$	$::=$	NUMBER $\mid$ QUOTEDTEXT	

In the grammar above, FUN is a function name (e.g., `SUM`, `IF` and `VLOOKUP`), NUMBER is a number, TEXT is some arbitrary text, QUOTEDTEXT is some arbitrary text in between quotes and `REF` is a reference. These elements are provided by the lexer, whose concrete description is not given as it is not relevant for this work. Hence, the different kinds of references are abstracted into a single `REF` terminal, ignoring the different kinds of possible references (e.g., `A1`, `R1C1` and named references).

The basic features and layout of spreadsheets are demonstrated in example 1. In this example, two visual representations of a concrete spreadsheet are shown, both including the coordinates of the columns (on top) and of the rows (on the left). The first representation (fig. 2.1a) displays the definition of the spreadsheet program. The second one (fig. 2.1b) displays de execution of the program where the formula in cell `B5` is evaluated.

Example 1 (List of items)**.** A simple kind of spreadsheet that is commonly found contains a list of entries and, at the end, a summation formula. In this example, there is a label for the table in cell `A1`, a list of three items in rows 2 through 4 with a description of the item in column `A` and the respective value in column `B`, and finally a summation formula in cell `B5` with its respective label in cell `A5`.

	A	B
1	**Items**	
2	apple	5
3	banana	2
4	cherry	8
5	**Total**	=SUM(B2:B4)

(a) The table showing the definition of the program.

	A	B
1	**Items**	
2	apple	5
3	banana	2
4	cherry	8
5	**Total**	15

(b) The table showing the execution of the program.

Figure 2.1: Spreadsheet holding a list of items with their respective value and their total.

Since the type of the cell is inferred from the parsed textual content, there is no guarantee that the content is of the supposed type. When a user inserts a value, the value is not checked to ensure it is of the correct type the logic defined expects, leading to the first kind of spreadsheet error:

Spreadsheet Error 2.1. *Incorrect type — the inserted value is not of the correct type for the spreadsheet logic or for the conceptual model behind the spreadsheet.*

This error can contribute to incorrect values being computed, with unexpected results. Spreadsheet hosts are quite fault tolerant, with these incorrect computations being overlooked or even unnoticed.

Example 2. Such situation is demonstrated in table 2.1, where, for two inputs of varying types, the result of three formulas are presented. The different formulas have the same goal: to calculate the sum of two values. However, this is performed in three distinct forms: using the addition arithmetic operator, using the `SUM` function giving the inputs as two arguments, and using the `SUM` function giving the inputs as a range in a single argument. With at least one of the values provided as text, the result is not the expected one when using the `SUM` function, but it is correct when using the addition operator.

Table 2.1: Results for the same inputs using similar formulas but with different input types ($\mathbb{N}$ for number and $\mathbb{T}$ for text) as obtained in LibreOffice Calc, Microsoft Excel and Google Sheets.

`A1`	`B1`	Formula	Result
1 $\mathbb{N}$	2 $\mathbb{N}$	=A1+B1	3 $\mathbb{N}$
1 $\mathbb{N}$	2 $\mathbb{T}$	=A1+B1	3 $\mathbb{N}$
1 $\mathbb{T}$	2 $\mathbb{T}$	=A1+B1	3 $\mathbb{N}$
1 $\mathbb{N}$	2 $\mathbb{N}$	=SUM(A1,B1)	3 $\mathbb{N}$
1 $\mathbb{N}$	2 $\mathbb{T}$	=SUM(A1,B1)	1 $\mathbb{N}$
1 $\mathbb{T}$	2 $\mathbb{T}$	=SUM(A1,B1)	0 $\mathbb{N}$
1 $\mathbb{N}$	2 $\mathbb{N}$	=SUM(A1:B1)	3 $\mathbb{N}$
1 $\mathbb{N}$	2 $\mathbb{T}$	=SUM(A1:B1)	1 $\mathbb{N}$
1 $\mathbb{T}$	2 $\mathbb{T}$	=SUM(A1:B1)	0 $\mathbb{N}$

This situation is hard to notice in a real-world spreadsheet, but there are some possible indicators that something is wrong, e.g., the visual aspect of the input is different when it is as text (usually left-aligned if textual and right-aligned if numeric).

Microsoft Excel has an option to help detect erroneous situations related to cell contents, e.g., signalling when numbers are inserted as text. Moreover, when in a series of cells, it can flag cells with “strange” contents that differ from the other contents in the series. This kind of error can also be found using a smell analyser (e.g., J. Cunha, Fernandes, Mendes, Martins, & Saraiva, 2012). However, some of these techniques are statistical and do not work in every situation. A possible native solution for this problem resorts to the use of validation rules (see section 2.2).

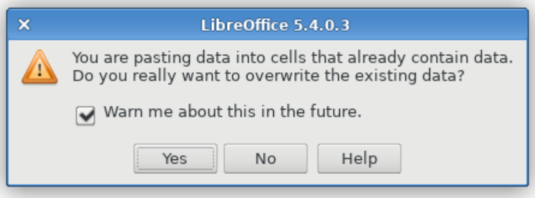
It is not only the flexibility of the cell contents that facilitate errors but also their positioning. Since there is no restriction to the contents of a cell, inserting a value in the cell next to the correct one can also lead to an invalid spreadsheet.

Spreadsheet Error 2.2. *Incorrect position — the inserted contents are not in the expected cell.*

Moreover, if the incorrect cell has already some content, it is possible to overwrite the contents of such cell inadvertently. This can be done by copying a range of cells and then pasting it in such a way that it overlaps with other non-empty cells, or by dragging a range with a similar outcome.

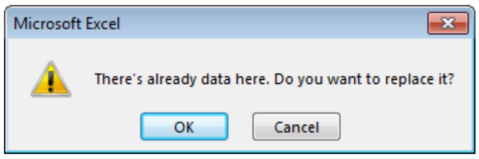
Spreadsheet Error 2.3. *Erroneous overwrite — the contents of a cell are overwritten incorrectly.*

Spreadsheet hosts have features to limit such errors, e.g., showing a warning dialogue when pasting or dragging values into cells already with some contents (see fig. 2.2). However, such warnings can be easily disabled in the settings of the spreadsheet host and thus not prompting the user in future similar situations. Another feature that can help prevent both previous errors (2.2 and 2.3) is the sheet and cell protection mechanisms, preventing users to change cell contents they are not supposed to and thus limiting the impact of such errors.


LibreOffice 5.4.0.3

You are pasting data into cells that already contain data.
Do you really want to overwrite the existing data?

(a) LibreOffice Calc warning when pasting data over non-empty cells.


Microsoft Excel

There's already data here. Do you want to replace it?

(b) Microsoft Excel warning when dragging data to non-empty cells.

Figure 2.2: Examples of warnings when overwriting existing data.

2.1.2 Cell References

The ability to reference cells is one of the features that make spreadsheets a valuable tool. It allows to separate data from computation and to decompose computations into several cells using a simple mechanism to use the contents of other cells.

Spreadsheet hosts provide distinct forms to reference cells, being the A1 and R1C1 notations the most common. In both formats, it is possible to have absolute and relative references. The difference between these are noticeable when building spreadsheets, where relative references are modified to accompany copy-and-paste or cell fill actions. In contrast, absolute references are never modified. Using the A1 notation, it is possible to indicate that a coordinate is absolute using a dollar sign (`$`):

- inserting the sign before the first coordinate (the column), e.g., `$A1`, makes the first coordinate absolute;
- inserting the sign before the second coordinate (the row), e.g., `A$1`, makes the second coordinate absolute; and,
- inserting before each coordinates, e.g., `$A$1`, makes both of them absolute.

The same result can be obtained with the R1C1 notation using a different syntax, but with the same concept for making the indices relative.

Example 3. The simplest form to demonstrate the use of a reference is by creating a formula that references another cell, thus showing the content of that cell, e.g., using `=A1` in `B1` to show the content of `A1` also in `B1`. To show the use of absolute or relative references, it is possible to duplicate a table of data by only inserting once the formula for the top-left cell and then copy-and-pasting it into other cells. This feature is depicted in table 2.2.

The correct usage of cell reference relativity allows spreadsheet developers to create a series of similar formulas without much effort. The developer has only to create the formula for the first element and then copy-and-paste or use the fill feature to insert the formula for

Table 2.2: Comparison of absolute references versus relative ones, duplicating a table of numeric values using references by defining the top-left formula and copying it to the other cells.

(a) Data for the comparison.

	A	B	C
1	1	2	3
2	4	5	6
3	7	8	9

(b) Relative reference. (`=A1`)

=A1	=B1	=C1
=A2	=B2	=C2
=A3	=B3	=C3

1	2	3
4	5	6
7	8	9

(c) Mixed reference: absolute column and relative row. (`=$A1`)

=$A1	=$A1	=$A1
=$A2	=$A2	=$A2
=$A3	=$A3	=$A3

1	1	1
4	4	4
7	7	7

(d) Mixed reference: relative column and absolute row. (`=A$1`)

=A$1	=B$1	=C$1
=A$1	=B$1	=C$1
=A$1	=B$1	=C$1

1	2	3
1	2	3
1	2	3

(e) Absolute reference. (`=$A$1`)

=$A$1	=$A$1	=$A$1
=$A$1	=$A$1	=$A$1
=$A$1	=$A$1	=$A$1

1	1	1
1	1	1
1	1	1

the other elements of the series. However, its incorrect usage might pass unnoticed providing improper results.

Spreadsheet Error 2.4. *Incorrect relativity — the cell reference is not made (partially) absolute when it should have been or it is made (partially) absolute when it should have been relative.*

In addition to references to single cells, cell references can be made to multiple cells. This is achieved using range operators:

- range operator (:)
- intersection operator (∩, `!` (exclamation point) in Calc, *space* in Excel)
- union operator (∪, `~` (tilde) in Calc, `,` (comma) in Excel)

Range references in formulas are updated when cells are added within the range. However, no update is performed when cells are added before or after the range, resulting in an incorrect spreadsheet.

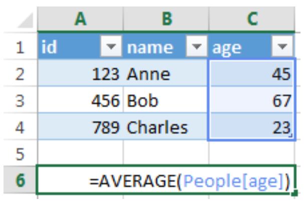
Spreadsheet Error 2.5. *Outdated range — the reference to a range does not reflect the current set of values.*

Similar issues may occur when using more advanced features. For instance, if a macro refers to a specific cell and this cell is moved to another coordinate, then the macro would be outdated and future uses of the macro might yield incorrect results as a consequence.

Spreadsheet systems usually provide tools to work with cell references. It is possible to use the mouse to select cells to be referenced when writing a formula, drag-and-drop to select a range of cells (: operator), and use the control key to select multiple ranges (∪ operator), to name the most common.

Spreadsheet Error 2.6. *Wrong selection — the cell (range) is not the correct one due to, e.g., an incorrect or involuntary click of the mouse.*

Spreadsheets allow other kinds of references. For instance, Microsoft Excel allows to define structured tables and then refer to cells in those tables using the name of the table and the name of the column (fig. 2.3). This is a similar feature common to other spreadsheet hosts, and also available in Microsoft Excel, called *named ranges*, that help prevent some issues related to range references. Named ranges are explained in the next section.



	A	B	C
1	id	name	age
2	123	Anne	45
3	456	Bob	67
4	789	Charles	23
5			
6	=AVERAGE(People[age])		

Figure 2.3: Example of a Microsoft Excel structured table called *People* and a formula referencing one of its columns.

2.1.3 Named Ranges

Named ranges are a feature that enables users to define a dictionary of names with an associated range in the spreadsheet. Then, when writing a formula, ranges can be referenced by their name.

They provide better readability in formulas and ensure the range reference in a series of formulas is the same. Moreover, when a range is modified, only the reference of the named range needs to be updated, whilst when using a concrete range reference in a series of formulas, the developer has to update the reference in all formulas. Without named ranges, the update can be performed using copy-and-paste or using automatic filling features, but is an extra step that needs to be performed and that can introduce errors.

On the other hand, named ranges add another level of indirection. Thus, when auditing a spreadsheet, the user has to check which named ranges are used and then verify the reference set in the named range. Furthermore, with this additional layer of indirection, it is more difficult to ensure the correctness of the reference when the original range is updated, e.g., when new cells are added at the beginning or at the end of the range, since in such cases the reference of the named range is not updated automatically by the spreadsheet host.

Spreadsheet experts often advise (e.g., in spreadsheet-related online forums) for the use of named ranges. However, researchers have found that they have a negative effect on novice users when debugging and developing spreadsheets (McKeever & McDaid, 2010, 2011; McKeever, McDaid, & Bishop, 2009).

2.2 Data Validation

The cells in a spreadsheet can have rules to validate the data they contain, provided by a feature called *data validity* (LibreOffice Calc) or *data validation* (Microsoft Excel and Google Sheets). This feature can help prevent the introduction of invalid values (e.g., by providing an error message) and also to verify existing values (e.g., adding rules to existing data and then asking for the invalid data).

Preventive usage. Setting data validation rules before inserting the data prevents the introduction of invalid data. When the user tries to insert an invalid value, the spreadsheet host warns the user of the error.

The behaviour of the spreadsheet host can vary when an invalid value is inserted. It can discard the introduced value, accept it or perform a programmed action. This behaviour is defined when creating the validation rule.

Corrective usage. Defining data validation rules to legacy spreadsheets can help find “a needle in a haystack”, i.e., it is possible to assign a rule to a range of cells without looking a their content and then select an option in the spreadsheet host system to find values that break the rules.

The validation rules can be applied to both numeric and textual values. Most of spreadsheet hosts support the following kind of rules:

- value is a number with format **decimal number** conditioned by a comparison;
- value is a number with format **whole number** conditioned by a comparison;
- value is a number with format **date** conditioned by a comparison;
- value is a number with format **time** conditioned by a comparison;
- value is textual with **text length** conditioned by a comparison;
- value is in a **list** or **cell range**, i.e., the value belongs to a predefined set of values;
- any value that is provided to a **formula expression** or **macro** which result evaluates to true.

Moreover, spreadsheet hosts provide an option to ignore blank cells, thus allowing the developer to select if an input cell can be left unset.

To help users introduce values, validation rules allow to define a help message composed by a title and some message. This help message is displayed to the user when the cell with the validation rule is selected (fig. 2.4).

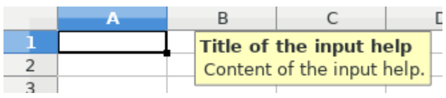
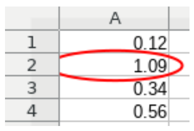

Title of the input help
Content of the input help.

Figure 2.4: Example of input help defined by a validity rule for cell `A1` in LibreOffice Calc.

Finally, the developer has to define if invalid values are accepted or not. In either case, a message with a custom title and description can be displayed when an invalid value is inserted. If invalid values are not accepted, then an error message is displayed. Otherwise, an warning or information message is displayed.

Adding data validation rules to existing data allows to find invalid values in legacy spreadsheets. The process is the same as when setting rules for new data. However, the user has then the option to mark invalid data which circles cells that do not follow their validation rules (fig. 2.5).



	A
1	0.12
2	1.09
3	0.34
4	0.56

Figure 2.5: Finding invalid values in LibreOffice Calc. The validity rule is set to allow decimal values between 0 and 1.

2.3 Other Elements

Spreadsheets support many other kinds of elements, including charts, pivot tables, and media objects. Some of these elements make use of data in spreadsheets (e.g., charts and pivot tables). As such, they also make use of cell references to refer to that data. However, these elements are not taken into account in this work.

Chapter 3

Tabula: Modeling Spreadsheet Tables

Tabula, the language resulting from the work performed in the context of this thesis to specify spreadsheet tables, emerges to overcome the limitations in other table modelling languages. It allows to implement to a greater extent the model-driven engineering methodology and to come closer to what spreadsheet users are familiar with.

The development of a spreadsheet using Tabula involves several artefacts, namely the Tabula model, an instance of such model and the concrete spreadsheet table (fig. 3.1). In this setting, the Tabula is an abstraction of the table, specifying its layout and logic, and the instance is an annotated representation of the table. However, the spreadsheet user only works with the Tabula and the table, the instance being an intermediate representation to support the model-driven ecosystem. These artefacts are explained in the next sections.

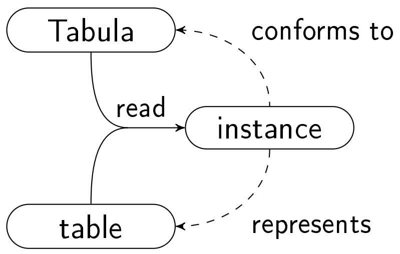


Figure 3.1: Overview of the artefacts in developing a spreadsheet table using Tabula.

3.1 Tabula

Tabula was first implemented as a technical solution to represent and work with ClassSheet models. The ClassSheet language is, from the related table modelling languages, the one that is at a higher conceptual level. It applies concepts from the object-oriented paradigm, with ties to UML, but it also comes close to the environment provided to spreadsheet users with its spreadsheet-like visual language. However, there was still a gap between ClassSheet models and spreadsheet developers.

In previous work, ClassSheets were embedded into spreadsheet themselves to provide an unified model-driven spreadsheet development environment directly within a spreadsheet host (J. Cunha, Fernandes, Mendes, Pacheco, & Saraiva, 2012; J. Cunha, Fernandes, Mendes, & Saraiva, 2011; Mendes, 2012). The implementation used a different abstract representation to work with ClassSheets, simplifying their manipulation and evolution within the development environment. Whilst evaluating the model-driven spreadsheet development using the embedded models, several limitations were found when modelling using the constraints imposed by the ClassSheet language. However, the abstract representation could provide support to a wider range of models if some of those constraints were not applied. Therefore, the abstract representation was paired with a new set of rules to provide a new language: Tabula.

Definition 5 (Tabula)**.** *A* Tabula *model (pl.* Tabulae*) T is a tuple $T = (N, C^\star, G)$ where:*

- *N is the name of the Tabula model;*
- *$C^\star$ is the set of classes C;*
- *G is the grid (definition 3) of model cells.*

The model consists in a grid specifying the contents of the cells in the table and in a list of classes to group the cells.

Definition 6 (Class)**.** *A class C is the abstract representation of an object in the table and is defined as a tuple $C = (N, R, E)$ where:*

- *N is the name;*
- *R is the range;*
- *E is the expansion information.*

Being defined by a range, a class is therefore rectangular, occupying a corresponding area in the grid of the Tabula. Moreover, a class can specify a singleton object in the table or a sequence of distinct objects with the same properties as defined by the kind of expansion of the class.

Definition 7 (Range)**.** *A range $R = (p_0, p_1)$ is a rectangular area in a grid starting with its top-left coordinate at point $p_0 \in P$ and ending with its bottom-right coordinate at point $p_1 \in P$, with P being the set of valid points in the grid.*

Definition 8 (Expansion)**.** *The expansion E of a class specifies if the class is a singleton or a sequence of zero or more elements, with the possible values:*

- *None — the class represents a singleton object;*
- *Down — the class represents a sequence of zero or more objects which are layed out in the table in a vertical fashion, i.e., one below the other;*
- *Right — the class represents a sequence of zero or more elements which are layed out in the table in a horizontal fashion, i.e., one to the right of the other;*
- *Both — the class represents a sequence of zero or more elements which are layed out in the table in a vertical and horizontal fashion.*

The layout is as presented by the grid, where each cell in the Tabula grid represents one or more cells in the spreadsheet. There are three kinds of Tabula cells: input, formula and label.

An *input* cell is specified with the contents in the format "name=default", where *name* is the name of the attribute it represents and *default* is the default value for the respective cells in the spreadsheet. This value is also used to define the type of the contents, where a numeric value implies the type being a number and a quoted text (possibly empty) implies the type being textual. Moreover, input cells can be constrained by a logical expression, a regular expression (regex) or by a set of values given from another attribute.

A *formula* cell is similar to an *input* cell. It is defined using the format “name=formula”, where *name* is the name of the attribute it represents and *formula* is the formula it defines. References in the formula are made using attribute names defined by *input* cells or by other *formula* cells.

A *label* cell is any cell that is neither of kind *input* nor *formula*. The content of the respective cells in the spreadsheet are exactly the same content as in the Tabula cell. Labels can be numeric, textual or even be empty.

