Universidade do Minho
Escola de Engenharia

André António dos Santos da Silva

Directed Evolution of Model-Driven Spreadsheets

Setembro 2013

Universidade do Minho
Escola de Engenharia
Departamento de Informática

André António dos Santos da Silva

Directed Evolution of Model-Driven Spreadsheets

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Doutor João Alexandre Saraiva
Doutor Jácome Miguel Cunha

Setembro de 2013

Acknowledgments

First of all, I would like to thank my supervisor and co-supervisor Prof. Dr. João Saraiva and Dr. Jácome Cunha, respectively, for granting me the opportunity to work on such an interesting subject matter, and for the huge support given to me during the duration of this thesis.

I would also like to extend my gratitude to Dr. João Paulo Fernandes for very valuable input regarding many aspects of the research.

My appreciation also goes to everyone that contributed to this thesis, be it a small or big contribution, especially to Jorge Mendes for all the work related to ClassSheet to automata conversion, integration with MDSheet, and for all the invaluable advice related to automata atomic operations.

Lastly I would like to give a very special thank you to the people that encouraged me in the last two years, and my family, especially my mother, Ana Santos.

This work is financed (or partially financed) by FEDER Funds through the Programa Operacional Factores de Competitvidade - COMPETE and by National Funds through FCT - Fundação para a Ciência e a Tecnologia under FCOMP-01-0124- FEDER-010048 and FCOMP-01-0124-FEDER-020532 projects.

Abstract

Directed Evolution of Model-Driven Spreadsheets

Spreadsheets are among the most used programming languages today. The easy to use and the intuitive nature of the visual interface makes them a preferred programming tool for any kind of individual or organization. The flexibility they provide to organize data as users need to is one of the reasons that makes them so popular. However, this flexibility also makes them very error-prone.

In order to improve spreadsheet quality and reduce the number of errors, software engineering practices were introduced, namely object oriented and model-driven techniques. These techniques enabled the specification of the spreadsheet business logic, which offers the possibility to better structure data, while at the same time narrowing the range of types of errors made by user input. While these developments had a huge impact, spreadsheet evolution is still an inherently human process, which is in itself error-prone.

In many real world applications of spreadsheets, they are used to store and disseminate data between different systems. Different systems can use different data formats, this leads to the need to change and adapt the data produced by a source system so that it complies to the data format consumed by a target system. Usually in these cases, both the initial and final data models are known in advance.

The objective of this thesis is to present techniques that enable data evolution to be made automatically, using model-driven spreadsheets.

Keywords: Spreadsheet; Model-Driven; Error; Automatic; Evolution.

Resumo

Evolução Dirigida de Folhas de Cálculo Orientadas por Modelos

Folhas de cálculo são um dos paradigmas de programação mais utilizados actualmente. A sua facilidade de utilização e reduzida curva de aprendizagem torna-as numa das ferramentas de programação mais utilizadas diariamente por milhões de indivíduos e organizações. A flexibilidade concedida pelas folhas de cálculo para organizar dados consoante a preferência dos utilizadores é uma das razões que as torna tão populares. Esta flexibilidade tem, contudo, uma grande desvantagem: torna-as muito propícias a erros.

De forma a elevar a qualidade, e reduzir o número de erros em folhas de cálculo, foram introduzidas práticas já estabelecidas em engenharia de software, nomeadamente técnicas de desenvolvimento orientado a objectos e desenvolvimento dirigido por modelos. Com estas técnicas passou a ser possível especificar a lógica de negócio de folhas de cálculo, o que proporciona a estruturação dos dados nelas contidos e, ao mesmo tempo, limita o tipo de erros passíveis de serem cometidos pelos utilizadores. Embora estes desenvolvimentos tenham tido um grande impacto, a evolução de folhas de cálculo continua a ser um processo inerentemente humano, o que pode, ainda assim, originar erros.

Em muitos casos reais de utilização de folhas de cálculo, elas são utilizadas para armazenar e disseminar informação entre diferentes sistemas. Diferentes sistemas podem utilizar diferentes formatos de dados, isto leva à necessidade de adaptar os dados produzidos por um sistema para que sejam compatíveis com um determinado sistema de destino. Normalmente nestes casos, ambos os modelos de dados são conhecidos à partida. O objectivo desta tese é apresentar um conjunto de técnicas que permitam fazer esta evolução de forma totalmente automática, utilizando para isso folhas de cálculo dirgidas por modelos.