Grammar 2 (Tabula cell content)**.** The content of a Tabula cell should match the following grammar, with $\langle\text{cell}\rangle_T$ being the entry point:

$\langle\text{cell}\rangle_T$	::=	$\langle\text{attribute}\rangle_T$	(attribute cell)
	$\mid$	$\langle\text{label}\rangle_T$	(label cell)
$\langle\text{attribute}\rangle_T$	::=	ATTRIBUTE ‘ = ’ $\langle\text{value}\rangle_S$	(input cell)
	$\mid$	ATTRIBUTE ‘ = ’ $\langle\text{value}\rangle_S$ ‘;’ $\langle\text{constraint}\rangle_T$	(constrained input cell)
	$\mid$	ATTRIBUTE ‘ = ’ $\langle\text{expr}\rangle_T$	(formula cell)
$\langle\text{constraint}\rangle_T$	::=	(‘ $<$ ’ $\mid$ ‘ $\leq$ ’ $\mid$ ‘ $\geq$ ’ $\mid$ ‘ $>$ ’ $\mid$ ‘ $=$ ’ $\mid$ ‘ $\neq$ ’) $\langle\text{value}\rangle_S$	(logical constraint)
	$\mid$	‘"’ REGEX ‘"’	(regex constraint)
	$\mid$	$\langle\text{ref}\rangle_T$	(attribute constraint)
$\langle\text{label}\rangle_T$	::=	NUMBER $\mid$ TEXT	(label value)
$\langle\text{expr}\rangle_T$	::=	FUN ‘(’ $\big(\langle\text{args}\rangle_T \mid \epsilon\big)$ ‘)’	(function call)
	$\mid$	(‘ + ’ $\mid$ ‘ − ’)$\langle\text{expr}\rangle_T$	(expression sign)
	$\mid$	$\langle\text{expr}\rangle_T$‘%’	(expression percentage)
	$\mid$	$\langle\text{expr}\rangle_T\langle\text{op}\rangle_S\langle\text{expr}\rangle_T$	(binary operation)
	$\mid$	$\langle\text{ref}\rangle_T$	(attribute reference)
	$\mid$	$\langle\text{value}\rangle_S$	
$\langle\text{args}\rangle_T$	::=	$\langle\text{expr}\rangle_T$	(argument)
	$\mid$	$\langle\text{expr}\rangle_T$‘ , ’$\langle\text{args}\rangle_T$	(list of arguments)
$\langle\text{ref}\rangle_T$	::=	ATTRIBUTE	(simple reference)
	$\mid$	CLASS ‘.’ ATTRIBUTE	(fully qualified reference)

This grammar is similar to the one for common spreadsheet cells, with differences in the top-level values and in the references. For the top-level values, there are productions for the three kinds of cells (input, formula and label). Input attributes can be optionally constrained, with three kinds of constraints. The regular expression is represented by the terminal REGEX, with the actual flavour (e.g., POSIX and PCRE) not being of impact for this work but just a technical detail. At the reference level, only references to class attributes are allowed. These are composed by an attribute name (represented by the terminal ATTRIBUTE, which is a sequence of alphanumeric characters starting with a letter) and by an optional class name (represented by the terminal CLASS, which is a sequence of alphanumeric characters starting with a letter similarly to attributes).

A Tabula is defined using the elements described above, with the classes identifying which objects a spreadsheet contains and how its data can evolve, and the attributes specifying the contents that are allowed in the spreadsheet. The following examples show the structure of Tabula models and a concise visual representation for them, with the semantics explained afterwards.

Example 1 (List of items, continuing from p. 27)**.** Recalling the spreadsheet from the example, it contains a simple structure where the first line is static, lines 2 through 4 have a similar structure and content to each other, i.e., a description of an item and a value, and the last line (5) has a formula that references elements in lines 2 to 4.

The Tabula for this example contains a two-by-three grid with two classes, where one class (**Items**) is for the whole list of items and another class (**Item**) for each item (lines 2, 3 and 4 in the spreadsheet):

$$\big(\text{“ItemList”}, \{(\text{“Items”}((0,0),(1,2))\text{None}), (\text{“Item”}((0,1),(1,1))\text{Down})\}, g\big)$$

where

g =

Items	
desc=""	value=0
Total	total=SUM(Item.value)

The graphical representation of this Tabula is presented in figure 3.2. It describes the grid in its tabular format and uses the background colour to identify the classes each cell belongs to. On the side there is a legend relating the colour to the class since the name is not always displayed in the table: the class **Items** (in yellow) has a label in its top-left cell but the class **Item** (in orange) does not. Moreover, in some cases, the label can be different from the class name. The arrow pointing down on the left of the attribute `desc` indicates that the class **Item** expands down.

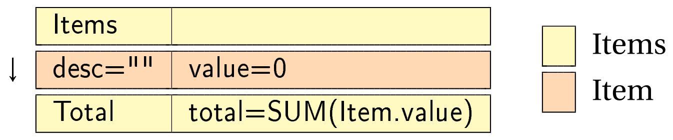


Figure 3.2: Tabula specifying a spreadsheet to hold a list of items as described in example 1.

The layout of a Tabula can also expand to the right. Moreover, the expansion area can contain multiple columns or rows. This configuration is presented in example 4.

Example 4. A spreadsheet can contain information about an item along the years. This information, which corresponds to `value` in example 1, can be stored in multiple attributes. In this example, the base class **Items** is kept, but the class **Item** is left out for simplification. Instead, a new class **Year** is created. It contains the attributes `year`, `stock` and `sold` to represent the year an object refers to, the stock of the item at the end of the year and the number of items sold that year, respectively. Finally, this information will be for an item *apple* and an average sold across the years is calculated at the end. This is pictured in figure 3.3.

In example 5, the conjugation of the configurations introduced in examples 1 and 4 is presented. Its result contains an additional class that relates the expandable classes from both of the previous Tabulae.

Example 5. Merging examples 1 and 4 implies to relate two data sets: the list of items and the data for the years. Since the former was specified vertically and the latter horizontally, this merging is trivial. The resulting Tabula contains the class **Items** from both examples, the class **Item** from example 1 and the class **Year** from example 4. Moreover, it contains a new class to represent the attributes that are in the relation between **Item** and **Year** named

	A	B	C	D	E	F
1	**Items**	2012		2013		
2		**stock**	**sold**	**stock**	**sold**	**Average sold**
3	apple	5	12	4	16	=AVERAGE(C3,E3)

(a) Spreadsheet definition.

	A	B	C	D	E	F
1	**Items**	2012		2013		
2		**stock**	**sold**	**stock**	**sold**	**Average sold**
3	apple	5	12	4	16	14

(b) Spreadsheet execution.

→

Items	year=2000		
	stock	sold	Average sold
apple	stock=0	sold=0	avg=AVERAGE(sold)

🟨	Items
🟦	Year

(c) Tabula specification of the spreadsheet.

Figure 3.3: Example of the yearly stock and quantity sold of apples.

ItemYear, which expands in both directions: to the right and down. This is specified as:

$$\begin{aligned}\big(\text{“ItemYears”}, \{(&\text{“Items”}, ((0,0),(3,3)), \text{None})\\ ,(&\text{“Item”}, ((0,2),(3,2)), \text{Down})\\ ,(&\text{“Year”}, ((1,0),(2,3)), \text{Right})\\ ,(&\text{“ItemYear”}, ((1,2),(2,2)), \text{Both})\},\ g\big)\end{aligned}$$

where

g =

Items	year=200		
	Stock	Sold	Average Sold
desc=""	stock=0	sold=0	avg=Average(sold)
Total	total=SUM(Item.stock)		

or graphically as in figure 3.4.

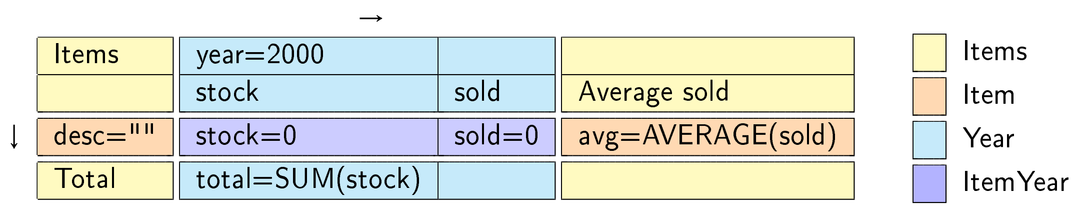
Classes are layed out in a layered fashion, where smaller classes stay on top of larger ones.

	A	B	C	D	E	F
1	**Items**	2012		2013		
2		**stock**	**sold**	**stock**	**sold**	**Average sold**
3	apple	5	12	4	16	=AVERAGE(C3,E3)
4	banana	2	10	3	12	=AVERAGE(C4,E4)
5	cherry	8	9	1	3	=AVERAGE(C5,E5)
6	**Total**	=SUM(B3:B5)		=SUM(D3:D5)		

(a) Spreadsheet definition.

	A	B	C	D	E	F
1	**Items**	2012		2013		
2		**stock**	**sold**	**stock**	**sold**	**Average sold**
3	apple	5	12	4	16	14
4	banana	2	10	3	12	11
5	cherry	8	9	1	3	6
6	**Total**	15		8		

(b) Spreadsheet execution.



(c) Tabula specification of the spreadsheet.

Figure 3.4: Example of the spreadsheet and Tabula for the relation between items and the yearly stock and quantities sold.

This layering is defined by the range inclusion relation.

Definition 9 (Range inclusion)**.** *For two ranged elements a and b, $a \sqsubseteq b$ means that all elements of the range a are included in the range b, i.e.:*

$$\sqsubseteq : R \rightarrow R \rightarrow \{True, False\}$$

$$a \sqsubseteq b \overset{def}{=} (\text{left}\, b \leq \text{left}\, a) \wedge (\text{right}\, a \leq \text{right}\, b) \wedge (\text{top}\, b \leq \text{top}\, a) \wedge (\text{bottom}\, a \leq \text{bottom}\, b)$$

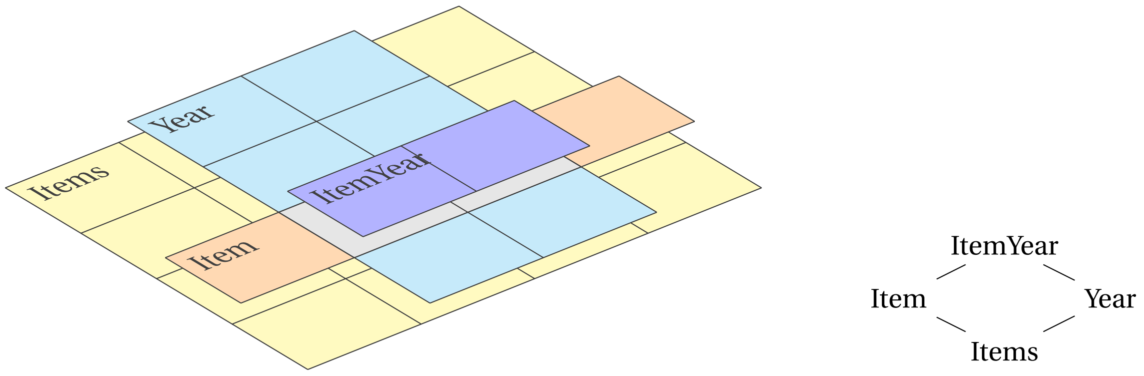
Definition 10 (Range member)**.** *For a point (x, y) and a range r, $(x, y) \sqsubseteq r$ means that (x, y) is* *a point within the bounds of r, i.e.:*

$$\sqsubseteq : P \to R \to \{True, False\}$$

$$(x, y) \sqsubseteq r \overset{def}{=} (\text{left}\, r \le x) \land (x \le \text{right}\, r) \land (\text{top}\, r \le y) \land (y \le \text{bottom}\, r)$$

Definition 11 (Class layers)**.** *The class layers of a Tabula t is defined by a partial order $\le_{\mathscr{L}}$ where, for the classes C of t, $c_0, c_1 \in C$, $c_0 \le_{\mathscr{L}} c_1 \Leftrightarrow c_0 \sqsubseteq c_1$.*

A graphical representation of the class layers for example 5 is demonstrated in figure 3.5, alongside the ordering of the classes.



(a) Graphical representation of the class layers. (b) Class partial ordering.

Figure 3.5: Class layers from example 5.

From the class layers comes also the notion of children and parents of a class. The children of a class are given by the covering relation $\lessdot_{\mathscr{L}}$.

Definition 12 (Covering relation)**.** *The covering relation is a subset of the partial order $\le_{\mathscr{L}}$ that relates elements without elements in between, i.e., $a \lessdot_{\mathscr{L}} b$ if and only if $a <_{\mathscr{L}} b$ and there is no element c such that $a <_{\mathscr{L}} c <_{\mathscr{L}} b$, where $<_{\mathscr{L}}$ contains elements $a \le_{\mathscr{L}} b$ such that $a \ne b$.*

Definition 13 (Well-formed Tabula layout)**.** *A well-formed Tabula layout must comply with the following rules:*

1. *There must always be a base class with the size of the Tabula grid. In the running example, this base class is **Items**.*

2. *Inner classes must either use the whole height or the whole width of the base class. In the former case, the class is called a* vertical class*, possibly expanding to the right, and in the latter case the class is called an* horizontal class*, possibly expanding down. This rule prevents spreadsheet layout deformation when adding new instances of a class.*

3. *An horizontal class must leave a line above and another below of the class below. A vertical class must leave a column to the left and another one to the right of the class below. This rule prevents ambiguity in the layer order and allows for a simpler visual representation. Note that there is no need to leave space between two classes at the same level.*

4. *Two classes are allowed to intersect when one is contained by the other, i.e., the intersection area is the area of the contained class, or when a class is a vertical one and the other is an horizontal class.*

5. *When a vertical class intersects with an horizontal one, a relation class must use the intersection area.*

6. *Only relation classes can expand both to the right and down at once.*

Definition 13 has a similar objective as the tiling rules for ClassSheets, ensuring a correct layout of the classes.

The references in the model are made to other attributes in the same Tabula. References only can be made using the name of the attribute or by prepending the name of the attribute with the name of the respective class. This characteristic minimizes spreadsheet error 2.6 (wrong selection) since it only allows to select existing attributes and due to the use of explicit names (if suitable names are given to the attributes). The former is called a *simple* reference whilst the latter option is said to be a reference using the *fully qualified* name of the attribute (see table 3.1 for an example). When using a fully qualified reference, the reference is unique as it is forbidden to have two classes with the same name and also to have two attributes within a single class with the same name. However, when using simple references, there can be ambiguity about the attribute a references is to.

Definition 14 (Attribute Name Resolution)**.** *Attribute name resolution consists in finding the concrete attribute that is the target of a reference. A simple reference is resolved by searching attributes with the name of the reference through different contexts in the following order:*

1. *within the same class;*
2. *within an inner class, i.e., a class that is contained in the class where the reference is made;*
3. *within an outer class, i.e., a class that contains the class where the reference is made;*
4. *within any other class.*

The first attribute found is the one that is referenced. This attribute must be unique in its context.

Table 3.1: Attributes from example 5, their class and fully-qualified name.

Attribute	**Class**	**Fully qualified name**
desc	Item	Item.desc
avg	Item	Item.avg
year	Year	Year.year
total	Year	Year.total
stock	ItemYear	ItemYear.stock
sold	ItemYear	ItemYear.sold

Definition 15 (Well-formed Tabula content)**.** *A well-formed Tabula content must comply with the following rules:*

1. *Classes have distinct names.*
2. *Attributes within a class have distinct names.*
3. *References are only to existing attributes.*
4. *Cells follow the given grammar and function attributes are in a correct number and of the correct type.*
5. *Constraints are valid.*

Definition 16 (Well-formed Tabula)**.** *A Tabula is said well-formed if and only if both its content and layout are well-formed (definitions 13 and 15).*

3.2 Tabula Instance

A spreadsheet that is a Tabula instance is an enhanced representation of the respective spreadsheet. On top of the normal spreadsheet, it contains a list of classes instances (objects), and does not need any named range.

Definition 17 (Tabula Instance)**.** *A Tabula instance I is a tuple $I = (N, O^\star, G)$ where*

- *N is the name;*
- *$O^\star$ is the set of objects O;*
- *G is the grid.*

Classes are instantiated as objects. These objects, similarly to classes, are bound to a range in the instance grid and have the name of their class. However, unlike classes, they do not have information about expansion.

Definition 18 (Object)**.** *An object O is a pair $O = (N, R)$ where*

- *N is the name;*
- *R is the range.*

The cells within a Tabula instance are identical to spreadsheet cells, following the same grammar.

An instance is well-formed if it has its object correctly layed out in its grid, thus following rules similar to the ones for a well-formed Tabula layout:

Definition 19 (Well-formed instance)**.** *A well-formed instance must comply with the following rules:*

1. *There must always be a base object with the size of the instance.*

2. *Inner objects must either use the whole height or the whole width of the base object. In the former case, the object is called a* vertical *object, and in the latter case the object is called an* horizontal *object. From this rule are excepted relation objects.*

3. *An horizontal object must leave a line above and another below of the object below. A vertical class must leave a column to the left and another one to the right of the object below.*

4. *A vertical object and an horizontal one are allowed to intersect when one is contained by the other, i.e., the intersection is the area of the contained object, or when an object is a vertical one and the other is an horizontal object.*

5. *When a vertical class intersects with an horizontal one, a relation object muse use the intersection area.*

Additionally, a Tabula instance must conform to its Tabula:

Definition 20 (Tabula instance conformance)**.** *A Tabula instance must comply with the following rules in order to conforms to its Tabula model:*

1. *there is a base object corresponding to the Tabula's base class;*

2. *if a class as a corresponding object, then all its non-expanding sub-classes have a corresponding object;*

3. *the cover relation of the objects follows the cover relation of the classes;*

4. *all cells in the instance are well-typed according to their respective cell in the Tabula.*

The relation between a Tabula cell $t_{(t_x,t_y)}$ and an instance cell $i_{(i_x,i_y)}$ is given by the relation of positions in the Tabula and in the instance:

$$t_{(t_x,t_y)} \ominus i_{(i_x,i_y)} \equiv (t_x, t_y) \ominus (i_x, i_y)$$

The notation $\cdot \ominus \cdot$ is used to denote both a relation between cells and between points. Moreover, it is used to denote a relation between the elements of the points, i.e., $t_x \ominus i_x$ and $t_y \ominus i_y$, since the coordinates can be looked at independently because, if an expansion exists, it uses the whole width or the whole height of the Tabula or instance.

A point (t_x, t_y) in a Tabula t with classes C is related to all points (i_x, i_y) in an instance i with objects O such that:

$$\begin{aligned} & i_x = \text{left } o + (t_x - \text{left } c - \Delta_{cx}) + \Delta_{ox} \\ \wedge\ & i_y = \text{top}\, o + (t_y - \text{top } c - \Delta_{cy}) + \Delta_{oy} \\ \wedge\ & c \boxminus o \ \wedge\ (t_x, t_y) \sqsubseteq c \ \wedge\ c \in C \ \wedge\ o \in O \end{aligned}$$

where

- $\boxminus$ is the relation between classes and objects;
- Δ_{cx} is the sum of the widths of the inner classes between the left of the class containing the Tabula point and the column of such point;
- Δ_{cy} is the sum of the heights of the inner classes between the top of the class containing the Tabula point and the row of such point;
- Δ_{ox} is the sum of the widths of the objects from the inner classes mentioned for Δ_{cx} which are within the current object;
- Δ_{oy} is the sum of the heights of the objects from the inner classes mentioned for Δ_{cy} which are within the current object.

Definition 21 (Well-typed cell)**.** *An instance cell is well typed by its respective Tabula cell if the rules bellow are satisfied. For these rules, it is assumed an environment* $\Gamma = (T, I, C_T, C_I)$*, where* T *is the Tabula,* I *is the instance,* C_T *is a Tabula cell at some position and* C_I *is an instance cell at some position.*

$$\frac{v_i \in \mathbb{N} \quad v_t \in \mathbb{N}}{v_i :: a = v_t}\ \textit{(numeric attribute)} \qquad \frac{v_i \in \mathbb{T} \quad v_t \in \mathbb{T}}{v_i :: a = v_t}\ \textit{(textual attribute)}$$

$$\frac{v_i :: a = v_t \quad [\![p]\!]\, v_i}{v_i \;::\; a = v_t; p} \textit{(constrained attribute)} \qquad \frac{e_i :: e_t \quad e_i \notin \mathbb{N} \quad e_i \notin \mathbb{T}}{= e_i \;::\; a = e_t} \textit{(formula)}$$

$$\frac{v_i \in \mathbb{N} \quad v_t \in \mathbb{N} \quad v_i = v_t}{v_i :: v_t} \textit{(number)} \qquad \frac{v_i \in \mathbb{T} \quad v_t \in \mathbb{T} \quad v_i = v_t}{v_i :: v_t} \textit{(text)}$$

$$\frac{a_{i\,0} :: a_{t\,0} \quad a_{i\,1} :: a_{t\,1} \quad \ldots \quad a_{i\,n} :: a_{t\,n} \quad \varphi_i = \varphi_t}{\varphi_i(a_{i\,0}, a_{i\,1}, \ldots, a_{i\,n}) \;::\; \varphi_t(a_{t\,0}, a_{t\,1}, \ldots, a_{t\,n})} \textit{(function)}$$

$$\frac{e_i :: e_t \quad \Box_i = \Box_t}{\Box_i e_i \;::\; \Box_t e_t} \textit{(signed expression) where } \Box \in \{+, -\} \qquad \frac{e_i :: e_t}{e_i\% :: e_t\%} \textit{(percentage)}$$

$$\frac{a_{i_0} :: a_{t\,0} \quad a_{i_1} :: a_{t\,1} \quad \Box_i = \Box_t}{a_{i\,0} \Box_i a_{i\,1} :: a_{t\,0} \Box_t a_{t\,1}} \textit{(operation) where } \Box \in \{<, \leqslant, \geqslant, >, =, \neq\}$$

$$\frac{\mathscr{R}\; t\; c.a\,(t_x, t_y)\,(i_x, i_y) \;=\; \{r_0, r_1, \ldots, r_n\}}{t, i, (t_x, t_y), (i_x, i_y) \vdash r_0 \cup r_1 \cup \cdots \cup r_n \;::\; c.a} \textit{(reference)}$$

The environment of the previous definition is only required to verify cell references. Well-typed cells prevent spreadsheet errors 2.1 (incorrect type) and 2.4 (incorrect relativity). It also prevents error 2.5 (outdated range) if type checking is ensured whenever changes are performed (further details in chapter 4).