Contents

Acronyms

MDSD	Model-Driven Spreadsheet Development
MDSE	Model-Driven Spreadsheet Engineering
OOD	Object-Oriented Development
MDD	Model-Driven Development
MDSD	Model-Driven Software Development
MDE	Model-Driven Engineering
CSV	Comma-Separated Values
XML	Extensible Markup Language
DFA	Deterministic Finite Automata
UML	Unified Modeling Language
USE	UML-based Specification Environment
OCL	Object Constraint Language
BA	Banco Alimentar de Braga
EuSpRiG	European Spreadsheets Risks Interest Group

List of Figures

List of Algorithms

Chapter 1

Introduction

For the last 4500 years tables have been used to structure information and as a major computational aid. Although they are very simplistic in nature, they are one of the most important mathematical tools used in scientific advancement. Its uses range from representation of mathematical functions to summarizing empirical values [Campbell-Kelly, 2007a].

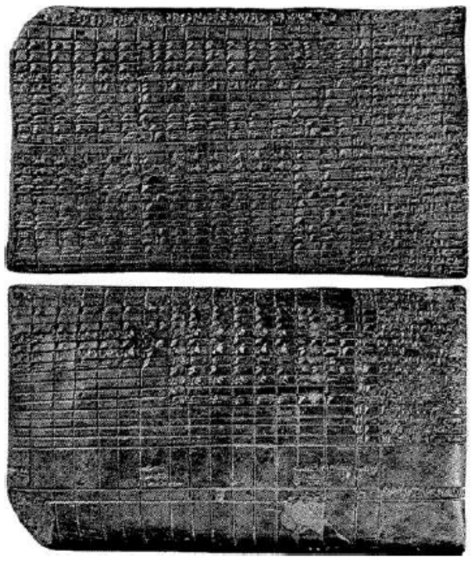
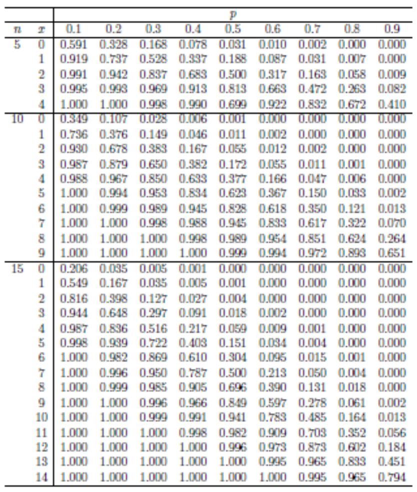


Figure 1.1: Month-by-month wage account for the Sumerian temple of Enlil at Nippur, for the year 1295 BC [Robson, 2007].

One of the first civilizations to use tables was the Sumerian. Sumerians employed tables to keep track of livestock and wage account. Figure 1.1 shows a tablet utilized to keep record of monthly salaries of temple personnel for the Sumerian temple of Enlil [Robson, 2007]. The tablet data layout is very similar to current days data organization: column headings at the top of the table indicate month names and each row exhibits monthly wages for a person, with subtotals each six months and a yearly total. The last row indicates names and professions [Campbell-Kelly, 2007b].

Since the Sumerian civilization, tables have been used in many other applications, from statistic (Figure 1.2), to economy (Figure 1.3) and, more recently, as the development of dynamic table structures in computer science (Figure 1.4). These show that there is still importance in display information in a tabular way.



		p								
n	x	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
5	0	0.591	0.328	0.168	0.078	0.031	0.010	0.002	0.000	0.000
	1	0.919	0.737	0.528	0.337	0.188	0.087	0.031	0.007	0.000
	2	0.991	0.942	0.837	0.683	0.500	0.317	0.163	0.058	0.009
	3	0.995	0.993	0.969	0.913	0.813	0.663	0.472	0.263	0.082
	4	1.000	1.000	0.998	0.990	0.699	0.922	0.832	0.672	0.410
10	0	0.349	0.107	0.028	0.006	0.001	0.000	0.000	0.000	0.000
	1	0.736	0.376	0.149	0.046	0.011	0.002	0.000	0.000	0.000
	2	0.930	0.678	0.383	0.167	0.055	0.012	0.002	0.000	0.000
	3	0.987	0.879	0.650	0.382	0.172	0.055	0.011	0.001	0.000
	4	0.988	0.967	0.850	0.633	0.377	0.166	0.047	0.006	0.000
	5	1.000	0.994	0.953	0.834	0.623	0.367	0.150	0.033	0.002
	6	1.000	0.999	0.989	0.945	0.828	0.618	0.350	0.121	0.013
	7	1.000	1.000	0.998	0.988	0.945	0.833	0.617	0.322	0.070
	8	1.000	1.000	1.000	0.998	0.989	0.954	0.851	0.624	0.264
	9	1.000	1.000	1.000	1.000	0.999	0.994	0.972	0.893	0.651
15	0	0.206	0.035	0.005	0.001	0.000	0.000	0.000	0.000	0.000
	1	0.549	0.167	0.035	0.005	0.001	0.000	0.000	0.000	0.000
	2	0.816	0.398	0.127	0.027	0.004	0.000	0.000	0.000	0.000
	3	0.944	0.648	0.297	0.091	0.018	0.002	0.000	0.000	0.000
	4	0.987	0.836	0.516	0.217	0.059	0.009	0.001	0.000	0.000
	5	0.998	0.939	0.722	0.403	0.151	0.034	0.004	0.000	0.000
	6	1.000	0.982	0.869	0.610	0.304	0.095	0.015	0.001	0.000
	7	1.000	0.996	0.950	0.787	0.500	0.213	0.050	0.004	0.000
	8	1.000	0.999	0.985	0.905	0.696	0.390	0.131	0.018	0.000
	9	1.000	1.000	0.996	0.966	0.849	0.597	0.278	0.061	0.002
	10	1.000	1.000	0.999	0.991	0.941	0.783	0.485	0.164	0.013
	11	1.000	1.000	1.000	0.998	0.982	0.909	0.703	0.352	0.056
	12	1.000	1.000	1.000	1.000	0.996	0.973	0.873	0.602	0.184
	13	1.000	1.000	1.000	1.000	1.000	0.995	0.965	0.833	0.451
	14	1.000	1.000	1.000	1.000	1.000	1.000	0.995	0.965	0.794

Figure 1.2: Binomial distribution table [1].

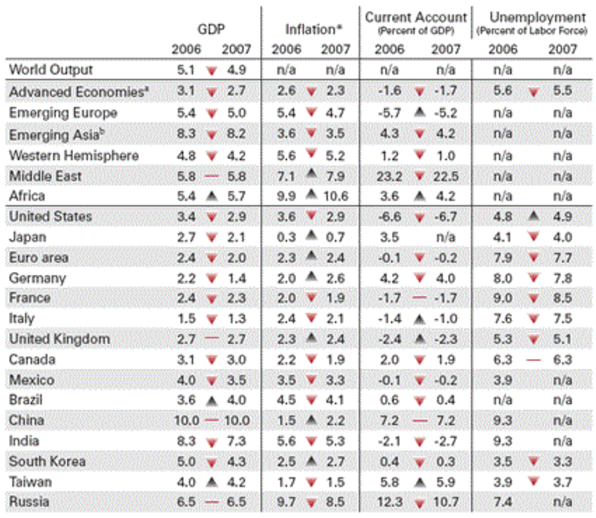
What makes tables so prevalent is that they allow to easily select, categorize, check, calculate and extract data. Data can be presented in such a way that it can be systematically processed.

Tables can be used to solve problems in almost every sphere of knowledge, the one we focus on this thesis is related to spreadsheets.

[1]`http://sites.stat.psu.edu/~mga/401/tables/binom.pdf`. (19-09-2013)