Definition 22 (Initial instance)**.** *An initial instance is one that contains the minimum contents, with default values, that is conforming to a Tabula.*

Initial instances can be generated from a Tabula by:

- copying the Tabula;
- removing all expandable classes, including their contents;
- converting all remaining classes to objects;
- translating attribute references to cell references; and,
- converting attribute cells to spreadsheet cells.

3.3 Spreadsheet Recognition

Spreadsheet tables do not have the information present in Tabula instances. In order to apply model-driven techniques to these tables, they are first converted into a Tabula instance using a Tabula as a guide. This is called spreadsheet recognition.

Spreadsheet recognition makes use of techniques applied to pictures, also known as two-dimensional language parsing. Some background on two-dimensional languages are presented in the next section (3.3.1), with a description of generic recognizers for these languages afterwards (section 3.3.2) and finally applying the concept to spreadsheet tables (section 3.3.3).

3.3.1 Two-Dimensional Languages

Existing spreadsheet models do not have a method to read a spreadsheet with the help of a model that indicates its structure and contents. This can be used for several ends, including verifying the conformance of a spreadsheet. Nevertheless, some work has been done in finding pre-defined patterns in spreadsheets, e.g., using inference rules (Abraham & Erwig, 2006b) or using automata (Hermans, 2013). The former example provides a set of inference rules with no algorithm on how to apply them, while the latter provides a concrete algorithm to parse a spreadsheet based on a grammar. Moreover, the latter is based on one-dimensional algorithms and the parse is tree-based

In this work, an algorithm made to parse two-dimensional pictures is used. This algorithm is better suited for spreadsheets due to their two-dimensional nature, where each cell corresponds to one element of a picture's two-dimensional array.

Concretely, a two-dimensional language is used to describe a picture, which is a rectangular array of symbols from a finite alphabet Σ, with # identifying the picture boundaries.

Example 6. The 3×2 picture with $\Sigma = \{a\}$ is

#	#	#	#	#
#	a	a	a	#
#	a	a	a	#
#	#	#	#	#

In the literature, many concepts and techniques were proposed to support two-dimensional language recognition. One such technique relies on *two-dimensional on-line tessellation automata* (2OTA) (Inoue & Nakamura, 1977), described in the next section.

3.3.2 A Two-Dimensional Language Recognizer

A two-dimensional language recognizer takes a language specification and a picture, and checks if the picture conforms to the given language. The work presented in this dissertation uses *two-dimensional on-line tessellation automata* (2OTA) (Inoue & Nakamura, 1977) to convert a spreadsheet into a Tabula instance, checking its contents and adding the required meta-information.

Definition 23. *A non-deterministic two-dimensional on-line tessellation automaton, referred as 2OTA, is completely defined by* $A = (\Sigma, Q, I, F, \delta)$ *where:*

- Σ *is the input alphabet;*
- Q *is a finite set of states;*
- $I \subseteq Q$ *is the set of "initial" states;*
- $F \subseteq Q$ *is the set of "final" states;*
- $\delta : Q \times Q \times \Sigma \longrightarrow 2^Q$ *is the transition function.*

The recognition of a picture using a 2OTA consists in assigning states to each cell, starting at the top-left cell and ending at the bottom-right one. A successful run assigns a state from the set of the 2OTA's final states to this last cell. The run for a picture p has $k = \text{width}_p + \text{height}_p - 1$ steps. At each step, states are assigned to the cells (x, y) with $x + y - 1 = k$, thus traversing the picture by diagonals.

3.3.3 Tabula Instance Recognizer

The Tabula instance recognizer, guided by a given Tabula, reads some regular spreadsheet into one of its instances. In order to perform this, the Tabula model is converted to a 2OTA, which is then run on the spreadsheet grid.

The alphabet the 2OTA works on is the set of cells that are possible to write in a spreadsheet and the boundary symbol (#).

The states used in the 2OTA are $Q = \{\mathit{Invalid}, \mathit{Border})\} \cup P$, where P is the set of cell positions in the given Tabula. The state *Invalid* indicates that an invalid state was reached, *Border* is the state of a border cell and a position indicates that a cell is represented by the cell at that position in the Tabula.

The initial state for a Tabula t is $I = \{\mathit{Border}\}$ and the final one is $F = \{(\text{width}_t - 1, \text{height}_t - 1)\}$.

The transition function δ has four cases: for the top-left cell, cells with a border on the left, cells with a border on the top and cells in the middle. For the first case and initial case, the top-left cell is analysed, checking if it type checks with the top-left cell of the Tabula:

$$\delta(\mathit{Border}, \mathit{Border}, a) = \text{if } a :: t_{(0,0)} \text{ then } \{(0,0)\} \text{ else } \{\mathit{Invalid}\}$$

Cells with a border on the left are checked against the cell above, given in *sAbove*, and checked against the type of cell below it in the model using the auxiliary function firstColumnCell. Moreover, in case of the above cell being contained in an expandable class, it is checked if the current cell is in a new object with the auxiliary function vRepetition.

$$\begin{aligned}
\delta(\mathit{Border}, \mathit{sAbove}, a) &= \text{let } s = \text{firstColumnCell} \cup \text{vRepetition} \\
&\qquad \text{in if } s \neq \{\} \text{ then } s \text{ else } \{\mathit{Invalid}\} \\
\text{firstColumnCell} &= \left\{(0,y) \mid y \in [1, \text{height}_t - 1] \wedge \mathit{sAbove} = (0, y-1) \wedge a :: t_{(0,y)}\right\} \\
\text{vRepetition} &= \Big\{(x, \text{top}\, c) \mid c \in \text{classes}\ t \wedge \text{expands}_V\ c \wedge a :: t_{(x, \text{top}\, c)} \\
&\qquad \wedge \left(\text{if } \mathit{sLeft} = \mathit{Border} \text{ then } x \in \{\text{left}\, c\} \text{ else } x \in [\text{left}\, c, \text{right}\, c]\right) \\
&\qquad \wedge \mathit{sAbove} = (x, \text{bottom}\, c) \wedge \left(\mathit{sLeft} = (x-1, \text{top}\, c) \vee \mathit{sLeft} = \mathit{Border}\right)\Big\}
\end{aligned}$$

Similarly to the previous case, cells with a border above are checked against the cell on the left *sLeft* using the auxiliary function firstRowCell and then type checked against the type of the cell to its right. There is also a check to see if the cell begins another repetition using the auxiliary function hRepetition.

$$\begin{aligned}
\delta(\mathit{sLeft}, \mathit{Border}, a) &= \text{let } s = \text{firstRowCell} \cup \text{hRepetition} \\
&\qquad \text{in if } s \neq \{\} \text{ then } s \text{ else } \{\mathit{Invalid}\} \\
\text{firstRowCell} &= \left\{(x,0) \;\middle|\; x \in [1, \text{width}_t - 1] \wedge \mathit{sLeft} = (x-1,0) \wedge a :: t_{(x,0)}\right\} \\
\text{hRepetition} &= \Big\{(\text{left}\, c, y) \;\Big|\; c \in \text{classes}\ t \wedge \text{expands}_H\, c \wedge a :: t_{(\text{left}\, c, y)} \\
&\qquad \wedge \left(\text{if } \mathit{sAbove} = \mathit{Border} \text{ then } y \in \{\text{top}\, c\} \text{ else } y \in [\text{top}\, c, \text{bottom}\, c]\right) \\
&\qquad \wedge \mathit{sLeft} = (\text{right}\, c, y) \wedge \left(\mathit{sAbove} = (\text{left}\, c, y-1) \vee \mathit{sAbove} = \mathit{Border}\right)\Big\}
\end{aligned}$$

The final case is for inner cells, checking the cell to the left *sLeft* and the one above *sAbove* and then type checking the current cell using the auxiliary function middleCell. Repetitions are given by the cells to the left and above and thus no further checking is necessary.

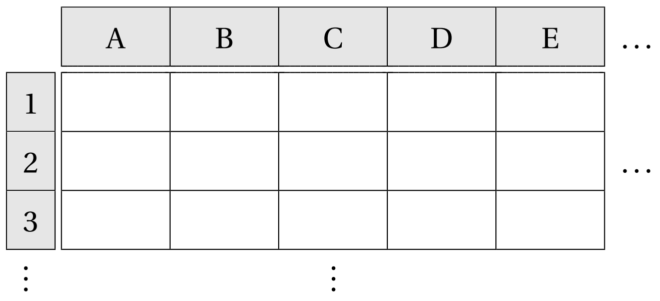
$$\begin{aligned}
\delta(\mathit{sLeft}, \mathit{sAbove}, a) &= \text{let } s = \text{middleCell} \\
&\qquad \text{in if } s \neq \{\} \text{ then } s \text{ else } \{\mathit{Invalid}\} \\
\text{middleCell} &= \Big\{(x,y) \;\Big|\; x \in [1, \text{width}_t - 1] \wedge y \in [1, \text{height}_t - 1] \\
&\qquad \wedge \mathit{sLeft} = (x-1, y) \wedge \mathit{sAbove} = (x, y-1) \wedge a :: t_{(x,y)}\Big\}
\end{aligned}$$

After recognizing a Tabula instance successfully, the table is converted to an instance by copying the grid and parsing the states given to each cell. With this parsing, each range of cells parsed is tagged with its corresponding object information.

3.4 Embedded Tabula

The embedding of a Domain-Specific Language (DSL) within a general-purpose language is a well-known technique (Swierstra, Henriques, & Oliveira, 1999). It exploits features of the host language as native of the domain-specific one and reduces the effort to create a complete development environment for the DSL.

In this case, the general-purpose language is the spreadsheet: a visual language containing a grid of cells which can have additional formatting. The plain view of a spreadsheet is like:



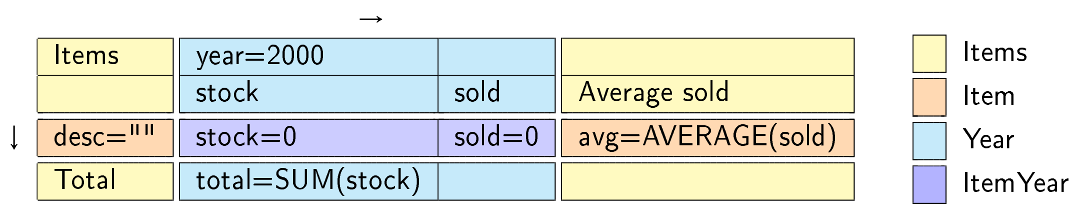
The embedding of the ClassSheet language within spreadsheets themselves (J. Cunha et al., 2011; J. Cunha, Fernandes, Mendes, & Saraiva, 2015) provided an unique environment that allowed to experiment with model-driven spreadsheets and evaluate their results without having to introduce different tools to spreadsheet users. This embedding was possible due to the ClassSheet visual language being similar to spreadsheets, but the ClassSheet visual required some changes to be technically feasible.

Tabula originated from ClassSheets, keeping a similar visual language which is also similar to spreadsheets. However, Tabula has additional features that are not easily embedded into spreadsheets, including the support for expandable classes within other expandable classes. Therefore, only a subset of the Tabula language is supported in this embedding.

The embedding of Tabula follows closely the visual notation used in section 3.1. Its contents are represented as follows:

- each cell of the Tabula grid corresponds to a cell in the spreadsheet;
- each class in the Tabula gives an unique colour to the classes it contains;
- if a class expands to the right, an additional column is present in the spreadsheet on the right of the cells of that class, where the cells have ellipses;
- if a class expands down, an additional row is present in the spreadsheet below the cells of that class, where the cells have ellipses.

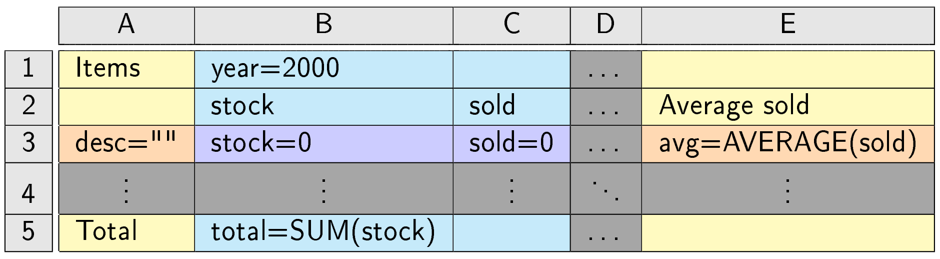
The Tabula of example 5 and its corresponding embedding are displayed in fig. 3.6.



Items	year=2000		
	stock	sold	Average sold
desc=""	stock=0	sold=0	avg=AVERAGE(sold)
Total	total=SUM(stock)		

🟨	Items
🟧	Item
🟦	Year
🟪	ItemYear

(a) Visual representation of the Tabula.



	A	B	C	D	E
1	Items	year=2000		...	
2		stock	sold	...	Average sold
3	desc=""	stock=0	sold=0	...	avg=AVERAGE(sold)
4	⋮	⋮	⋮	⋱	⋮
5	Total	total=SUM(stock)		...	

(b) Embedding of the Tabula.

Figure 3.6: Tabula from example 5 (3.6a) and its corresponding embedding (3.6b).

Chapter 4

Spreadsheet Evolution

Evolution is a need in most software, where maintenance or upgrades are required. Spreadsheets are no exception. They can have a long lifetime (e.g., in a study by Hermans, Pinzger, and van Deursen (2011), the average lifetime of the spreadsheets was longer than five years), with requirements being changed and bugs being corrected.

A spreadsheet can be evolved at two levels: at its specification and at its implementation. The former corresponds to Tabula evolution, detailed in section 4.1, and the latter to Tabula instance evolution, detailed in section 4.2. Evolution at either level reflects the possible basic atomic operations that can be performed on plain spreadsheets:

add sheet	— add a new empty sheet to a spreadsheet;
delete sheet	— delete a sheet, discarding its contents, from a spreadsheet;
rename sheet	— rename a sheet, without any change to its contents;
add column	— add a new empty column to a sheet;
delete column	— delete a column, discarding its contents, from a sheet;
add row	— add a new empty row to a sheet;
delete row	— delete a row, discarding its contents, from a sheet;
insert cell	— add a new cell, shifting existing ones in a selected direction; and,
set cell	— define new contents for a cell.

All but the last operation above perform changes to the layout of the spreadsheet. In some cases, further steps are required to obtain a correct spreadsheet, e.g., updating references. Furthermore, some of these steps can be performed automatically, like the update of references when adding or removing rows or columns. However, in some cases, it produces invalid or unexpected results and in other cases an update is expected but is not performed (see spreadsheet error 2.5).

4.1 Tabula Evolution

The structure the Tabula language follows closely the one of spreadsheets. However, it only models sheets, hence operations on sheets have no equivalent operation on Tabulae. Furthermore, the Tabula language contains a significant restriction: one cannot insert a single cell into the model without adding a whole column or a whole row. This restriction is important since it prevents layout deformations and simplifies the mechanics of models. Therefore, from the operations on spreadsheets presented previously, also the *insert cell* operation cannot be performed on Tabulae. Moreover, three additional basic atomic operations related to classes are available for Tabulae. This results in the following operations:

add class	— add a class to an existing range of a Tabula;
delete class	— delete a class from a Tabula, without deleting its contents;
rename class	— changes the name of a class;
add column	— add a new empty column to a Tabula;
delete column	— delete a column, discarding its contents, from a Tabula;
add row	— add a new empty row to a Tabula;
delete row	— delete a row, discarding its contents, from a Tabula; and,
set cell	— define new contents for a cell.

Each of these operations, with some arguments applied, operate on a well-formed Tabula and return a new well-formed Tabula, thus making the operations composable.

Add class. Adds a class to a Tabula, without performing changes in its grid. The type of this operation is:

$$addClass_T : Class \rightarrow Tabula \rightarrow Tabula$$

The given class should follow the rules to have a well-formed Tabula layout (definition 13). This operation adds the required relation classes, otherwise it could create an ill-formed Tabula.

Delete class. Deletes a class from a Tabula, without performing changes in its grid. The type of this operation is:

$$delClass_T : Class \rightarrow Tabula \rightarrow Tabula$$

The class to delete should not have attributes with the same name as its parent classes in order not to produce a class with duplicated attribute names, thus creating ambiguity when referencing one of those attributes. This restriction is imposed by item 2 of the well-formed Tabula content definition. This operation deletes first its inner classes, applying the same restrictions.

Rename class. Renames a class in a Tabula. The type of this operation is:

$$renameClass_T : Class \rightarrow Name \rightarrow Tabula \rightarrow Tabula$$

The new name should not be used by any another class in order to be unique in the resulting Tabula. Moreover, this operation updates all fully-qualified attribute references to the class.

This operation could be achieved by removing the given class and then adding it again with the new name. However, in order to be able to perform the renaming in two steps, the composite operation would have to be as restrictive as the $delClass_T$ operation.

Add column. Adds a column to the Tabula at a relative position to the given index:

$$addColumn_T : RelPos_T \rightarrow Index \rightarrow Tabula \rightarrow Tabula$$

The relative position $RelPos_T$ can have one of the following values:

- *BeforeClass* — add the column at the given index, but outside of the class at that index if it is its leftmost column;
- *BeforeIndex* — add the column at the given index, within the class at the same index;
- *AfterIndex* — add the column after the given index, within the class at the same index;
- *AfterClass* — add the column after the given index, but outside of the class at that index if it is its rightmost column.

Delete column. Deletes the column at the given index from the Tabula:

$$delColumn_T : Index \rightarrow Tabula \rightarrow Tabula$$

If this column is the last one of a class, that class is also deleted.

Add row. Adds a row to the Tabula at a relative position to the given index:

$$addRow_T : RelPos_T \rightarrow Index \rightarrow Tabula \rightarrow Tabula$$

The relative position $RelPos_T$ can have the same values as for the *add column* operation, with the index being relative to the row instead of the column.

Delete row. Deletes the row at the given index from the Tabula:

$$delRow_T : Index \rightarrow Tabula \rightarrow Tabula$$

Similarly to the deletion of columns, if the row is the last one of a class, that class is also deleted.

Set cell. Defines the cell at the given position of the Tabula.

$$setCell_T : Position \rightarrow Cell_T \rightarrow Tabula \rightarrow Tabula$$

The contents of the cell must be as such that they do not break any rule of the well-formed Tabula content definition.

4.2 Instance Evolution

Instance evolution is similar to spreadsheet evolution, with the support of the classes to add new objects with the correct structure and to update the references when required. However, it is not allowed to edit arbitrary cells in the instance. This limitation is described below in the definition of the *set cell* operation.

Add object. Adds an object to the instance:

$$addObject_I : Object \rightarrow Instance \rightarrow Instance$$

The object to be added should be such that it keeps the conformance with the Tabula. This operation adds the required columns or rows. It also adds the required inner objects. References are updated afterwards.

Delete object. Deletes an object from the instance, including its columns and rows:

$$delObject_I : Object \rightarrow Instance \rightarrow Instance$$

The inner objects are also deleted. References are updated afterwards.

Rename object. Changes the name of an object in the instance:

$$renameObject_I : Object \rightarrow Name \rightarrow Instance \rightarrow Instance$$

This operation maintains the structure of the instance intact and references are position-based instead of name-based, thus references do not need to be updated.

Add column. Adds a column to the instance at a relative position to the given index:

$$addColumn_I : RelPos_I \to Index \to Instance \to Instance$$

References are updated afterwards. The relative position *RelPos$_I$* can have one of the following values:

- *BeforeObject* — add the column at the given index, before the object at the same index;
- *BeforeIndex* — add the column at the given index, within the object at the same index;
- *AfterIndex* — add the column after the given index, within the object at the same index;
- *AfterObject* — add the column after the given index, after the object at the same index.

From these, only *BeforeIndex* and *AfterIndex* are allowed for user transformations, being *BeforeObject* and *AfterObject* reserved for the bidirectional transformations.

Delete column. Deletes a column from the instance:

$$delColumn_I : Index \to Tabula \to Tabula$$

References are updated afterwards.

Add row. Adds a row to the instance at a relative position to the given index:

$$addRow_I : RelPos_I \to Index \to Instance \to Instance$$

References are updated afterwards. The relative poosition *RelPos$_I$* can have the same values as for the *add column* operation, with the index being relative to the row instead of the column.

Delete row. This deletes a row from the instance. It has the index of the row to delete as a parameter.

$$delRow_I : Index \rightarrow Instance \rightarrow Instance$$

References are updated afterwards.

Set cell. Defines the contents of a cell at the given position in the Tabula. Its type is:

$$setCell_I : Position \rightarrow Cell \rightarrow Instance \rightarrow Instance$$

This operation is only valid if setting a cell that implements an input value. The content should be either numeric or textual (i.e., formulas are not allowed), with the type identified in the Tabula and the value should be a valid one relative to the constraint defined in the model, if any.