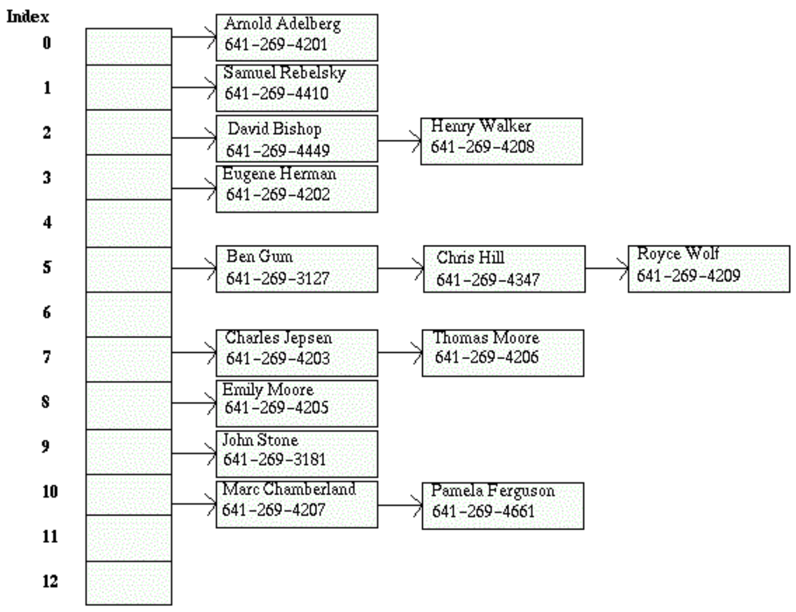
[2]`http://www.ibrc.indiana.edu/ibr/2006/outlook/international.html`. (19-09-2013)

[3]`http://www.cs.grinnell.edu/~walker/courses/153.sp02/lab-hashtables-inheritance.html`. (19-09-2013)



	GDP			Inflation*			Current Account (Percent of GDP)			Unemployment (Percent of Labor Force)		
	2006		2007	2006		2007	2006		2007	2006		2007
World Output	5.1	▼	4.9	n/a		n/a	n/a		n/a	n/a		n/a
Advanced Economies[a]	3.1	▼	2.7	2.6	▼	2.3	-1.6	▼	-1.7	5.6	▼	5.5
Emerging Europe	5.4	▼	5.0	5.4	▼	4.7	-5.7	▲	-5.2	n/a		n/a
Emerging Asia[b]	8.3	▼	8.2	3.6	▼	3.5	4.3	▼	4.2	n/a		n/a
Western Hemisphere	4.8	▼	4.2	5.6	▼	5.2	1.2	▼	1.0	n/a		n/a
Middle East	5.8	—	5.8	7.1	▲	7.9	23.2	▼	22.5	n/a		n/a
Africa	5.4	▲	5.7	9.9	▲	10.6	3.6	▲	4.2	n/a		n/a
United States	3.4	▼	2.9	3.6	▼	2.9	-6.6	▼	-6.7	4.8	▲	4.9
Japan	2.7	▼	2.1	0.3	▲	0.7	3.5		n/a	4.1	▼	4.0
Euro area	2.4	▼	2.0	2.3	▲	2.4	-0.1	▼	-0.2	7.9	▼	7.7
Germany	2.2	▼	1.4	2.0	▲	2.6	4.2	▼	4.0	8.0	▼	7.8
France	2.4	▼	2.3	2.0	▼	1.9	-1.7	—	-1.7	9.0	▼	8.5
Italy	1.5	▼	1.3	2.4	▼	2.1	-1.4	▲	-1.0	7.6	▼	7.5
United Kingdom	2.7	—	2.7	2.3	▲	2.4	-2.4	▲	-2.3	5.3	▼	5.1
Canada	3.1	▼	3.0	2.2	▼	1.9	2.0	▼	1.9	6.3	—	6.3
Mexico	4.0	▼	3.5	3.5	▼	3.3	-0.1	▼	-0.2	3.9		n/a
Brazil	3.6	▲	4.0	4.5	▼	4.1	0.6	▼	0.4	n/a		n/a
China	10.0	—	10.0	1.5	▲	2.2	7.2	—	7.2	9.3		n/a
India	8.3	▼	7.3	5.6	▼	5.3	-2.1	▼	-2.7	9.3		n/a
South Korea	5.0	▼	4.3	2.5	▲	2.7	0.4	▼	0.3	3.5	▼	3.3
Taiwan	4.0	▲	4.2	1.7	▼	1.5	5.8	▲	5.9	3.9	▼	3.7
Russia	6.5	—	6.5	9.7	▼	8.5	12.3	▼	10.7	7.4		n/a

Figure 1.3: Projected world economic growth for 2006 [2].

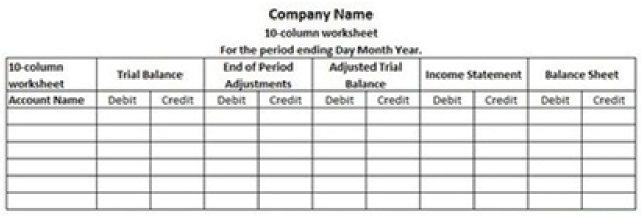


Index
0
1
2
3
4
5
6
7
8
9
10
11
12
Arnold Adelberg 641-269-4201
Samuel Rebelsky 641-269-4410
David Bishop 641-269-4449
Henry Walker 641-269-4208
Eugene Herman 641-269-4202
Ben Gum 641-269-3127
Chris Hill 641-269-4347
Royce Wolf 641-269-4209
Charles Jepsen 641-269-4203
Thomas Moore 641-269-4206
Emily Moore 641-269-4205
John Stone 641-269-3181
Marc Chamberland 641-269-4207
Pamela Ferguson 641-269-4661

Figure 1.4: Storage of names and numbers in a Hash Table [3].

1.1 Spreadsheets

The terms spreadsheet and worksheet originated in accounting even before electronic spreadsheets existed. Both had the same meaning, but the term worksheet was mostly used until 1970 [Campbell-Kelly, 2007b]. These worksheets, as shown in Figure 1.5, were standardized 6 or 10 column pre-printed paper sheets used by accountants to construct trial balances. After 1970 the term spreadsheet became more widely used [Campbell-Kelly, 2007b].


Company Name
10-column worksheet
For the period ending Day Month Year.

10-column worksheet	Trial Balance		End of Period Adjustments		Adjusted Trial Balance		Income Statement		Balance Sheet	
Account Name	Debit	Credit	Debit	Credit	Debit	Credit	Debit	Credit	Debit	Credit

Figure 1.5: 10-Column Worksheet [4].

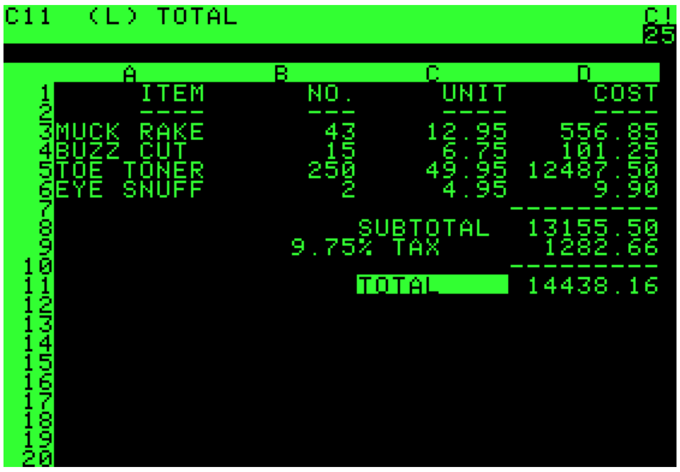
While spreadsheets were very used on paper, they were not used electronically due to the lack of software solutions. During the 1960s and 1970s most financial software bundles were developed to run on mainframe computers and time-sharing systems. Two of the main problems of these software solutions were that they were extremely expensive and required a technical expertise to operate [Campbell-Kelly, 2007b]. All that changed in 1979 when VisiCalc was released for the Apple II system [Bricklin]. The affordable price and the easy to use tabular interface made it a tremendous success, mainly because it did not need any programming knowledge to be operated. VisiCalc was the first spreadsheet software to include a textual interface composed by cells, as seen in Figure 1.6, and established how the graphical interface of every other spreadsheet software that came after it would be like. Other important aspect included the fast recalculation of values every time a cell was changed, as opposed to previous solutions that took hours to compute results under the same circumstances [Campbell-Kelly, 2007b].