4.3 Bidirectional Transformations

Performing transformations on either the Tabula or corresponding instance can lead to states where the conformance between the instance and the Tabula is not met. Additional steps are required to restore the conformance. The chosen solution is to perform coupled transformations (Lämmel, 2004), i.e., performing transformations in all artefacts in a way the conformance is maintained. More specifically, since operations are allowed in both the Tabula and in the instance at any given moment, bidirectional transformations are used.

Bidirectional transformations are used to keep the conformance relation between the instance and its model intact. In many contexts, these kind of transformations are state based. However, in this case, the approach is based on operations, as spreadsheets are a highly interactive tool and the operations are available to the system, providing more information about the changes to the different artefacts. This allows a coupled transformation system to be performed synchronously.

The bidirectional transformation environment for Tabula models and respective instances is designed with user interaction in mind for use in a spreadsheet host system. Thus, the environment is used in an online setting, where it reacts immediately to each user modification, and the transformations are based on operations users perform. In this environment, an user operation may require several transformations, both in the artefact where the user performs the operation and its related artefact. Therefore, the bidirectional transformation environment is defined by a pair of unidirectional transformations with the following signature:

$$\begin{aligned} to_{T,I} &: \mathrm{Op}_T^\star \to \mathrm{Op}_I^\star \\ from_{T,I} &: \mathrm{Op}_I^\star \to \mathrm{Op}_T^\star \end{aligned}$$

The $to_{T,I}$ transformation takes a sequence of operations on the Tabula, from the operations defined in section 4.1, and returns a sequence of operations to be performed on the instance. The $from_{T,I}$ transformation takes a sequence of operations on the instance, from the operations defined in section 4.2, and returns a sequence of operations to be performed on the Tabula.

Having a Tabula t and a conforming instance i, performing a sequence of operations $op_T^\star$ on t, yielding a Tabula t', and the sequence of operations $to_{T,I}\ op_T^\star$ on i, yielding an instance i', results in i' conforming to t'. In other words, the conformance relation is maintained. Similarly, performing a sequence of operations $op_I^\star$ on i, yielding an instance i'', and the sequence of operations $from_{T,I}\ op_I^\star$ on t, yielding a Tabula t'', results in i'' conforming to t''.

$$\begin{tikzcd} t \arrow[d, "op_T^\star"'] & i \arrow[l, dashed, "\text{conforms to}"'] \arrow[d, "to_{T,I}\ op_T^\star"] \\ t' & i' \arrow[l, dashed, "\text{conforms to}"'] \end{tikzcd}$$

(a) Evolution of the Tabula and co-evolution of the instance.

$$\begin{tikzcd} t \arrow[d, "from_{T,I}\ op_I^\star"] & i \arrow[l, dashed, "\text{conforms to}"'] \arrow[d, "op_I^\star"] \\ t'' & i'' \arrow[l, dashed, "\text{conforms to}"'] \end{tikzcd}$$

(b) Evolution of the instance and co-evolution of the Tabula.

Figure 4.1: Conformance after performing operations.

However, the supported sequences of operations is limited and defined based on concrete operations required by end users. From the point of view of the user, the following operations on Tabulae are available:

- add class;
- delete class;
- rename class;
- add Tabula column;
- delete Tabula column;
- add Tabula row;
- delete Tabula row; and,
- set Tabula cell.

Add class. Performs the *addClass*$_T$ operation. When the class to add does not expand, an object for each object of the parent class is added to the instance. When the class expands to the right or down (but not in both directions), the columns and rows, respectively, in the instance that correspond to the ones in the Tabula are removed, i.e., a new expandable class does not have objects in the instance. It translates to the sequence of instance operations:

$$
\mathit{to}_{T,I}\ \{\mathit{addClass}_T\ c\} = \begin{cases} \{\mathit{addObject}_I\,(\mathit{instantiate}\,o'\ c) \mid c \lessdot c' \wedge c' \boxminus o'\} & \text{if } \mathit{expansion}\ c \equiv \mathit{None} \\ \{\mathit{delColumn}_I\, x_I \mid x_T \in [\mathrm{left}\, c, \mathrm{right}\, c] \wedge x_T \ominus x_I\} & \text{if } \mathit{expansion}\ c \equiv \mathit{Right} \\ \{\mathit{delRow}_I\, y_I \mid y_T \in [\mathrm{top}\, c, \mathrm{bottom}\, c] \wedge y_T \ominus x_I\} & \text{if } \mathit{expansion}\ c \equiv \mathit{Down} \end{cases}
$$

where *instantiate* creates an object equivalent to the given class. The delete operations must be performed from right to left or from bottom to top. Moreover, classes expanding both to the right and down are relation classes and thus cannot be added explicitly with an *addClass*$_T$ operation. The definition above omits one situation: when a non-expanding class is added and relation classes are created. In such a case, relation objects are also added.

Delete class. Performs the *delClass*$_T$ operation. It deletes all objects of the class, adds columns and rows to restore the size of the class and sets its contents.

$$
\begin{aligned}
to_{T,I}\,\{delClass_T\ c\} = {} & \{delObject_I\, o \mid c \boxminus o\} \\
& \cup \begin{cases} \{addColumn_I\ BeforeObject\, x_I \mid x_T \in [\text{left}\, c, \text{right}\, c] \wedge x_T \ominus x_I\} \\ \hfill \text{if } c \text{ is vertical} \\ \{addRow_I\ BeforeObject\, y_I \mid y_T \in [\text{top}\, c, \text{top}\, c] \wedge y_T \ominus y_I\} \\ \hfill \text{if } c \text{ is horizontal} \end{cases} \\
& \cup \{setCell_I(x_I, y_I)(defaultCell\ T_{(x_T,y_T)}) \mid (x_T, y_T) \sqsubseteq c \wedge (x_T, y_T) \ominus (x_I, y_I)\}
\end{aligned}
$$

where *defaultCell* creates an instance cell with a default value from a Tabula cell.

Rename class. Performs the *renameClass*$_T$ operation. In the instance, all objects of the renamed class are also renamed.

$$to_{T,I}\,\{renameClass_T\ \ c\ \ n\} = \{renameObject_I\ \ o\ \ n \mid c \boxminus o\}$$

Add Tabula column. Performs the *addColumn*$_T$ operation. A new column is added to each object of the class where the column is added.

$$to_{T,I}\,\{addColumn_T\ \ pos_T\ \ x_T\} = \{addColumn_I\ \ pos_I\ \ x_I \mid x'_T \ominus x_I\}$$

where

$$
\begin{cases}
pos_I = BeforeObject & \wedge \quad x'_T = \min\{x \mid x_T \leqslant x \wedge \exists x_I : x \ominus x_I\} & \text{if } pos_T \equiv BeforeClass \\
pos_I = BeforeIndex & \wedge \quad x'_T = x_T & \text{if } pos_T \equiv BeforeIndex \\
pos_I = AfterIndex & \wedge \quad x'_T = x_T & \text{if } pos_T \equiv AfterIndex \\
pos_I = AfterObject & \wedge \quad x'_T = \max\{x \mid x_T \geqslant x \wedge \exists x_I : x \ominus x_I\} & \text{if } pos_T \equiv AfterClass
\end{cases}
$$

There is no guarantee that an expanding class has objects and thus an adequate index is used to get the correct index in the instance where to add the new column.

Delete Tabula column. Performs the *delColumn*$_T$ operation. Each corresponding column in the instance is deleted.

$$to_{T,I}\ \{delColumn_T\ \ x_T\} = \{delColumn_I\ \ x_I \mid x_T \ominus x_I\}$$

Add Tabula row. Performs the *addRow*$_T$ operation. A new row is added to each object of the class where the row is added.

$$to_{T,I}\ \{addRow_T\ \ pos_T\ \ y_T\} = \{addRow_I\ \ pos_I\ \ y_I \mid y'_T \ominus y_I\}$$

where pos_I and y'_T are calculated in a similar fashion to pos_I and x'_T, respectively, for the *add Tabula column* operation.

Delete Tabula row. Performs the *delRow*$_T$ operation. Each corresponding row in the instance is deleted.

$$to_{T,I}\ \{delRow_T\ \ y_T\} = \{delRow_I\ \ y_I \mid y_T \ominus y_I\}$$

Set Tabula cell. Performs the *setCell*$_T$ operation. Each corresponding cell in the instance is set to a new default value.

$$to_{T,I}\ \{setCell_T\ (x_T, y_T)\ c\} = \{setCell_I\ \ (x_I, y_I)\ \ (defaultCell\ \ c) \mid (x_T, y_T) \ominus (x_I, y_I)\}$$

Similar operations to the ones applied on Tabulae are available to be applied on the instance. However, they require multiple operations on the instance in order to keep the conformance with model. The operations available on instances are:

- add object;
- delete object;
- add instance column;
- delete instance column;
- add instance row;
- delete instance row; and,
- set instance cell.

Add object. Performs the $addObject_I$ operation. This operation produces an instance conforming to the Tabula and thus does not require any change in the Tabula.

$$from_{T,I}\ \{addObject_I\ \ o\} = \{\}$$

Delete object. Performs the $delObject_I$ operation. This operation produces an instance conforming to the Tabula and thus does not require any change in the Tabula.

$$from_{T,I}\ \{delObject_I\ \ o\} = \{\}$$

Add instance column. Performs the $addColumn_I$ operation. This operation produces an instance not conforming to the Tabula. Two main cases exist: the column is added to an object that is the unique object of its class; the column is added to an object with other objects of its class that remain unchanged. For the first case, the solution is simple:

$$from_{T,I}\ \{addColumn_I\ \ pos_I\ \ x_I\} = \{addColumn_T\ \ pos_T\ \ x_T \mid x_T \ominus x_I\}$$

i.e., a column is added to the object's class.

For the second case, where multiple objects of the same class exist, additional changes are required in the instance in order to create a valid transformation, with multiple changes being applied to the Tabula.

$$\begin{aligned}
&from_{T,I}\ (\{renameObject_I\ \ o'\ \ n_{o1} \mid \langle o' \textit{ is a left inner object in the same parent as } o\rangle\}\\
&\qquad \cup \{renameObject_I\ \ o'\ \ n_{o2} \mid \langle o' \textit{ is a inner object of } o\rangle\}\\
&\qquad \cup \{renameObject_I\ \ o'\ \ n_{o3} \mid \langle o' \textit{ is a right inner object in the same parent as } o\rangle\}\\
&\qquad \cup \{renameObject_I\ \ o'\ \ n_{o4} \mid \langle o' \textit{ is a inner object with another parent}\rangle\}\\
&\qquad \cup \{addColumn_I\ \ pos_I x_I\})\\
&\qquad\quad = \{addColumn_T\ \ BeforeClass\ (\text{left}\,c) \mid \langle \textit{(width c) times, if any object with name } n_{o1}\rangle\}\\
&\qquad\quad\ \ \cup \{addColumn_T\ \ AfterClass\ (\text{right}\,c) \mid \langle \textit{(width c) times, if any object with name } n_{o3}\rangle\}
\end{aligned}$$

$$\cup \{\mathit{renameClass}_T\ c'\ n_{o2} \mid \langle c' \textit{ is an inner class of } c\rangle\}$$
$$\cup \{\mathit{addClass}_T\ c'_{o1} \mid \langle c'_{o1} \textit{is the copy of a class of } c \textit{ into the first set of new columns}\rangle\}$$
$$\cup \{\mathit{addClass}_T\ c'_{o3} \mid \langle c'_{o3} \textit{is the copy of a class of } c \textit{ into the second set of new columns}\rangle\}$$
$$\cup \{\mathit{setCell}_T\ (x'_T, y'_T)\ T_{(x_T,y_T)} \mid \langle \textit{for all } (x_T, y_T) \textit{ of } c \textit{ into corresponding cell of } c'_{o1}\rangle\}$$
$$\cup \{\mathit{setCell}_T\ (x'_T, y'_T)\ T_{(x_T,y_T)} \mid \langle \textit{for all } (x_T, y_T) \textit{ of } c \textit{ into corresponding cell of } c'_{o3}\rangle\}$$
$$\cup \{\mathit{setCell}_T\ (x'_T, y'_T)\ (\mathit{updateReferences}\ T_{(x'_T,y'_T)})\}$$
$$\cup \{\mathit{addColumn}_T\ \mathit{pos}_T\ x_T \mid x_T \ominus x_I \langle \textit{after all other transformations}\rangle\}$$

where n_{o1}, n_{o2} and n_{o3} are new unique names based on the ones from the original classes, and n_{o4} is equal to n_{o1} if any new object has this name, or it is equal to n_{o3} otherwise. The object o is the inner-most expandable vertical object where the operation is to be performed. The sequence of instance operations does not perform any layout nor content changes unless for the wanted operation. It renames some objects to conform to the layout changes in the Tabula. On the Tabula part, the class c such that $c \boxminus o$ is replicated. It is replicated to the left if there is any object on the left of the object o of the same class, and it is replicated to the right if there is any object on the right of the object o of the same class. Moreover, references to any attribute in the original class is converted to the union of references to the replicated attributes using the *updateReferences* function.

Delete instance column. Performs the *delColumn*$_I$ operation. The set of operations on the instance and respective operations on the Tabula follow the same pattern as for the *add instance column* operation:

$$\mathit{from}_{T,I}\,(\ldots \cup \{\mathit{delColumn}_I\ x_I\}) = \ldots \cup \{\mathit{delColumn}_T\ x_T \mid x_T \ominus x_I\}$$

Add instance row. Performs the *addRow*$_I$ operation. The set of operations on the instance and respective operations on the Tabula follow the same pattern as for the *add instance col-*

umn operation:

$$from_{T,I}\left(\ldots \cup \left\{addRow_I \ \ pos_I \ \ y_I\right\}\right) = \ldots \cup \left\{addRow_T \ \ pos_T \ \ y_T \middle| y_T \ominus y_I\right\}$$

Delete instance row. Performs the $delRow_I$ operation. The set of operations on the instance and respective operations on the Tabula follow the same pattern as for the *add instance column* operation:

$$from_{T,I}\left(\ldots \cup \left\{delRow_I \ \ y_I\right\}\right) = \ldots \cup \left\{delRow_T \ \ y_T \middle| y_T \ominus y_I\right\}$$

Set instance cell. Performs the $setCell_I$ operation. Since this instance operation can only be performed on input cells, no changes are required on the Tabula.

$$from_{T,I} \left\{setCell_I \ \ (x_I, y_I) c\right\}) = \{\}$$

This operation could also be allowed to make non-conformant changes. However, it has two drawbacks: there would be no restriction on cell modifications, and thus users could overwrite formulas or insert values of the wrong type; and an additional bidirectional transformation engine at the cell level would be needed. Ignoring these drawbacks, the implementation would follow the pattern of *add instance column* operation, but twice: one replication for the vertical classes and another for the horizontal classes.

4.4 Bidirectional Transformation Properties

Since the aim of the bidirectional transformations is to restore the conformity between instances and models, a basic requirement is that they satisfy *correctness* (Stevens, 2007) properties entailing that propagating edits on spreadsheets or on models leads to consistent states:

$$\frac{i :: t}{(to_{t,i} \ \mathrm{op}_T)\, i \ :: \ \mathrm{op}_T \, t}\ (\textit{to}\text{-correct}) \qquad \frac{i :: t}{\mathrm{op}_I \, i \ :: \ (from_{t,i} \ \mathrm{op}_I)\, t}\ (\textit{from}\text{-correct})$$

In other words, an instance i conforms to a Tabula t if $i :: t$, for a binary consistency relation $(::) \in \mathit{Instance} \leftrightarrow \mathit{Tabula}$.

As most interesting bidirectional transformation scenarios, spreadsheet instances and models are not in bijective correspondence, since multiple spreadsheet instances may conform to the same model, or vice versa. Another bidirectional property, *hippocraticness* (Stevens, 2007), postulates that transformations are not allowed to modify already consistent pairs, as defined for *from*:

$$\frac{i :: t \quad \mathrm{op}_I i :: t}{(\mathit{from}_{t,i}\ \mathrm{op}_I) = \emptyset}\ (\mathit{from}\text{-hippocratic})$$

Reading the above law, if an operation on the instance op_I preserves conformity with the existing Tabula, then $\mathit{from}_{t,i}$ op_I produces an empty sequence of operations for the Tabula. Operationally, such a property is desired in our framework. For example, if a user adds a new object of an expandable class, preserving conformity, the original Tabula is expected to be preserved because it still reflects the structure of the data. However, hippocracticness for $\mathit{to}_{t,i}$ is deemed too strong because, even if the updated model is still consistent with the old data, we still want to update the data to reflect a change in the structure, making a better match with the model (Diskin, 2008). For example, if the user adds a column to the model, the intuition is to insert also a new column in the data, even if the old data remains consistent.

Moreover, as common for incremental transformations (that translate operations rather than whole states), the application of sequential transformations naturally satisfies an *history ignorance* (Diskin, 2008) property, meaning that the application of consecutive updates does not depend on the update history.

4.5 Directed Evolution

Spreadsheets are many times required to be evolved, where the current spreadsheet must meet the requirements of a new model. In order to have the spreadsheet conforming to the new model, the legacy one can be evolved to be equal to the new model taking advantage of the coupled evolution that performs the evolution also on the spreadsheet. Thus, in the end, the legacy spreadsheet conforms to the new model.

This evolution can be performed manually, with the user performing evolution steps on the current model until it becomes equal to the new one. Another approach is to find such steps automatically. This section focuses on the latter, providing two distinct approaches based on spreadsheet differences and on bounded model checking (sections 4.5.2 and 4.5.3, respectively).

4.5.1 Evolution Sequence

The problem of directed evolution can be divided in two phases: finding a sequence of evolution steps and performing that sequence on the original model. Both approaches to the directed evolution problem are focused on the former, i.e., finding a suitable sequence of evolution steps to apply to the model. The latter consists in the composition of the evolution steps, which is a natural operation in the bidirectional transformation environment (fig. 4.2).

$$\begin{tikzcd} t \arrow[rrrrr, "\text{directed evolution}"] \arrow[dr, dashed] & & & & & t' \\ & t_1 \arrow[r, dashed] & t_2 \arrow[r, dashed] & \dots \arrow[r, dashed] & t_n \arrow[ur, dashed] & \end{tikzcd}$$

Figure 4.2: Directed evolution as the application of a sequence of operations.

Any Tabula t can be evolved into another Tabula t'. This is easily accomplished by resizing t in accordance to t', removing the classes of t, adding the classes of t', and setting all cells to the value of the corresponding cell in t'. Formally, this would give the following operations:

$$
\begin{array}{rll}
 & \{\mathit{addColumn\ AfterClass}\ (x-1) & \mid x \in [\text{right}\, t+1, \text{right}\, t']\,\} \\
\cup & \{\mathit{delColumn}\ x & \mid x \in [\text{right}\, t'+1, \text{right}\, t]\,\} \\
\cup & \{\mathit{addRow\ AfterClass}\ (y-1) & \mid y \in [\text{bottom}\, t+1, \text{bottom}\, t']\,\} \\
\cup & \{\mathit{delRow}\ y & \mid y \in [\text{bottom}\, t'+1, \text{bottom}\, t]\,\} \\
\cup & \{\mathit{delClass}\ c & \mid c \in \text{classes}\ t\ \wedge \langle \mathit{c\ is\ vertical\ or\ horizontal} \rangle\,\} \\
\cup & \{\mathit{addClass}\ c' & \mid c' \in \text{classes}\ t' \wedge \langle \mathit{c\ is\ vertical\ or\ horizontal} \rangle\,\} \\
\cup & \{\mathit{renameClass}\ c\ (\text{name}\ c') & \mid c \in \text{classes}\ t\ \wedge \langle \mathit{c\ is\ neither\ vertical\ nor\ horizontal} \rangle \\
 & & \ \ \wedge\, c' \in \text{classes}\ t'\ \wedge\ \text{range}\ c \equiv \text{range}\ c'\,\} \\
\cup & \{\mathit{setCell}\ (x,y)\ t'_{(x,y)} & \mid x \in [\text{left}\, t', \text{right}\, t']\ \wedge\ y \in [\text{top}\, t', \text{bottom}\, t']\,\}
\end{array}
$$

Although correct, this universal method to evolve a legacy Tabula into a new one has a fundamental flaw: all the data in the spreadsheet is lost when it is co-evolved. The two proposed approaches intend to minimize this issue.

4.5.2 Spreadsheet Differences

One method to prevent data loss is to evolve the Tabula without touching the parts common to both the legacy and the new Tabulae. Such a way consists in finding the differences between the two models and applying evolution steps to remove those differences.

The automatic application of this method, without an oracle identifying the common parts, can still lead to some data loss. Moreover, another issue arises, namely the use of old data that should have been deleted. An example of this issue consists in the renaming of a class instead of deleting the original class and adding a new one. In the former situation, all the data of that class is kept, whilst in the latter the data is deleted and default values are set when the new class is added.

This method follows the universal method presented in the previous section coupled with an algorithm to find spreadsheet differences. Moreover, when parts of the Tabula detected as deleted are equal to other parts detected as added, it is assumed those parts were moved.