In 1984, Lotus 1-2-3 was released with major improvements, which included graphics generation, better performance, and user friendly interface, which led it to dethrone VisiCalc as the number one spreadsheet system. It was only in 1990, when Microsoft Windows gained significant market share, that Lotus 1-2-3 lost the position as the most sold spreadsheet software. At that time only Microsoft Excel[6] was compatible with Windows, which raised sales by a huge amount making it the market leading spreadsheet system [Campbell-Kelly, 2007b]. A lot has changed since then: new forms of collaborative editing of spreadsheets

[4]http://www.scoop.it/t/basic-accounting-concepts/p/716103071/what-is-a-10-column-worksheet-in-accounting-click-here. (23-09-2013)

[5]http://pt.wikipedia.org/wiki/VisiCalc. (23-09-2013)

[6]http://office.microsoft.com/en-us/excel/. (27-10-2013)



```
C11   (L) TOTAL                               C!
                                              25
          A          B          C          D
 1      ITEM        NO.       UNIT       COST
 2      ----        ---       ----       ----
 3MUCK RAKE          43      12.95     556.85
 4BUZZ CUT           15       6.75     101.25
 5TOE TONER         250      49.95   12487.50
 6EYE SNUFF           2       4.95       9.90
 7                                  ---------
 8                        SUBTOTAL   13155.50
 9                 9.75% TAX          1282.66
10                                  ---------
11                        TOTAL      14438.16
12
13
14
15
16
17
18
19
20
```

Figure 1.6: VisiCalc spreadsheet system on an Apple II [5].

over the Internet were developed, like Microsoft Office 365[7], and Google Drive[8], and freeware alternatives were released, like LibreOffice Calc[9] and OpenOffice Calc[10], but Excel still holds the position as the most sold spreadsheet system today.

In spite of the huge evolution over the years, electronic spreadsheet systems still preserve the same basic interface established by VisiCalc. The data is presented in a tabular-like layout composed by cells, which can be individually referenced by a coordinate. Each coordinate is composed by a column, which is identified by a letter, and a row, which is identified by a number, for instance C3. Each cell can contain values or formulas, and a formula can reference a cell by the respective coordinate, or a range of cells , for instance, SUM(B3:B10).

1.2 Motivation

Spreadsheets are extensively used today in development of business applications. Estimates say that every year are produced tens of millions of spreadsheets worldwide [Panko and Ordway, 2008]. Besides being used to display data in a tabular-like interface they are also used collect information from different systems and to adapt data from one system to the format required by another [Cunha et al., 2012a].

[7]`www.microsoft.com/office365`. (27-10-2013)
[8]`https://drive.google.com/`. (27-10-2013)
[9]`http://www.libreoffice.org/features/calc`. (27-10-2013)
[10]`http://www.openoffice.org/product/calc.html`. (27-10-2013)

The fact that any person can exploit the full potential of a spreadsheet environment without any programming knowledge makes them a very attractive solution. Operating spreadsheets is very simple in nature: the structuring of data in a flexible two-dimensional tabular layout; basic operations like inserting and deleting of rows and columns; and direct data manipulation. These characteristics make it very intuitive and with a very soft learning curve. This flexibility, however, is also one of the main drawbacks of using this kind of environments: operators can insert data as they see fit but nothing prevents them from making mistakes. The lack of strict rules to define how a correct spreadsheet should be structured makes them very error-prone.

Some studies state that the vast majority, some of them going as high as 90%, of spreadsheets contain errors [Panko and Ordway, 2008; Panko, 2008; Powell and Baker, 2011; EuSpRiG]. What is more disturbing is that some of these spreadsheets are used in critical systems and can cause serious social, economic and political impact, as in the following examples:

- A missing minus sign in a spreadsheet formula caused the announced dividends to be distributed to Fidelity's Magellan fund shareholders to be off by $2.6 billion. What was being accounted as $1.3 billion of net capital gain was actually a net capital loss [Catless, 1995].

- Canadian power company TransAlta lost $24M caused by a simple copy and paste operation on a spreadsheet [Register, 2003].

- University of Toledo, in the United States, loses $2.4M in projected revenue caused by a mistake in a spreadsheet formula [Blade, 2004].

- Australian party, Country Liberal Party, was forced to admit, on the eve of an election, that their reported financial costs had a difference of tens of millions of dollars and attributed it to a spreadsheet error committed by their accounting firm [ABC, 2005]. The next day the main opposing party, the Australian Labor Party, won the election with 52.5% of the vote, while the Country Liberal Party fell behind with 35.3%.

- In the London 2012 Olympics, 10,000 tickets were oversold to the synchronized swimming sessions. The error was caused when a member of the staff typed the value 20,000, instead of 10,000 tickets remaining into a spreadsheet [Telegraph, 2012].