This moving translates to operations on the spreadsheets, where data is copied from the old positions to the new ones. Overall, the steps are:

i. identify grid operations;

ii. identify class operations;

iii. identify cell editions;

iv. move detection;

v. copy spreadsheet data.

Grid operations. The operations included in this step are $delColumn_T$, $delRow_T$, $addColumn_T$ and $addRow_T$. The set of operations to be applied to the legacy Tabula is obtained by finding a core grid, which is common to both the legacy and the new Tabulae, i.e., without any of the differences between the Tabulae.

In order to find this core grid, the RowColAlign algorithm (Harutyunyan, Borradaile, Chambers, & Scaffidi, 2012) is used. The objective is to find the maximal sub-sequence of rows and columns that both grids have in common, also known as the Longest Common Subsequence (LCS) problem. RowColAlign provides an alignment that is the basis to infer the operations needed to evolve the legacy model into the new one. Moreover, this alignment is not strict, allowing for a larger common set of columns and rows in exchange for a small set of different cells.

The end result of this step is a sequence of $delColumn_T$ and $delRow_T$ operations to convert the legacy Tabula's grid into the common grid followed by another sequence of $addColumn_T$ and $addRow_T$ operations to convert the common grid into the new Tabula's grid. This sequence of operations, applied to the legacy Tabula, results in a Tabula with the same width and height of the new Tabula, possibly with different cells and classes.

Class operations. The information about classes is not taken into account when devising the grid operations. Thus, the range of the classes in the legacy Tabula might not match the range of the classes in the new Tabula.

In order to diminish the effects of deleting no longer existing classes (either due to renaming of repositioning), a first pass on the grid operations is performed. This pass adjusts the arguments to the *add* operations, setting the relative position to *BeforeClass*, *BeforeIndex*, *AfterIndex* or *AfterClass*, and modifying the index accordingly.

Then, classes in the legacy Tabula are matched by their range with classes in the new Tabula. If a class in the legacy Tabula is not matched, is is deleted. If a class in the new Tabula has no match, it is added. For all other classes, if the names of the matched classes are not the same, the legacy Tabula class is renamed to the name of the new Tabula class.

Cell editions. Applying the grid and class operations, each cell in this evolved Tabula will correspond to the cell of the new Tabula at the same position. Iterating over each position and comparing both cells gives a set of cells that do not have the value they should have. For these cells, their contents is set to the content of the cell in the new Tabula. Reducing the number of $setCell_T$ operations also reduces the loss of data.

Move detection. It is possible to prevent loss of data due to repositioning of the rows or columns by detecting such "moves". The basic idea is find pairs of removal and insertion (e.g., $delColumn_T$ followed by an $addColumn_T$ and multiple $setCell_T$ replicating the column contents). However, finding moves in the Tabula can be ambiguous: an attribute was deleted from a class and added to another, was this a move? In some situations a move is desirable whilst in others not. Many more examples could be provided, where the correct choice is difficult to make without further information. Hence, only column and row moves are taken into account in this work.

In order to find column moves, a set of deleted columns from the legacy Tabula and a set of added columns to common grid are obtained. Basically, the first set is given by the indices of the $delColumn_T$ operations whilst the second is given by the $addColumn_T$ arguments. Each deleted column is compared with each added column, but taking into account only the rows in the common grid. The contents of the former set come from the legacy Tabula whilst the contents of the latter come from the new Tabula, thus removing the need to take into account *setCell*$_T$ operations. When a pair of columns is equal, then it is assumed a move was performed. To find row moves, the process is analogous.

Spreadsheet data copy. The data corresponding to moved cells needs to be copied in order to not be lost. Before performing the co-evolution of the operations inferred in the previous steps, the data that should be moved is copied. Only the data in the matched cells in the columns and row comparisons is used. After performing the co-evolution of the operations, the data is inserted into their respective cells. When the number of elements copied is different from the number of elements to be pasted (e.g., due to a move from a class to another), two situations are possible:

- more elements where copied than the ones are needed;
- more elements needed than the ones copied.

In the former case, a prefix of the elements is kept, whilst in the latter additional values are left at their default value.

4.5.3 Bounded Model Checking

Another method to find evolution steps between two Tabulae is using a solver to perform bounded model checking (Biere, Cimatti, Clarke, Strichman, & Zhu, 2003). It follows work on bounded model checking of temporal formulas (A. Cunha, 2014) and model repair (Macedo, Jorge, & Cunha, 2017), which this problem can be seen as. Model repair consists in finding a new model that is consistent with some property. In this case, the property is a new Tabula. Some approaches to solve the model repair problem also rely on solvers.

This method uses a description of each Tabula, a description of the possible evolution steps and a formula stating there is no way from the legacy Tabula to the new one using the given possible steps. Moreover, limits are imposed on the elements present in the description (i.e., they are bounded), preventing an explosion of states during the solving. Two results are possible:

- the formula is true and thus there is no way between both Tabulae with the given constraints; or,

- the formula is false and at least one counter-example is given, i.e., a set of steps that transform the legacy Tabula into the new one.

Conceptually, this method is simpler than the one based on spreadsheet differences as the effort is transferred to an external solver. It is also more powerful since multiples options can be given to the user in order to choose the one more suited to the situation and weights can be given to implement heuristics. These heuristics would allow to show the user "better" operation sequences first (e.g., based on prevention of data loss).

The implementation of this approach (section 5.2.2) is based in the relational language Alloy (Jackson, 2002, 2012). It requires thus to express Tabulae in a relational form, along with the Tabula operations. Well-formedness of Tabulae does not need to be specified, as the given legacy and new Tabulae must already be well-formed, and applying the operations must produce well-formed Tabulae. Moreover, contents of cells and names are not required.

Tabula representation. In order to represent a Tabula, the following types are used:

- Tabula
- Row
- Cell
- Class
- Column
- Expansion

Using these types, a Tabula is defined by two relations:

$$\begin{aligned} &\mathit{classes} &&: \mathit{Class} \rightarrow \mathit{Tabula} \\ &\mathit{content} &&: \mathit{Row} \rightarrow \mathit{Column} \rightarrow \mathit{Cell} \rightarrow \mathit{Tabula} \end{aligned}$$

A class is defined by four relations:

$$\begin{aligned}
&\mathit{left} &&: \mathit{Class} \to \mathit{Column} \to \mathit{Tabula}\\
&\mathit{right} &&: \mathit{Class} \to \mathit{Column} \to \mathit{Tabula}\\
&\mathit{top} &&: \mathit{Class} \to \mathit{Row} \to \mathit{Tabula}\\
&\mathit{bottom} &&: \mathit{Class} \to \mathit{Row} \to \mathit{Tabula}\\
&\mathit{expansion} &&: \mathit{Class} \to \mathit{Expansion} \to \mathit{Tabula}
\end{aligned}$$

where *Expansion* can be one of *None*, *Right*, *Down* or *Both*. Moreover, for each row and column, there is a relation *next* to order them:

$$\begin{aligned}
&\mathit{next} : \mathit{Row} \to \mathit{Row} \to \mathit{Tabula}\\
&\mathit{next} : \mathit{Column} \to \mathit{Column} \to \mathit{Tabula}
\end{aligned}$$

Operation representation. Each operation is represented by a corresponding type which is ultimately a subtype of *Operation*. It is the *Operation* type that imposes an order on the application of operations, with two relations:

$$\begin{aligned}
&\mathit{pre} &&: \mathit{Event} \to \mathit{Tabula}\\
&\mathit{post} &&: \mathit{Event} \to \mathit{Tabula}
\end{aligned}$$

to identify the Tabula before the operation and the Tabula after the operation, respectively. Moreover, there is a *RowOperation* type specifically for rows to identify on which row the operation is performed, and a *ColumnOperation* type for columns to identify on which column the operation is performed.

There is a type for each Tabulae operation and argument not represented by one of the available types, e.g., there is the type *AddColumnBeforeIndex* for the $\mathit{addColumn}_T$ operation with its RelPos_T argument being *BeforeIndex*. Moreover, constraints are applied to each of these types to identify the changes from the *pre* Tabula to the *post* one.

Operation sequence. The result from executing the solver consists in an operation sequence. To achieve that, two concrete subtypes of *Tabula* are used: *Source* and *Target*. The *Source* is used to identify the legacy Tabula and it is in the *pre* relation of the first operation. The *Target* is used to identify the new Tabula and it is in the *post* relation of the last operation. Any consistent representation of the environment is a valid operation sequence that can be used to evolve the legacy Tabula into the new one.

Chapter 5

A Concrete Implementation

The Tabula language, its instances and their evolution were implemented in a library, allowing reuse in other software artefacts. In order to evaluate the use of model-driven spreadsheet development with Tabula, an extension to LibreOffice Calc was created using the library, dubbed MDSheet.

The implementation first started as an enhancement to the HaExcel framework (J. Cunha, Saraiva, & Visser, 2009), a framework to manipulate, transform and query spreadsheets. It was implemented to support ClassSheet models and transformations on them, with the automatic co-evolution of their spreadsheets (J. Cunha et al., 2011; Mendes, 2011). The advantages of this model-based environment were immediately recognized when it was first presented to the scientific community at the 2011 VL/HCC. However, questions also arose, namely what happened when changes where performed in the instance, and people wanted to make changes there. This showed the value of the presented work, the value of having a functional tool and the feedback of its users, but also next steps to research.

As work on the implementation advanced, new features were added thus augmenting complexity of the extension. It was decided then to separate the code, with a library for the implementation of the developed theory and an extension for LibreOffice Calc to experiment on it with end users. Both of these artefacts are described below.

5.1 Core Library

The Tabula core library is implemented in Haskell. It describes data types for Tabulae and respective instances, functions to evaluate the correctness of the values of those types and functions to evolve them, amongst other auxiliary functions.

Thus, data types exist for Tabulae and instances:

```
data Tabula   = Tabula   Name [Class]  (Grid TCell)
data Instance = Instance Name [Object] (Grid Cell)
```

translating their definitions from chapter 3, and using auxiliary types for names, classes, objects, grids and cells. Therefore, the use of the library is similar to the use of their definitions, with the Tabula from example 1 being implemented as:

```
Tabula "ItemList"
    [Class "Items" ((0,0),(1,2))   None,
     Class "Item"  ((0,1),(1,1))   Down]
    Grid.fromLists [["Items",      ""]
                   ,["desc=\"\"","value=0"]
                   ,["Total",      "total=SUM(Item.value)"]]
```

Name, TCell, Cell being instances of *IsString* and using the `OverloadedStrings` language extension.

Then, there are the following functions to validate Tabulae, instances and the conformity of the instance with its Tabula:

```
checkTabula   :: Tabula → Bool
checkInstance :: Instance → Bool
models        :: Tabula → Instance → Bool
```

With a pair of well-formed Tabula and conforming instance, the following functions can be used to evolved either of these artefacts:

```
type Evolution = (Tabula, Instance) → (Tabula, Instance)
  -- Tabula evolution and instance co-evolution
addClass_T   :: Class → Evolution                -- add class
delClass_T   :: Class → Evolution                -- delete class
addColumn_T  :: RelPos → Int → Evolution         -- add column
delColumn_T  :: Int → Evolution                  -- delete column
addRow_T     :: RelPos → Int → Evolution         -- add row
delRow_T     :: Int → Evolution                  -- delete row
setCell_T    :: Int → Int → TCell → Evolution    -- set cell
  -- instance evolution and Tabula co-evolution
addClass_I   :: Class → Int → Evolution          -- add an object
delClass_I   :: Class → Int → Evolution          -- delete an object
addColumn_I  :: RelPos → Int → Evolution         -- add column
delColumn_I  :: Int → Evolution                  -- delete column
addRow_I     :: RelPos → Int → Evolution         -- add row
delRow_I     :: Int → Evolution                  -- delete row
setCell_I    :: Int → Int → Cell → Evolution     -- set cell
```

The implementation also includes functions to export Tabulae and instances to HTML for easy visualization without the use of a spreadsheet system, to import a plain spreadsheet into a Tabula instance, and other auxiliary functions to be used by external tools (e.g., for querying data).

5.2 Directed Evolution

The directed evolution was two distinct implementations, reflecting the two approaches undertaken. The next two sections present how each of them is implemented.

5.2.1 Spreadsheet Differences

The spreadsheet differences are based on the RowColAlign algorithm (Harutyunyan et al., 2012). An implementation of this algorithm was already available, but it was implemented in MATLAB (MathWorks). This MATLAB implementation deals with spreadsheets as matrices and has auxiliary functions to convert CSV files into such matrices, with each cell being hashed for easier comparison of their contents.

The MATLAB implementation was translated to Haskell to be included with the Tabula core library. Moreover, since the RowColAlign algorithm only provides alignments, a few modifications were made in order to work with Tabulae and to return operations. The main function to use the algorithm is:

$$rowColAlign :: Tabula \rightarrow Tabula \rightarrow (Operation_T, [[Integer]], [[Integer]])$$

which, from a legacy Tabula and a new one, returns a list of operations, to evolve the grid of the former into the grid of the latter, and two alignment tables between those two Tabulae.

The operations resulting from *rowColAlign* are refactored in order to take into account the classes of the Tabulae, to which operations to rename, delete and add classes are appended. The operations, with the help of the two alignments, are then used to detect column and row moves.

Finally, the old data is copied, the operations are applied to the pair of Tabula and instance, and the data is pasted into the correct place.

5.2.2 Bounded Model Checking

The solver used to infer the steps for the directed evolution is KodKod (Torlak & Dennis, 2006), a SAT-based constraint solver. This solver was chosen for several reasons, such as it provides an API, it has a tight relation with Alloy, a language the author of this thesis is already familiar with, and there were knowledgeable individuals in the same research centre.

The directed evolution was first implemented in Alloy, creating a concise and easily readable description of the relations and the universe of possible atoms. From the Allow description, and using sample models and evolution steps, Java code was generated. This code uses the KodKod API and it is specific for the given samples, having 4844 lines of code for two sample models with three columns and two rows and one $setCell_T$ operation. Moreover, in order to be compiled, the generated code requires refactoring so that it becomes syntactically correct. The specific parts of the code are then identified and generalised, resulting in code with half the number of lines of codes. The generalised code uses the Java representation of the involved artefacts, with Java classes for cells, classes, Tabulae and operations. Furthermore, restrictions described in Alloy were transposed to Java code, slightly optimizing the final code.

The steps to perform the directed evolution is found by creating two Tabulae, representing the legacy and new ones, and then executing the solver. It is also possible to set an upper bound for the number of operations expected to be obtained. This is exemplified by the following sample Java code:

```
Tabula t0 = new Tabula( ... );
Tabula t1 = new Tabula( ... );
DirectedEvolution steps = new DirectedEvolution(t0, t1);
steps.setSteps(5);
steps.solve();
```

The result of the solver is a `Solution` Java object which, if a solution is found, contains the atoms of the solution found. From these atoms, the list of operations is extracted.

Although at a higher level than difference-based step finding, bounded model checking, and specifically the implemented approach, has important drawbacks:

- it may not find a solution within the defined number of steps;
- it is slower for larger inputs; and,
- it accepts only “small” inputs.

When trying to find steps for real-world spreadsheets, the relations require a large universe which KodKod cannot technically handle[1]. This situation was obtained with a real-world spreadsheet, where the legacy Tabula has 102 columns, 2 rows and 1 class, and the new Tab-

[1]KodKod returns the error `kodkod.engine.CapacityExceededException`.

ula has 105 columns, 2 rows and 1 class. Therefore, further work using this method was not pursued, including finding move operations.

5.3 LibreOffice Calc Extension

LibreOffice has a powerful modular extension model, allowing to install external components (extensions) without changes to the LibreOffice code. These components define services that can be used by LibreOffice, but also by other components, using the Unified Network Objects framework (fig. 5.1).

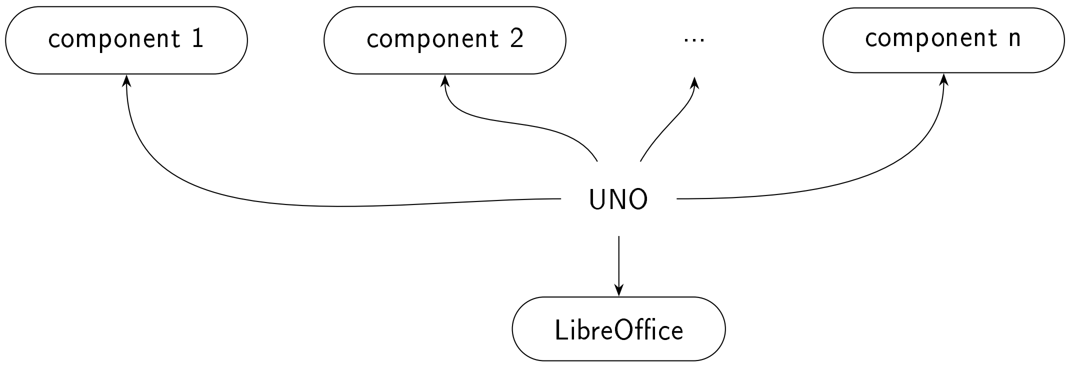


Figure 5.1: Overview of the LibreOffice extension model using components.

The extension for LibreOffice Calc was developed using the Tabula core library with additional code to connect the user interface with the corresponding actions in the library. It was developed in C/C++, Haskell and LibreOffice Basic.

Thus, Haskell code defines the main behaviour of the extension and the simplification of some data structures to be marshalled to C structures. Around this code, there is C code to invoke the Haskell one as Haskell's Foreign Function Interface (FFI) does not support C++. Finally, the C++ code call the C one, but also interacts with some of the LibreOffice features, implementing callbacks for some actions and dealing with the objects required for the embedding (e.g., named ranges and buttons in the instance sheet).

Finally, in the user interface, a toolbar is provided with buttons to perform the evolution steps and other functions from the library (see fig. 5.3). When one of those buttons is clicked,

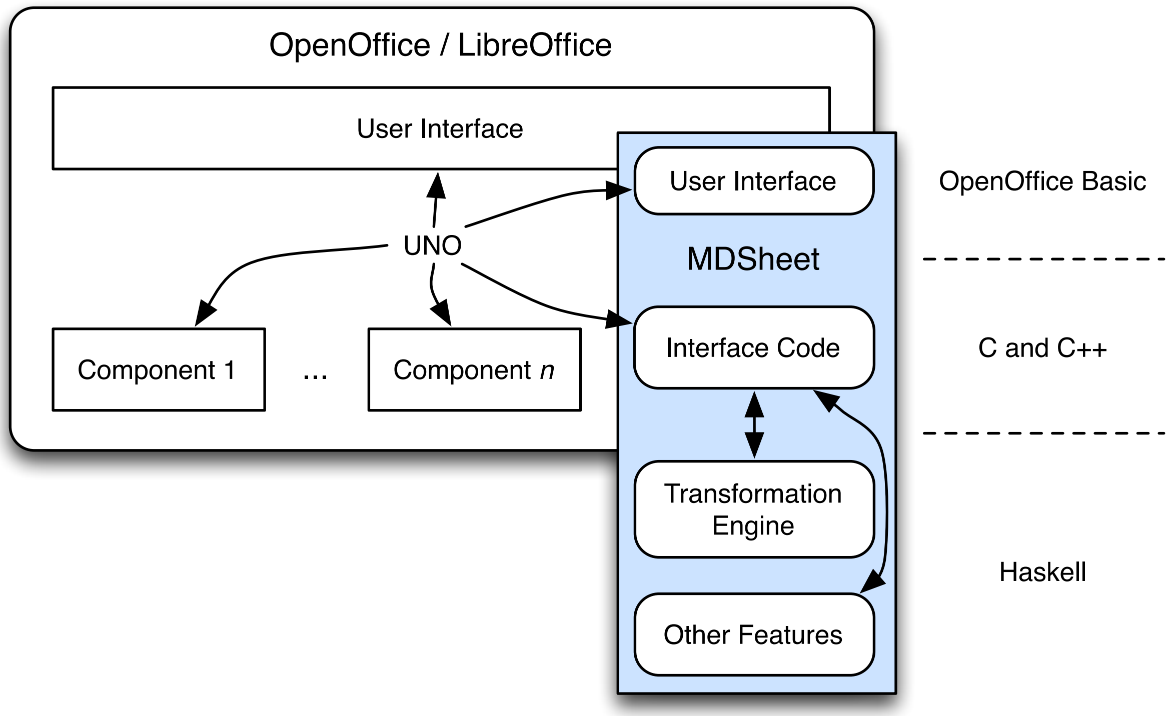


Figure 5.2: Overview of the MDSheet architecture within the LibreOffice environment.

a routine defined in Basic is invoked. Other user interactions also invoke actions, e.g., editing cells in the Tabula sheet and clicking in buttons in the instance sheet to add new objects.

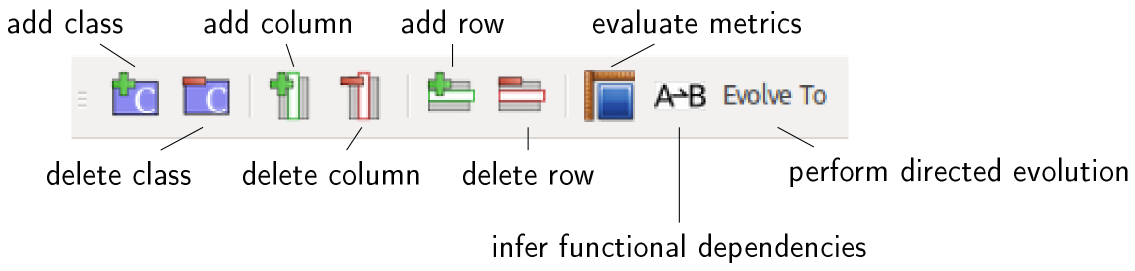


Figure 5.3: MDSheet extension toolbar.

LibreOffice Basic and C/C++. Most of the interface with the user and between LibreOffice Calc and the MDSheet extension is programmed in LibreOffice Basic. Actions to be performed by the extension are triggered by user actions in LibreOffice's user interface, invoking a Basic routine that gathers the necessary data and then invokes the appropriate code from the extension. This last code is written in C++ but, since both LibreOffice Basic and C++ have UNO bindings, the interaction between the two is transparent with the conversion between the different representations of a data type being performed automatically.

C/C++ and Haskell. C++ code is not directly compatible with Haskell, thus the C++ parts that need to interact with Haskell code have C wrappers. The C wrappers for functions are compatible with the Haskell FFI and the wrappers for C++ objects are used as black boxes, without direct access from Haskell code. Moreover, some types are used at every level of this stack:

- integers;
- real numbers; and,
- text.

Integers and real numbers are converted without difficulties between the different layers. Text is a bit more complex since LibreOffice uses Unicode text in order to support a large set of languages and symbols. In Basic, text is identified with the type `String`, in UNO IDL it is identified with the type `string` and in C++ it is identified with the type `OUString`. All three of these types are equivalent as they are integrated in the UNO framework. In Haskell, the type used to store Unicode is `Text` from the `Data.Text` module of the `text` Haskell package. The convertion between C++ and Haskell takes advantage of the fact that both C++ and Haskell types use the same format in their structures. Thus, using the FFI from the `Data.Text` Haskell module, marshalling of the structures is made with little additional complexity.

5.4 Google Sheets Add-On

Google Sheets is an online spreadsheet host system with a web-based interface and applications for mobile devices. It provides a basic functional spreadsheet system with additional features (e.g., formatting and pivot tables). Furthermore, similarly to extensions in LibreOffice, Google Sheets supports add-ons that allow developers to provide new features. Therefore, Google Sheets was also targeted as a host for embedding spreadsheet models using the MDSheet framework.

The approach used consists in providing a web service with MDSheet's functionalities that is used by an add-on in Google Sheets. Since there is no framework to unite both ends, unlike the UNO framework in LibreOffice, an intermediate data representation is used, namely JavaScript Object Notation (JSON). The architecture of this version of MDSheet, pictured in fig. 5.4, consists in an add-on to be installed on the client side, as a Google Sheets add-on, that communicates with a server providing the bidirectional transformation engine used in the MDSheet extension. The add-on is programmed in Javascript, using the Google Sheets API[2], whilst the web service server is written in Haskell.

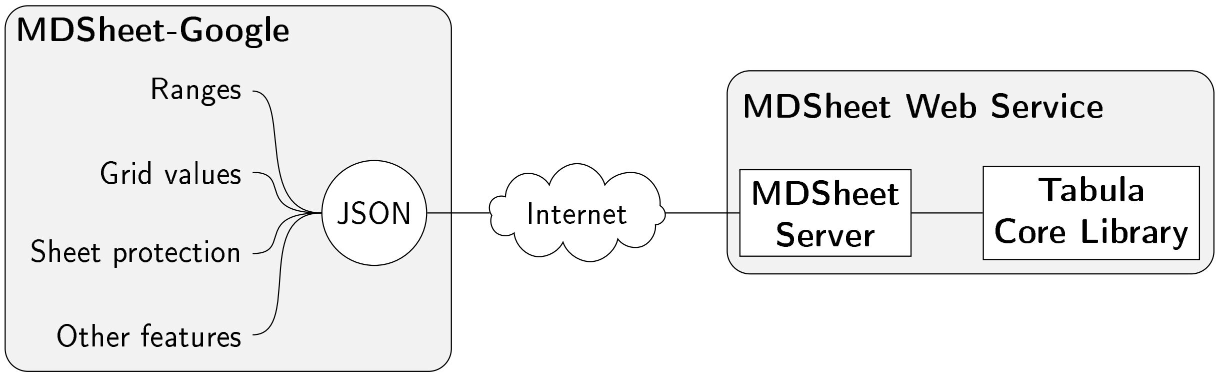


Figure 5.4: Architecture of the MDSheet-Google system.

Unfortunately, at the time of development of this version of MDSheet, the Google Sheets API lacked in some fundamental features. One of them was the ability to manipulate named ranges (it was only possible to create them). This issue was worked around by adding an additional sheet to keep the required data.

Moreover, the architecture used added too much latency and was not suitable for a production environment.

[2]`https://developers.google.com/sheets/api/`

Chapter 6

Empirical Evaluation

User interaction is an important part of spreadsheet development. In order to assess the impact of this work, two aspects were evaluated: the usage of model-driven spreadsheets (section 6.1) and the understanding of the bidirectional evolution environment (section 6.2).

Both these studies focus on the evolution and editing of existing model-driven spreadsheets by end users. The design of the language and the creation of new model-driven spreadsheets were not considered in these evaluations. In both cases, it is assumed a spreadsheet modelling expert develops the initial Tabula.

6.1 Model-Driven Spreadsheets[1]

In the context of software engineering research, empirical validation is widely recognized as essential in order to assess the validity of newly proposed techniques or methodologies (Wohlin et al., 2012). While our previous work on the construction of a model-driven spreadsheet development environment has received good feedback from the research community, the fact is that its assessment in a realistic and practical context was still lacking. In this line, we have designed an empirical study that we describe in this section and whose results we also analyze

[1]Also published in section "empirical evaluation" of: "Embedding, Evolution, and Validation of Model-Driven Spreadsheets", J. Cunha, J. P. Fernandes, J. Mendes, and J. Saraiva. *IEEE Transactions on Software Engineering* (Volume: 41, Issue: 3, March 1 2015), © 2015 IEEE

in detail here.

The experiment that we envisioned is motivated by the need to understand the differences in individual performance that users achieve under MDSheet (using what we call *model-driven* spreadsheets) against traditional spreadsheet systems (from now on termed *plain*). The perspective of the experiment is from the point of view of a researcher who would like to know whether there is a systematic difference in the users' performance.

In this section we detail the different stages that we underwent in preparing and designing our study (section 6.1.1), in running it (section 6.1.2), and in analyzing (section 6.1.3) and interpreting (section 6.1.4) the results, which are discussed afterwards (section 6.1.5). Finally, we can summarize the scope of this study as suggested in (Wohlin et al., 2012) as follows:

Analyze *the spreadsheet development process*
for the purpose of *evaluation*
with respect to its *effectiveness and efficiency*
from the point of view of the *researcher*
in the context of *the usage of two different spreadsheets by Master students.*

6.1.1 Design

The goal of our study is to analyze several aspects of spreadsheet development, and to evaluate the implications of using a model-driven approach against the more commonly used approach of designing and introducing spreadsheet data from scratch and immediately within spreadsheets themselves.

In particular, we want to evaluate the *effectiveness* and *efficiency* of using MDSheet. As we described earlier, it is quite common to find errors in spreadsheets. As one of the objectives of our approach is to improve this scenario, to evaluate its effectiveness was important. However, it could be the case that users would make less mistakes, but were (much more) slower. This would decrease the practical interest of our approach. Thus, we also want to evaluate its efficiency.

The study that we conducted was designed for a controlled environment mostly because our tool was never tested in production. In order to achieve this controlled environment, we decided to perform the study in an off-line setting (in an academic environment and not in industry), and with university students attending a Master's program. Furthermore, our study analyzes the specific use of ClassSheet-based models (a subset of the Tabula language equivalent to ClassSheets), and does not consider generic model-driven spreadsheet development. Finally, in our study, participants were asked to solve realistic problems, in situations that were closely adapted from real-world situations.

Hypotheses

MDSheet uses models to specify spreadsheets, hence it benefits from advantages such as: (i) users are freed from the risks associated with editing formulas directly, and (ii) users do not have to manually identify parts of the spreadsheet that are repeatable (class expansions). In theory, (i) reduces the number of errors and (ii) improves spreadsheet development performance. However, this needs to be tested. Thus, we can informally state two hypotheses:

1. In order to perform a given set of tasks, users spend less time when using model-driven spreadsheets instead of plain ones.

2. Spreadsheets developed in the model-driven environment contain less errors than plain ones.

Formally, two hypotheses are being tested: H_T for the time that is needed to perform a given set of tasks, and H_E for the error rate found in different types of spreadsheets. They are respectively formulated as follows:

1. *Null hypothesis,* H_{T_0}: The time to perform a given set of tasks using MDSheet is not less than that taken with plain spreadsheets. $H_{T_0} : \mu_d \leq 0$, where μ_d is the expected mean of the time differences.

   Alternative hypothesis, $H_{T_1} : \mu_d > 0$, i.e., the time to perform a given set of tasks using MDSheet is less than with plain spreadsheets.

   Measures needed: time taken to perform the tasks.

2. *Null hypothesis*, H_{E_0}: The error rate in spreadsheets when using MDSheet is not smaller than with plain spreadsheets. $H_{E_0} : \mu_d \leqslant 0$, where μ_d is the expected mean of the differences of the error rates.

   Alternative hypothesis, $H_{E_1} : \mu_d > 0$, i.e., the error rate when using MDSheet is smaller than with plain spreadsheets.

   Measures needed: error rate for each spreadsheet.

Variables

The independent variables are: for H_T the *time to perform the tasks*, and for H_E the *error rate*.

Subjects and Objects

The subjects for this study were first year Master students undergoing a course at the University of Minho. Out of a total number of thirty-five students that were invited, twenty-five actually accepted the invitation and participated in our study. More details about the subjects participating in the study are presented in Section 6.1.3.

The objects for the study consisted in three different kinds of spreadsheets that are described later in this paper, in Section 6.1.1. One spreadsheet was used to support an in-study tutorial that was given to participants before they were actually asked to perform a series of tasks on the model-driven versions of the remaining two spreadsheets. This design choice attempts to minimize the threats to construct validity, namely the mono-operation bias (please refer to Section 6.1.4 for more details on threats to validity).

Design

In our study, we followed a standard design with one factor and two treatments, as presented in (Wohlin et al., 2012). The factor is *the development method*, that is, spreadsheet development without using MDSheet. The treatments are *plain* and *model-driven*. The dependent variables are measurable in a ratio scale, and hence a parametric test is suitable.

Moreover, blocking is provided in the sense that each hypothesis is tested independently for each object. This reduces the impact of the differences between the two spreadsheets.

Instrumentation

As we have been describing, our study was supported by three distinct kinds of spreadsheets. For the spreadsheet that was used in the tutorial, we have only constructed its model-driven version, but for the two remaining spreadsheets we have used both their model-driven and plain versions.

The spreadsheet that was used in the tutorial was designed to record (simplified) information on the flights of an airline company, namely its planes, their crew and the meals available on-board. As for the two remaining kinds of spreadsheets, they were selected based on their practical interest: one of them, from now on termed *budget*, is an adapted version of the *Personal budget worksheet* that is available from Microsoft Office's webpage[2] (this spreadsheet has been downloaded over 4 million times); the other, *payments* is an adapted version of a spreadsheet that is being used to register the entire information regarding the payments that occur in the municipal utilities company *Agere*[3] that is responsible for the water supply in the city of Braga, Portugal. In order to be usable, we have reduced the size of both spreadsheets, without changing their complexity.

The given budget spreadsheet had 66 rows and 80 columns with 3306 filled cells. It contained the information for 12 months of 6 consecutive years, organized in 11 different categories of income/expenses subdivided in income/expense items. Each year and category had a subtotal, and there was a grand total for the years and another for the categories, all of them being formulas.

The given payments spreadsheet had 33 rows and 55 columns with 1645 filled cells. It contained the information for 3 years, subdivided in 9 payment forms with 6 kinds of totals, and 1 month (January) with 31 days. At the end of each month/year there was also a grand total.

[2]http://office.microsoft.com/en-us/templates/personal-budget-worksheet-TC006206279.aspx
[3]http://www.agere.pt/

Guidelines were also provided to participants: they consisted of the list of tasks to be performed. For both spreadsheets, three tasks were given (non-essential data is omitted, being replaced by "[...]"):

Budget

i) "Add to the budget two new categories of expenses, [...], with the following expenses [...]."

ii) "Add a new year, [...], to the budget keeping the structure of the spreadsheet, and insert the following data: [...]."

iii) "Delete the information from categories [...]."

Payments

i) "Add a new month, [...], maintaining the structure of the spreadsheet and add the following data: [...]."

ii) "Add a new year, [...], keeping the structure of the spreadsheet and insert the following data: [...]."

iii) "Change the spreadsheet in order to remove the information related to kind of payment [...], removing the corresponding column."

Several versions of the lists of tasks were prepared, where each version had a specific task order, thus each participant performed the tasks sequentially in a random order. This was done to avoid learning effects on the tasks, as also done in other studies (for instance in (Henley & Fleming, 2014)). The data to be inserted did not contain more than six values per task. The participants had to update the formulas to include new references to cells when needed. The full description of the six tasks, that include the data to be added to the spreadsheets, is available at the tool webpage presented earlier.

To evaluate the participants' work, each task was sub-divided in small parts equivalent to each "atomic" operation that they should perform. This division into several operations allows to create a better profile of spreadsheet errors, enabling us to see where users make mistakes: insertion of cell for the new data, input of data, update of formulas. They are precise enough but without introducing unnecessary complexity to the analysis of the results. With this division we intend to analyze common errors that users make, e.g., input of wrong values, oblivion to update formulas, and (implicit) layout changes.

For the budget spreadsheet, we have the following sub-division, with a total of twenty-three items:

- ten items for i), corresponding to inserting the lines for the data, inserting the values, and updating the formulas;
- nine items for ii), corresponding to inserting the new cells, inserting the data, inserting the formulas, and updating the formulas for the totals;
- four items for iii), corresponding to removing the categories, and updating the formulas.

For the payments spreadsheet, we have the following sub-division, with a total of ten items:

- four items for i), corresponding to inserting new cells for the data, inserting the data, and updating the formulas;
- four items for ii), corresponding to inserting new cells for the data, inserting the data, and updating the formulas;
- two items for iii), corresponding to deleting the columns and updating the formulas.

In order to understand the background of the subjects and the difficulties they experienced when participating in the study, two questionnaires were prepared: one answered before the study itself (pre-questionnaire) and another after (post-questionnaire).

The data collected consists of the modified spreadsheets by the participants, and some information about the performance of our model-driven environment. For that, the MDSheet add-on was modified in order to provide a log of the user actions when working with the model-driven spreadsheet. This log contains the action performed (e.g., "add instance" and "remove class"), and how much time the system took for each action.

Data Collection Procedure

Several steps were planned to run the study, with two distinct options in the order they should be performed. One of them is:

1. Filling the pre-questionnaire.
2. Performing the sets of tasks on the two plain spreadsheets, with a time limit of fifteen minutes.
3. Attending the tutorial on MDSheet.
4. Performing the sets of tasks on the two model-driven spreadsheets, with a time limit of fifteen minutes.
5. Filling the post-questionnaire.
6. Collecting all spreadsheets, questionnaires and logs.

The other option is: (1), (3), (4), (2), (5), and (6). This option consists of first performing the operations with the model-driven environment and then the operations on plain spreadsheets, as opposed to the first option in which the operations on plain spreadsheets is first. This design choice attempts to minimize the learning effects between treatments and so the validity threats. This is a common technique used in other studies, for instance in (Henley & Fleming, 2014).

In steps (3) and (6), our team was expected to have a direct participation, giving the tutorial in step (3) and retrieving, in step (6), all the artifacts used or created by the subjects.

All the subjects of our study were expected to perform the two sets of tasks on the respective spreadsheets. The goal of the study is not to compare one spreadsheet against another, but instead to compare two methods to develop spreadsheets. Furthermore, using two spreadsheets permits to reduce the mono-operation bias (see Section 6.1.4).

Analysis Procedure and Evaluation of Validity

The analysis of the collected data is achieved performing paired tests where the performance of each subject on the plain version of the spreadsheet is tested against the model-driven version. For this, the following tests are available: paired t-test, Wilcoxon sign rank test, and the dependent-samples sign-test.

Since the study is composed of several tasks, and each participant may not complete all of them, participants that do not complete all the tasks for both treatments of a spreadsheet will be discarded from the global analysis of that spreadsheet. This will allow us to compare the results of the remaining participants against each other, and not doing so could lead to incorrect illations given that a concrete task may be more error-prone than the others.

To ensure the validity of the data collected, several kinds of support were planned: constant availability to clarify any doubt, tutorial to teach the model-driven development process, and slightly supervise the work done by the subjects in a way that do not interfere with their work. This last point consists of navigating through the room and see which subjects look like they are having problems and try to help them if it is about something that does not influence the results of the study.

6.1.2 Execution

The study was performed in two classrooms with twenty-five university students, eleven in one room and fourteen in the other one. The participants were randomly assigned to each room. All performed the study at the same time, but with different execution orders depending on the room that they were in.

The participants first started filling the pre-questionnaire, with generic information about themselves (gender, age range, and undergraduate major). They also answered some questions so we could assess their previous experience with spreadsheets.

While they filled the questionnaire, we checked if their environment was correctly set. This environment consisted in a virtual machine with Lubuntu 11.10 as the installed operating system, where we pre-installed LibreOffice Calc, and our add-on – MDSheet.

The list of tasks to be performed was then distributed amongst the participants, and they had fifteen minutes to perform all the tasks on the spreadsheets, but without the assistance of our framework.

Before telling the participants to perform the tasks on the spreadsheets with the models, they attended the tutorial that we prepared on how to use the MDSheet framework. During the tutorial, we answered all the questions that the participants had, making sure that they could use the framework.

The subjects in one room first performed the tasks with plain spreadsheets, then attended the tutorial and then performed the tasks on model-driven spreadsheets. The subjects in the other room first attended the tutorial, then performed the tasks on model-driven spreadsheets, and finally performed the tasks on plain spreadsheets.

At the end, we gave the post-questionnaire to the participants to evaluate the confidence that they had on their performance during the study. Finally, we collected the modified spreadsheet files, so that we could analyze them later on.

6.1.3 Analysis

The global quantitative analyses were performed with results of the subjects that completed all the tasks for any of the given spreadsheets as described in Section 6.1.1. This results in 15 subjects for the budget spreadsheet, and 12/11 for the payments one (for time/error analysis, respectively, as participant 3 only submitted the times taken in each task, but not the resulting spreadsheets themselves).

Descriptive Statistics

Subjects: Basic information about the subjects was gathered, namely their gender, age, studies' background, and familiarity with spreadsheets. From the twenty-five subjects, twenty-one are male and four are female. Most of them are aged between twenty and twenty-eight, with two subject being over thirty-one. The subjects come from different areas, most of them having a background in *informatics engineering* or *computer science*, but others come from *information technology and communication*, *IT for health*, or *IT management*. Two of the subjects never worked with spreadsheets previously and the levels of experience vary from having used at least once to an heavy usage.

Time spent: As expected, differences were found in the time that subjects used to perform the tasks. The minimum times recorded on each spreadsheet were by participants using the model-driven environment, with average times being lesser for the model-driven approach.

Figure 6.1 shows the time each participant took to achieve the given tasks, both with and without the model-driven environment, for the budget spreadsheet. Only the results for the subjects that performed all the tasks are displayed to allow for an easier comparison.

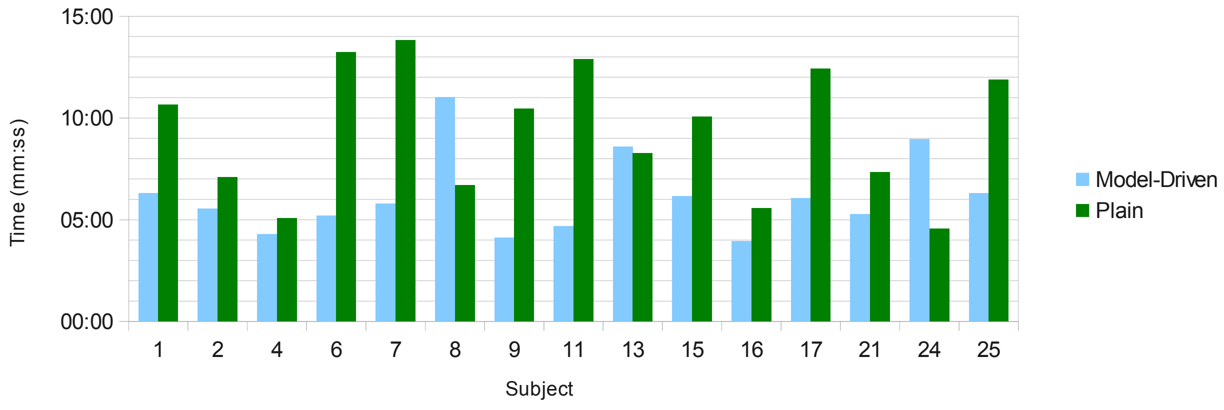


Figure 6.1: Time used to perform the tasks on the budget spreadsheet.

Similar results were obtained for the payments spreadsheet, as shown in Figure 6.2.

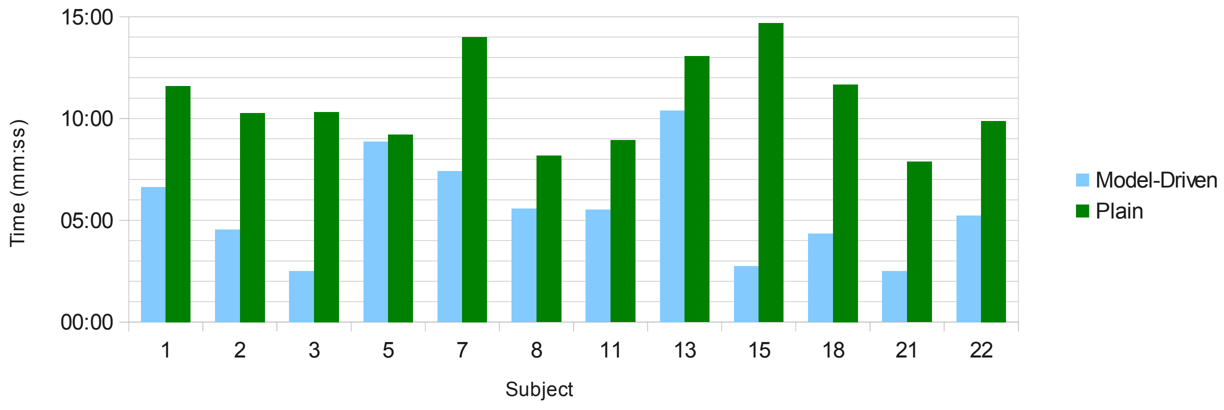


Figure 6.2: Time used to perform the tasks on the payments spreadsheet.

Error rates: To evaluate the correctness of the spreadsheets produced during the study, error rates are used. Each of the six tasks requires a set of spreadsheet operations to be correctly performed. Such operations included: adding a new row, adding a new column, changing the value of a cell, or changing the value of a formula. One error occurs when the participant does not perform one of those operations (e.g., the formula was not updated after inserting a new record), or the operation was performed incorrectly (e.g., the wrong value was introduced in the cell). The errors obtained correspond to the percentage of (sub)tasks that were not performed correctly.

Error rates for the plain budget spreadsheet are around 50%, most of the errors being related to wrong formulas. Some errors are also present in the model-driven version of this spreadsheet, but they are in much lesser quantity since the environment deals automatically with the formulas. The errors present in the model-driven version (and also present in the plain one) result from the input of wrong values, or their input in the wrong places. This information is graphically shown in Fig. 6.3.

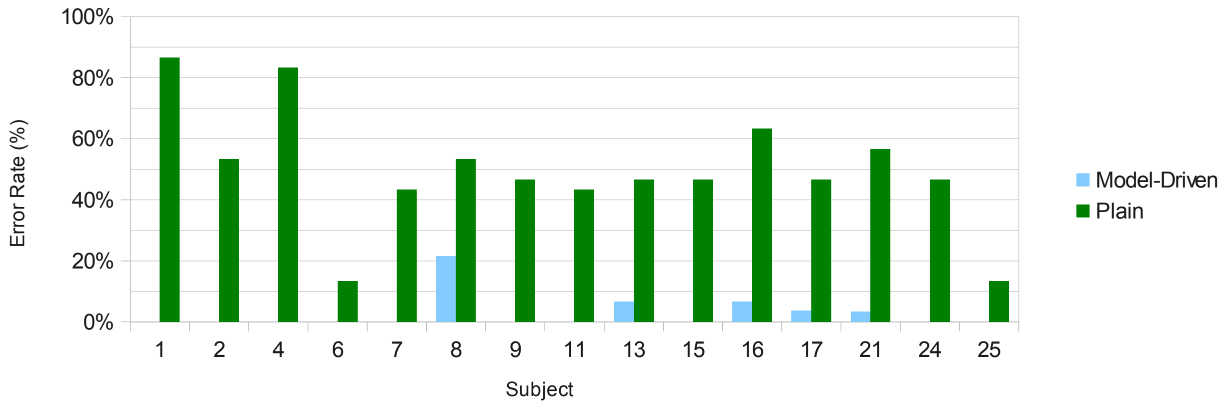


Figure 6.3: Error rate in the budget spreadsheet.

Similar results were obtained for the payments spreadsheet (see Fig. 6.4), with a slightly higher error rate for the plain version of the spreadsheet (around 60%).

This analysis was also performed at the task level, yielding similar results. They are not shown here since they do not bring any relevant information than what is already presented.

Subjects made mistakes in both plain and model-driven spreadsheets. Typical errors made in plain spreadsheets are the wrong or misplaced values, a wrong formula, and the lost of the spreadsheet structure. In the model-driven version the only type of errors are the wrong and

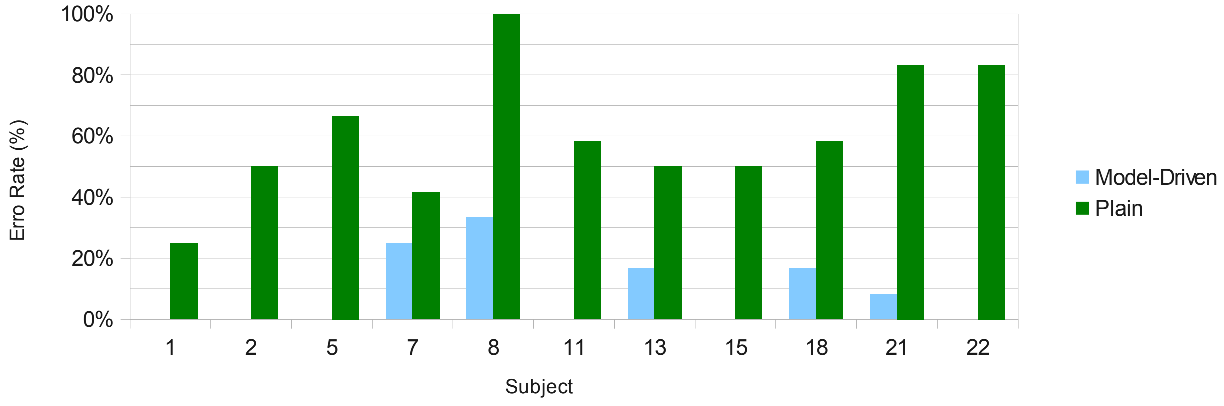


Figure 6.4: Error rate in the payments spreadsheet.

misplaced values.

Hypothesis Testing

The significance level used throughout the evaluation of all the tests is 0.05. The evaluation of the tests was performed using the R environment for statistical computing (R Core Team, 2013).

Comparison of times: The difference of times between the execution of tasks in plain spreadsheets and model-driven ones do not follow a normal distribution. Thus, we used the Wilcoxon test, which is the best fit for these cases (Wohlin et al., 2012).

The results obtained from the tests show that for model-driven spreadsheet development, in the particular case when our tool is used, the time taken to perform the tasks is statistically less than when using a plain spreadsheet (with p-value of 0.007882 for the budget spreadsheet and 0.0002441 for the payments one).

Comparison of error rates: The differences of error rates between plain spreadsheets and model-driven ones do not follow a normal distribution. Thus, we used a Wilcoxon test to test the null hypotheses for both spreadsheets, in order to be able to compare the results.

The results obtained from the tests show that for model-driven spreadsheet development, in the particular case when our tool is used, the number of errors are statistically less than when using a plain spreadsheet (with p-value of 0.0003579 for the budget spreadsheet and 0.001911 for the payments one).

6.1.4 Interpretation

The results from the analysis suggest that a model-driven approach to spreadsheet development can improve users' performance, while reducing the error rate. Moreover, from the questionnaires we can conclude that subjects felt more confident in the results of the model-driven approach compared to the the plain one. This indicates that some restrictions on the development process are welcome by the user since they understand this will make them perform better on their tasks.

Threats to validity

The goal of the study is to demonstrate a causal relationship between the use of a model-driven approach and improvements in the spreadsheet development process. Moreover, this study is defined to ensure that the actual setting used represents the theory we developed.

Next, validity threats for this study are analyzed, divided in four categories as defined in (Cook & Campbell, 1979), namely: conclusion validity, internal validity, construct validity, and external validity.

Conclusion validity. The main concern is the low statistical power due to the low number of participants. To overcome this issue more powerful statistical tests were performed where possible, taking always into account the necessary assumptions.

Problems related to measures can also arise, e.g., the times that the subjects took to perform the tasks. Nevertheless no significant differences to the real values are expected. Moreover, subjects performed the same tasks and in an environment that they are used to. Also, subjects have a similar background, which minimizes the risk of the variation being due to individual differences instead of the use of different treatments, but introduces problems when generalizing (see further on).

Internal validity. In order to minimize the effects on the independent variables that would reflect on the causality, several actions were taken. First, this study, with these subjects, is executed only once, some starting with plain spreadsheets and subsequently with the model-driven environment, while others starting with the model-driven environment and then working with the plain spreadsheets, with the goal to reduce learning effects. Second, the time to perform the study was reduced as much as possible so that the subjects could remain focused during all the study. Third, the instruments used (e.g., spreadsheets and questionnaires) were defined so that we could collect just what was needed. Fourth, all the subjects performed the same tasks, so issues from having different groups with distinct treatments do not arise. However, the tasks were performed in a random order to avoid learning effects. Specifying as much as possible the study, we obtained more control and reduced possible internal validity threats.

Construct validity. For this validity, several hypotheses to cover the aspects to analyze were defined with much detail, and mono-operation bias was reduced using two spreadsheets to collect data. Furthermore, the subjects were guaranteed to not be affected by this study, since they were not under evaluation (this was said to them several times during the study execution), and they were put at ease so that they would perform as much like in a real-world setting.

The tasks we asked users to perform are common in spreadsheets. Some of them, e.g., updating a formulas after adding/removing data, are automated by our setting exploiting information that is inferred from the underlying model. While this may suggest that MDSheet is being favored by our study, the fact is that such framework incorporates multiple automation opportunities while also adding ways to interact with spreadsheets that users are not accustomed to. So, our framework and its usage could actually suffer from participants: a) exploiting one automation opportunity when asked to explore another, e.g., due to misuse of MDSheet's features; and b) participants ending up constructing spreadsheets with an incorrect structure, e.g., due to misuse of instance evolution steps. Moreover, it would not be possible to compare both treatments equally if part of the operations were not considered.

Thus, we believe the study construction permits to evaluate the use MDSheet in a real scenario of usage.

External validity. This validity is related to the ability to generalize the results of the experiment to industrial practice. For that, the spreadsheets used to collect the data were based on real-world ones. However, since this study was performed with a small homogeneous group, the results from this study cannot be generalized without analyzing the domain where to apply.

Inferences

Since this study was performed in a very specific setting, we cannot generalize to every case. Nevertheless, our setting was the most similar possible to a real one, where spreadsheets were based on real ones and Master students studies can come close to ones with professionals (Höst, Regnell, & Wohlin, 2000), so the results could be as if it was performed in a industry setting with professionals. This can bring the possibility that model-driven spreadsheet development can be useful, and studies in the industry can be used to assess the methodology in specific cases.

6.1.5 Discussion

The empirical study we conducted reveals very promising results for model-driven spreadsheets. Although participants had little time to learn model-driven spreadsheets, they completed their tasks faster and with less errors using such spreadsheets compared to the ones without models. This suggests that with little training users can greatly benefit from the use of models guiding them. Moreover, as we theorized, it was apparently easy to use the model-driven setting in an environment users are used to, that is, a spreadsheet system.

Nevertheless, from the study results we can also see how to improve our framework. The errors committed by participants in the model-driven environment were similar to the ones found in the plain spreadsheets. Two kinds of errors were found: both correct and incorrect values inputted in the wrong cell and incorrect values inputted in the correct cell.

Although our observations seem to indicate that we are in the right research direction, they also indicate that more techniques and tools are necessary to aid users with *always* inputting the correct values in the correct places.

To improve on the first scenario, where values are inserted in wrong places, we believe the following solutions could help:

- Labels' context: as often happens, the spreadsheets we gave to participants had blocks of cells repeated over columns/rows (e.g., payments over months, budgets over years). Given this scenario, it is quite easy to scroll over the spreadsheet and lose visual contact with labels, mainly in large spreadsheets. We believe that keeping labels always visible would help users to input the values in the correct cells. Thus, we plan to make labels as visible as possible. This can be achieve in two ways: first, one could use the spreadsheet mechanism that allows to keep some columns/rows visible always; second, given that our spreadsheets have a corresponding model where labels are known, MDSheet could repeat label cells every number of columns/rows it would see appropriate, or it could also suggest the person creating the model to repeat such labels when again it would see fit.

- Complete row/column context: we believe the previous solution can still be improved. Even with labels visible, inputting values "far" from labels can be error-prone. A possible helper mechanism would be to highlight the column/row corresponding to the cell the user is currently selecting. This would give extra context to the users' actions and would possibly minimize inputting values in the wrong cells.

To help in the second scenario, that considers wrong values inserted in the right places, we believe to have already created improvement mechanisms in the past:

- The first one is the possibility of restricting the range and kind of values users can insert in the cells (J. Cunha, Fernandes, & Saraiva, 2012). For instance, when creating the model, it is possible to specify that a range of cells can only have values between a range of integers (for instance, between 0 and 10, for an exam marking spreadsheet). This will probably not avoid all input errors, but will avoid some. It can also be seen as documentation, as the error messages the users get explain them which range/kind of values are expected (J. Cunha, Fernandes, Mendes, & Saraiva, 2012).

- A second mechanism is the set of tools we have created to find bad smells and faults in spreadsheets. A bad smell may not be an error, but may lead to one. By pin pointing the bad smells in a spreadsheet may help the user to find the errors contained in it (J. Cunha, Fernandes, Mendes, Martins, & Saraiva, 2012; J. Cunha, Fernandes, Ribeiro, & Saraiva, 2012). These smells can further be analyzed by a fault-localization algorithm which will point more precisely to the faults contained in a spreadsheet (Abreu et al., 2014a, 2014b).

All these potential solutions need, of course, to be empirically validated. We plan to include them in MDSheet and pursue further validation both individually and in combination.

To improve the second problem, the wrong values, we propose the integration of known techniques such as testing (Abraham & Erwig, 2006a, 2007a, 2007b, 2009; Fisher II, Cao, Rothermel, Cook, & Burnett, 2002; Fisher II et al., 2006; Rothermel, Burnett, Li, Dupuis, & Sheretov, 2001) and smell finders (Asavametha, 2012; J. Cunha, Fernandes, Ribeiro, & Saraiva, 2012; Hermans, Pinzger, & Deursen, 2012; Hermans, Pinzger, & van Deursen, 2012). It is quite difficult to know if an inputted value is correct or not, but testing techniques could aid the user to know better. Moreover, smells can help users to search for places that can be potentially dangerous in the sense that errors can arise from those spreadsheet locations.

6.2 Bidirectional Evolution Environment

An empirical study was performed in order to assess the practicality and understandability of the bidirectional transformation features implemented in the MDSheet LibreOffice extension. This study, in contrast to the one presented in the previous section, was made to obtain feedback from the users and thus falls into the qualitative category.

Therefore, four people performed the study. They work in a secretariat at the University of Minho and have large experience in using spreadsheets. However, they are only end users and do not have any programming experience nor model-driven software development background.

The study was composed by:

- an explanation on the concepts and features of MDSheet;
- a tutorial where the participant was shown how to perform a given task, and then the participant would have to do a similar one;
- a set of tasks performed on two spreadsheets.

The study was designed for a controlled environment and in an off-line setting. The tasks were defined to represent realistic problems from real-world spreadsheets and were performed in a virtual machine in the author's computer.

Initial explanation. The explanation started with a description of the concepts behind model-driven engineering, i.e., the use of a model to describe the spreadsheet. Then, the concrete approach to model-driven spreadsheets was presented, with the description of what the model and its contents abstract in the spreadsheet.

Tutorial. The tutorial started with a description of the work, explaining the application of model-driven engineering to spreadsheets as is described in this work. It was supported by a list of tasks displayed in a web browser and a spreadsheet to perform those tasks. A spreadsheet to register the flights of an aviation company served as the base spreadsheet to perform the tasks tutorial. The tasks consisted in adding data to the spreadsheet (the instance), adding new features to the model (e.g., a column or row and some attribute), adding a new feature to the model but only in one repetition of the instance. With these tasks it was shown the necessary features to perform the tasks of the study. The duration of the preparation of the study was of about 30 minutes (10 minutes for the explanation part and 20 minutes to perform the tutorial tasks).

Study tasks. The study tasks were performed using two distinct spreadsheets. One of the spreadsheets was for budgeting, with two main categories for income and expenses along a series of months. Each of these categories contained multiple items (originally three and six, respectively) and there were records along twelve months. The other spreadsheet contained records of hourly oil extraction in different locations for a given day, (originally twenty-four hours and two locations).

Each participant performed tasks on both spreadsheets, but the order of each spreadsheet was random. The tasks consisted in adding a formula attribute, adding an entry to insert new data, and adding a new attribute in one object of the instance. The first task only required changes to the model, the second only to the instance and the third required changes to the instance and then to the model. However, no indication on which artefact the task should be performed was given. Therefore, the participant would have to think which of the artefacts would be the correct one to perform the task, thus showing if the participant understood the concept and how it was supposed to be used.

The study was one-on-one, i.e., subjects participated one at a time with the supervision of the author, and during their working hours. The tasks were performed by themselves, under the supervision of the author, with help when they were blocked for some reason. The participants were encouraged to think aloud, which allowed to understand the difficulties they faced.

Analysis. The participants faced some difficulties, namely due to concrete implementation of the embedding. One of the issues was the closeness of the button to add new objects to the data, which lead to the involuntary press of this button and the consequent addition of a new object. Nevertheless, this was of no importance to the study and was ignored when it happened (the participants told when they pressed involuntarily). Another technical issue was the limitation in the use of the *undo* feature. In one case, the participant corrected an incorrect action using further evolution operations. However, in other cases, the participant was said to ignore the issue and to proceed, with the incident being logged for posterior analysis.

The feedback given by the participants was related to the concrete implementation (without impact to the study) and related to the model-driven approach (also without impact to the study). Some said that certain tasks were simple due to the automation inherent to the approach and that the approach was accessible.

Interpretation. This study had only four participants and thus the interpretation of the results needs some caution. Nevertheless, the results show some important aspects to improve.

First, two of the participants were nervous and unfocused while the other two showed more interest in the tool and the approach. For the ones the former case, a study spread over time would have been more beneficial in order to introduce some familiarity and allowing the participants to feel more at ease and focused on the subject. This is directly related to how the study was performed. Nevertheless, all participants said that with some more experience they would have performed better.

Second, the given tasks were not very precise in terms of the concrete actions to perform. This was with the purpose to not bias them towards the correct answer. At first, mainly during the tutorial and a little while performing the study tasks and they were blocked, the participants were asked to talk about what was the solution they were thinking. Feedback was then given as the participants were not familiar with neither the problem nor the model-driven approach.

Finally, all participants made mistakes but, by the end of the study, the participants that were more focused were able to understand which artefact was the one where to perform the task. Also, as a result of the approach and implementation, some mistakes were prevented, namely the introduction of data or new attributes in the spreadsheet instance. This made them think about what they were doing and lead them to change their action to the correct one.

Discussion. This study approached the subject in a lightweight form, gathering feedback from a little exposure to the model-driven spreadsheets, the bidirectional environment itself and the used spreadsheets. In order to have an in-depth analysis of the value of the bidirectional transformation environment, a more detailed study is required. This study would require a more robust tool, allowing it to be used in an on-line scenario. This would allow to gather deeper information on the bidirectional environment, as the subjects would work on solutions that they are familiar with, and with a new paradigm that would be integrated in their workflow.

Nevertheless, the results are positive as some of the participants understood the approach at the end of the study and appreciated the benefits it can provide.

Chapter 7

Conclusion

Spreadsheets are used as a programming language to solve computation and storage problems. However, end user interaction is error prone. Spreadsheets lack tools to support application development. Many spreadsheet users are not aware of best practices to prevent errors, nor are aware of tools to help them audit or verify the correctness of their spreadsheet.

Proposed solutions in the literature are not sufficient. They either tackle specific issues or the solutions they provide are not user friendly. For instance, a situation that fits in the former case is the analysis of only spreadsheet formulas to help refactor them in order to prevent mistakes and make them more understandable. Another instance, which fits in the latter situation, consists in defining spreadsheet models that are usable by machines (e.g., to create or to process spreadsheets) but that users cannot take advantage of. Note that the user interaction is the most delicate part.

7.1 Model-Driven Spreadsheet Development

The work from this thesis brings the means to spreadsheet users build their solutions safely and with minimal effort. This work relies on common software development methodologies, namely model-driven engineering, and builds upon previous work on this topic, specifically on model-driven spreadsheet development.

As a first contribution, a new domain-specific modelling language for spreadsheets is presented. Tabula is a new language with origins on ClassSheets, with the ability to use a visual notation similar to spreadsheets to describe models. Its internal representation is completely different, allowing a more powerful specification of spreadsheets and overcoming limitations from the ClassSheet language. Tabula are also embedded within spreadsheets themselves providing a unique tool to develop spreadsheets, which users are already familiar with.

As another contribution, a bidirectional transformation engine was developed on top of Tabula and spreadsheets, allowing the evolution of spreadsheet specifications thus giving support to users along the life cycle of a spreadsheet.

The use of model-driven spreadsheets was evaluated with end users through empirical studies, using a prototype provided as an extension to LibreOffice Calc. This prototype, MD-Sheet, started by implementing embedded ClassSheets (now replaced by the Tabula language), its instances, the bidirectional transformation engine and further features for experimentation. The empirical studies aimed at:

1. assessing the expressiveness of the language to model real world spreadsheets;
2. validating the benefits of model-driven spreadsheets over plain spreadsheet usage, namely the decrease of errors and the increase in productivity;
3. assessing if end users understand the role of the bidirectional transformation engine and if they can use and benefit from it.

The expressiveness of the initial modelling language (embedded ClassSheets) was not enough to model many kinds of real-world spreadsheets, which lead to the development of Tabula, a more powerful language. Nevertheless, the application of a model-driven methodology to spreadsheet development is beneficial, reducing error rates and increasing the performance of end users. These benefits derive from the restrictions that the models impose on the development, preventing a class of errors by construction. However, users have some difficulties in understanding the bidirectional transformation effects in some cases at first, but they felt confident these difficulties would be removed with training and with more experience using the prototype.

The proposed solution intends to take common software development tools and techniques to create safe spreadsheets and improve end user performance automating repetitive and monotonous tasks. The results of the empirical studies show that this was accomplished to a certain point, with users being able to use the prototype, making less mistake and taking less time to accomplish the proposed tasks.

7.2 Spreadsheet Engineering

Spreadsheet users may profit from further research unrelated to model-driven engineering.

One topic of research is the use of other programming paradigms. On this topic, the author of this thesis cooperated with Pedro Maia, Henrique Rebêlo and João Saraiva to bring Aspect-Oriented Programming (AOP) to spreadsheets (Maia, Mendes, Cunha, Rebêlo, & Saraiva, 2015). This work presents a new language to apply aspects to spreadsheets, i.e., small parts of spreadsheet "code" that promote modularization and allows the introduction of new features to a spreadsheet in a non-intrusive manner.

Another research topic is the analysis of spreadsheets and their classification. Having spreadsheet corpora helps researchers to understand better the structure of spreadsheets and how users use them. Some existing corpora include the EUSES spreadsheet corpus (Fisher II & Rothermel, 2005), the Enron spreadsheets (Hermans & Murphy-Hill, 2015) and VEnron versioned corpus (Dou et al., 2016). From these examples, only the EUSES corpus includes some classification of the spreadsheets it contains, but which was performed manually. Therefore, taking the EUSES corpus as a starting point, a new methodology to classify spreadsheets into different domains was devised in collaboration with Kha N. Do and João Saraiva (Mendes, Do, & Saraiva, 2016). Although with good results in terms of classification, using the EUSES as a training data set is not ideal due to its biased and ad hoc manual classification, resulting in unrealistic automatic classification. Further work is required to produce an initial data set suitable to train the classification algorithms, allowing then researchers to benefit from the ability to process large spreadsheet corpora with minimal manual effort.

Further related work that relies on spreadsheet analysis includes smell detection, which helps to find potentially erroneous cells or patterns. Spreadsheet host systems already provide smell detection to some degree, but many more situations can be harmful to spreadsheets. These were taken from smell detection in common software, as introduced by Fowler (1999). These smells were applied to spreadsheet formulas and references (Hermans, Pinzger, & Deursen, 2012; Hermans, Pinzger, & van Deursen, 2012) and to cells in general (Abreu et al., 2014a, 2014b; J. Cunha, Fernandes, Mendes, Martins, & Saraiva, 2012; J. Cunha, Fernandes, Ribeiro, & Saraiva, 2012).

Many more research topics related to improving the spreadsheet paradigm exist, with much research for common software engineering being applicable to spreadsheets.

7.3 Future Work

The work presented in this dissertation covers the core of spreadsheet specification, with many features commonly used still needing to be formalized. Thus, future work is required to specify these features, but also to provide a solid spreadsheet development environment.

Tabula is a language to specify single spreadsheet tables. The use of multiple worksheets is not taken into account. However, support for multiple tables and instances, thus forming a complete spreadsheet file, can be added by expanding the referencing notation, e.g., by including the Tabula name as another level. Additional sheets and respective models bring further issues, e.g.:

- Are references to another model to all the Tabula instances, to a specific one or to a subset of the instances?

- How to specify a reference to a specific sheet in the model?

In order to find an optimal and practical solution, studies on real world spreadsheets are required. From these studies, we should gather information on how sheets are related at a higher level (i.e., at the level of the model), if that is a good solution, and how should that high-level information be related in a concrete spreadsheet solution.

Furthermore, since Tabulae model tables, how should the other features in spreadsheets be modelled? These other features include pivot tables and charts amongst other common features.

An initial solution for these problems as already been presented and is being developed (Mendes et al., 2017a, 2017b). This solution, instead of modelling the spreadsheet itself, models the process to construct a spreadsheet, using ClassSheet models to specify the tables. The use of these ClassSheet models can be replaced by Tabula models. This work shows the necessity of having powerful formal models able to specify artefacts that are developed and used by users lacking in programming knowledge, but also to guide those users.

Additionally, Tabula models can be used in any other kind of software to describe tables that do not just follow the pattern with one header row and multiple other rows for data and that also contain computation using formulas.

References

Abraham, R., & Erwig, M. (2006a). Autotest: A tool for automatic test case generation in spreadsheets. In *2006 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 43–50). IEEE Computer Society. doi: 10.1109/VLHCC.2006.11

Abraham, R., & Erwig, M. (2006b). Type inference for spreadsheets. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP)* (pp. 73–84). ACM. doi: 10.1145/1140335.1140346

Abraham, R., & Erwig, M. (2007a). GoalDebug: A spreadsheet debugger for end users. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)* (pp. 251–260). IEEE Computer Society. doi: 10.1109/ICSE.2007.39

Abraham, R., & Erwig, M. (2007b). UCheck: A spreadsheet type checker for end users. *Journal of Visual Languages & Computing, 18*(1), 71–95. doi: 10.1016/j.jvlc.2006.06.001

Abraham, R., & Erwig, M. (2009). Mutation operators for spreadsheets. *IEEE Transactions on Software Engineering, 35*(1), 94–108. doi: 10.1109/TSE.2008.73

Abraham, R., Erwig, M., Kollmansberger, S., & Seifert, E. (2005). Visual specifications of correct spreadsheets. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 189–196). IEEE Computer Society. doi: 10.1109/VLHCC.2005.70

Abreu, R., Cunha, J., Fernandes, J. P., Martins, P., Perez, A., & Saraiva, J. (2014a). FaultySheet Detective: When smells meet fault localization. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 625–628). IEEE Computer Society. doi: 10.1109/ICSME.2014.111

Abreu, R., Cunha, J., Fernandes, J. P., Martins, P., Perez, A., & Saraiva, J. (2014b). Smelling faults in spreadsheets. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 111–120). IEEE Computer Society. doi: 10.1109/ICSME.2014.33

Asavametha, A. (2012). *Detecting bad smells in spreadsheets* (Master's thesis). Oregon State University.

Biere, A., Cimatti, A., Clarke, E. M., Strichman, O., & Zhu, Y. (2003). Bounded model checking. *Advances in Computers*, *58*(11), 117–148.

Cook, T. D., & Campbell, D. T. (1979). *Quasi-experimentation: Design & analysis issues for field settings*. Houghton Mifflin.

Cunha, A. (2014). Bounded model checking of temporal formulas with Alloy. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z* (pp. 303–308). Springer Berlin Heidelberg. doi: 10.1007/978-3-662-43652-3_29

Cunha, J., Fernandes, J. P., Mendes, J., Martins, P., & Saraiva, J. (2012). SmellSheet Detective: A tool for detecting bad smells in spreadsheets. In *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 243–244). IEEE Computer Society. doi: 10.1109/VLHCC.2012.6344535

Cunha, J., Fernandes, J. P., Mendes, J., Pacheco, H., & Saraiva, J. (2012). Bidirectional transformation of model-driven spreadsheets. In *Theory and Practice of Model Transformations – ICMT 2012* (pp. 105–120). Springer-Verlag Berlin Heidelberg. doi: 10.1007/978-3-642-30476-7_7

Cunha, J., Fernandes, J. P., Mendes, J., & Saraiva, J. (2011). Embedding and evolution of spreadsheet models in spreadsheet systems. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 179–186). IEEE Computer Society. doi: 10.1109/VLHCC.2011.6070396

Cunha, J., Fernandes, J. P., Mendes, J., & Saraiva, J. (2012). Extension and implementation of ClassSheet models. In *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 19–22). IEEE Computer Society. doi: 10.1109/VLHCC.2012.6344473

Cunha, J., Fernandes, J. P., Mendes, J., & Saraiva, J. (2015). Embedding, evolution, and validation of model-driven spreadsheets. *IEEE Transactions on Software Engineering, 41*(3), 241–263. doi: 10.1109/TSE.2014.2361141

Cunha, J., Fernandes, J. P., Ribeiro, H., & Saraiva, J. (2012). Towards a catalog of spreadsheet smells. In *Computational Science and Its Applications – ICCSA 2012* (pp. 202–216). Springer-Verlag Berlin Heidelberg. doi: 10.1007/978-3-642-31128-4_15

Cunha, J., Fernandes, J. P., & Saraiva, J. (2012). From relational classsheets to UML+OCL. In *ACM Symposium on Applied Computing (SAC)* (pp. 1151–1158). ACM. doi: 10.1145/2245276.2231957

Cunha, J., Saraiva, J., & Visser, J. (2009). From spreadsheets to relational databases and back. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)* (pp. 179–188). ACM. doi: 10.1145/1480945.1480972

Czarnecki, K., Foster, J. N., Hu, Z., Lämmel, R., Schürr, A., & Terwilliger, J. F. (2009). Bidirectional transformations: A cross-discipline perspective. In *Theory and Practice of Model Transformations – ICMT 2009* (pp. 260–283). Springer-Verlag Berlin Heidelberg. doi: 10.1007/978-3-642-02408-5_19

Diskin, Z. (2008). Algebraic models for bidirectional model synchronization. In *Model Driven Engineering Languages and Systems – MODELS 2008* (pp. 21–36). Springer Berlin Heidelberg. doi: 10.1007/978-3-540-87875-9_2

Dou, W., Xu, L., Cheung, S.-C., Gao, C., Wei, J., & Huang, T. (2016). VEnron: A versioned spreadsheet corpus and related evolution analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)* (pp. 162–171). ACM. doi: 10.1145/2889160.2889238

Engels, G., & 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)* (pp. 124–133). ACM. doi: 10.1145/1101908.1101929

Erwig, M., Abraham, R., Cooperstein, I., & Kollmansberger, S. (2005). Automatic generation and maintenance of correct spreadsheets. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)* (pp. 136–145). ACM. doi: 10.1145/1062455.1062494

EuSpRiG. (n.d.). *Spreadsheet mistakes - news stories collated by the European Spreadsheet Risks Interest Group.* Retrieved 2017-09-30, from `http://www.eusprig.org/horror-stories.htm`

FAST. (2014). *The FAST Standard.* Retrieved 2017-09-30, from `http://www.fast-standard.org/the-standard`

Fisher II, M., Cao, M., Rothermel, G., Cook, C., & Burnett, M. (2002). Automated test case generation for spreadsheets. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)* (pp. 141–153). ACM. doi: 10.1145/581339.581359

Fisher II, M., & Rothermel, G. (2005). The euses spreadsheet corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms. In *Proceedings of the First Workshop on End-User Software Engineering (WEUSE)* (pp. 1–5). ACM. doi: 10.1145/1082983.1083242

Fisher II, M., Rothermel, G., Brown, D., Cao, M., Cook, C., & Burnett, M. (2006). Integrating automated test generation into the WYSIWYT spreadsheet testing methdology. *ACM Transactions on Software Engineering and Methodology (TOSEM), 15*(2), 150–194. doi: 10.1145/1131421.1131423

Fowler, M. (1999). *Refactoring: Improving the design of existing code.* Addison-Wesley.

Harutyunyan, A., Borradaile, G., Chambers, C., & Scaffidi, C. (2012). Planted-model evaluation of algorithms for identifying differences between spreadsheets. In *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 7–14). IEEE Computer Society. doi: 10.1109/VLHCC.2012.6344471

Henley, A. Z., & Fleming, S. D. (2014). The patchworks code editor: Toward faster navigation with less code arranging and fewer navigation mistakes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)* (pp. 2511–2520). ACM. doi: 10.1145/2556288.2557073

Hermans, F. (2013). *Analyzing and visualizing spreadsheets* (PhD thesis). Delft University of Technology.

Hermans, F., & Murphy-Hill, E. (2015). Enron's spreadsheets and related emails: A dataset and analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*

(ICSE) (pp. 7–16). IEEE. doi: 10.1109/ICSE.2015.129

Hermans, F., Pinzger, M., & Deursen, A. v. (2012). Detecting and visualizing inter-worksheet smells in spreadsheets. In *2012 34th International Conference on Software Engineering (ICSE)* (pp. 441–451). IEEE. doi: 10.1109/ICSE.2012.6227171

Hermans, F., Pinzger, M., & van Deursen, A. (2011). Supporting professional spreadsheet users by generating leveled dataflow diagrams. In *2011 33rd International Conference on Software Engineering (ICSE)* (pp. 451–460). ACM. doi: 10.1145/1985793.1985855

Hermans, F., Pinzger, M., & van Deursen, A. (2012). Detecting code smells in spreadsheet formulas. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)* (pp. 409–418). IEEE. doi: 10.1109/ICSM.2012.6405300

Höst, M., Regnell, B., & Wohlin, C. (2000). Using students as subjects – a comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering, 5*(3), 201–214. doi: 10.1023/A:1026586415054

ICAEW. (2014). *Twenty principles for good spreadsheet practice.* Retrieved 2017-09-30, from `https://www.icaew.com/en/technical/information-technology/excel/twenty-principles`

Inoue, K., & Nakamura, A. (1977). Some properties of two-dimensional on-line tessellation acceptors. *Information Sciences, 13*(2), 95–121. doi: 10.1016/0020-0255(77)90023-8

Ireson-Paine, J. (1997). Model Master: an object-oriented spreadsheet front-end. In *Computer-Aided Learning using Technology in Economies and Business Education (CALECO).*

Ireson-Paine, J. (2001). Ensuring spreadsheet integrity with Model Master. In *Proceedings of EuSpRIG 2001.*

Ireson-Paine, J. (2005). Excelsior: Bringing the benefits of modularisation to Excel. In *Proceedings of EuSpRIG 2005.*

Jackson, D. (2002). Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM), 11*(2), 256–290. doi: 10.1145/505145.505149

Jackson, D. (2012). *Software abstractions: Logic, language, and analysis.* MIT press.

Kelso, P. (2012, Jan. 04). *London 2012 olympics: lucky few to get 100m final tickets after synchronised swimming was overbooked by 10,000.* Retrieved 2017-09-30, from `http://www.telegraph.co.uk/sport/olympics/8992490/London-2012-Olympics-lucky-few-to-get-100m-final-tickets-after-synchronised-swimming-was-overbooked-by-10000.html`

Konczal, M. (2013, Apr. 16). *Researchers finally replicated Reinhart-Rogoff, and there are serious problems.* Retrieved 2017-09-30, from `http://rooseveltinstitute.org/researchers-finally-replicated-reinhart-rogoff-and-there-are-serious-problems/`

Lämmel, R. (2004). Coupled software transformations. In *First International Workshop on Software Evolution Transformations* (pp. 31–35).

Macedo, N., Jorge, T., & Cunha, A. (2017). A feature-based classification of model repair approaches. *IEEE Transactions on Software Engineering, 43*(7), 615–640. doi: 10.1109/TSE.2016.2620145

Maia, P., Mendes, J., Cunha, J., Rebêlo, H., & Saraiva, J. (2015). Towards the design and implementation of aspect-oriented programming for spreadsheets. *CoRR, abs/1503.03463.* Retrieved from `http://arxiv.org/abs/1503.03463`

MathWorks. (n.d.). *Matlab.* Retrieved 2017-09-30, from `https://www.mathworks.com/products/matlab.html`

McKeever, R., & McDaid, K. (2010). How do range names hinder novice spreadsheet debugging performance? *CoRR, abs/1009.2765.* Retrieved from `http://arxiv.org/abs/1009.2765`

McKeever, R., & McDaid, K. (2011). Effect of range naming conventions on reliability and development time for simple spreadsheet formulas. *CoRR, abs/1111.6872.* Retrieved from `http://arxiv.org/abs/1111.6872`

McKeever, R., McDaid, K., & Bishop, B. (2009). An exploratory analysis of the impact of named ranges on the debugging performance of novice users. *CoRR, abs/0908.0935.* Retrieved from `http://arxiv.org/abs/0908.0935`

Mendes, J. (2011). ClassSheet-driven spreadsheet environments. In *2011 IEEE Symposium on*

Visual Languages and Human-Centric Computing (VL/HCC) (pp. 235–236). IEEE Computer Society. doi: 10.1109/VLHCC.2011.6070409

Mendes, J. (2012). *Evolution of model-driven spreadsheets* (Master's thesis). Universidade do Minho.

Mendes, J., Cunha, J., Duarte, F., Engels, G., Saraiva, J., & Sauer, S. (2017a). Systematic spreadsheet construction processes. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 123–127). IEEE Computer Society. doi: 10.1109/VLHCC.2017.8103459

Mendes, J., Cunha, J., Duarte, F., Engels, G., Saraiva, J., & Sauer, S. (2017b). Towards systematic spreadsheet construction processes. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)* (pp. 356–358). IEEE. doi: 10.1109/ICSE-C.2017.141

Mendes, J., Do, K. N., & Saraiva, J. (2016). Towards an automated classification of spreadsheets. In *Software Technologies: Applications and Foundations – STAF 2016* (pp. 346–355). Springer International Publishing. doi: 10.1007/978-3-319-50230-4_26

Meyer, B. (1997). *Object-oriented software construction, 2nd edition.* Prentice-Hall.

Panko, R. R., & Ordway, N. (2008). Sarbanes-oxley: What about all the spreadsheets? *CoRR, abs/0804.0797*. Retrieved from `http://arxiv.org/abs/0804.0797`

Petre, M. (1995). Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, *38*(6), 33–44. doi: 10.1145/203241.203251

PwC. (2014, Oct.). *Treasury shown in a new light – PwC global treasury survey 2014.*

PwC. (2017, Jan.). *The 'virtual reality' of treasury – global treasury benchmark survey 2017.*

R Core Team. (2013). R: A language and environment for statistical computing [Computer software manual]. (`http://www.r-project.org`)

Reinhart, C. M., & Rogoff, K. S. (2010). *Growth in a time of debt* (Tech. Rep.). National Bureau of Economic Research.

Reuters. (2012, Jan. 09). *Astrazeneca reaffirms outlook after mistaken release.* Retrieved 2017-09-30, from `http://uk.reuters.com/article/uk-astrazeneca-idUKTRE8080BX20120109`

Rothermel, G., Burnett, M., Li, L., Dupuis, C., & Sheretov, A. (2001). A methodology for testing spreadsheets. *ACM Transactions on Software Engineering and Methodology (TOSEM), 10*(1), 110–147. doi: 10.1145/366378.366385

Schmidt, D. C. (2006). Guest editor's introduction: Model-Driven Engineering. *Computer, 39*(2), 25–31. doi: 10.1109/MC.2006.58

SSRB. (2010). *Best Practice Spreadsheet Modelling Standards (version 6.1).* Retrieved from `http://www.ssrb.org/standards`

Stevens, P. (2007). Bidirectional model transformations in QVT: Semantic issues and open questions. In *Model Driven Engineering Languages and Systems – MODELS 2007* (p. 1-15). Springer Berlin Heidelberg. doi: 10.1007/978-3-540-75209-7_1

Swierstra, S. D., Henriques, P. R., & Oliveira, J. N. (Eds.). (1999). *Advanced functional programming* (Vol. 1608). Springer.

Torlak, E., & Dennis, G. (2006). Kodkod for alloy users. In *First Alloy Workshop.*

van Deursen, A., Klint, P., & Visser, J. (2000). Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices, 35*(6), 26–36. doi: 10.1145/352029.352035

Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L. C. L., ... Wachsmuth, G. (2013). *DSL engineering – designing, implementing and using domain-specific languages.* dslbook.org. Retrieved from `http://www.dslbook.org`

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., & Wesslén, A. (2012). *Experimentation in software engineering.* Springer.