European Spreadsheets Risks Interest Group (EuSpRiG)[11] website has many other examples were the economic impact can range from a few thousands to billions of dollars.

1.3 Model-Driven Spreadsheet Engineering

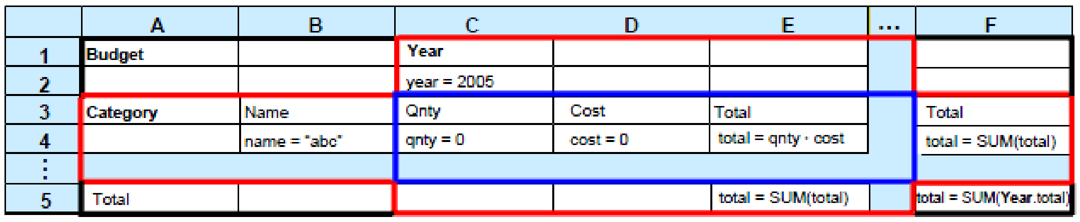
The need to reduce errors in spreadsheets has been acknowledged long ago by the scientific community, leding to the development of techniques and tools that endow end users with a Model-Driven Spreadsheet Development (MDSD) methodology. One of the first forms of model-driven spreadsheets was proposed in [Ireson-Paine, 1997]. Using this method, models were defined using a textual language that would be later processed by a compiler. The result was the generation of spreadsheets conforming to the specified model. This method had some limitations, as it only worked with spreadsheet models with database-like tables.

In [Erwig et al., 2005] the tool *Gencel* was introduced: it takes a model of a spreadsheet, dubbed template, and generates a spreadsheet conforming to that model. The consistency between model and instance is enforced by restricting the editing operations allowed to be carried out over the spreadsheet data. New operations, like add a set of columns with default cell values, are also added to help reduce errors. This method has some drawbacks: if an end user intends to update a model to reflect changes in a business model, this will break conformity with the instances.

In 2005 *ViTSL* [Abraham et al., 2005] was presented , along with a *ViTSL* editor. This allows to visually specify spreadsheets templates that can be passed to *Gencel* to generate the corresponding instances. This still has the same disadvantage of breaking conformity between model and instances when the model is updated.

Also in 2005 ClassSheets [Engels and Erwig, 2005] were proposed. This method is based on templates and uses concepts from Object-Oriented Development (OOD) and Model-Driven Development (MDD), namely classes and attributes. Using a Model-Driven Software Development (MDSD) method, an end user can specify a spreadsheet business model, as seen in Figure 1.7, and using a ClassSheet editor similar to Gencel, generate the spreadsheet that holds the actual data. The same limitations of the previous cases are still present.

[11]`http://www.eusprig.org/horror-stories.htm`. (19-09-2013)



	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 1.7: ClassSheet example using the ClassSheet editor, taken from [Engels and Erwig, 2005].

A method to derive models from spreadsheets, based in Model-Driven Engineering (MDE) concepts, was presented in [Hermans et al., 2010]. This allows to take advantage of patterns, usually found in spreadsheets, to generate the corresponding class diagrams.

Techniques for spreadsheet improvement were proposed in [Cunha, 2011], which allows, but is not limited, to migrate and refactor spreadsheets. This led to the development of HaExcel, a tool that, in some cases, improves spreadsheet productivity and reduces errors on spreadsheets, as concluded from the results of a study with end users [Beckwith et al., 2011a,b].

The HaExcel framework was later used to support ClassSheets evolution [Cunha et al., 2011], this is realized as a transformation system, with operations over models and instances, using the 2LT framework [Cunha et al., 2006; Cunha and Visser, 2007].

In [Cunha et al., 2012b] MDSheet was introduced: this tool allows for model specification in the same spreadsheet environment used for instance manipulation. It supports bidirectional transformations of models and instances, meaning that when a model changes, the instances are instantly co-evolved, and when an instance changes the model co-evolves automatically.

1.4 Model-Driven Evolution

Large software companies, banks, insurance corporations, and other enterprise class businesses rely on spreadsheets to store, disseminate, and adapt data produced by a system to be processed by a different one. As a result, spreadsheets are part of the decision-making process for these companies, which means that they must be continually updated to reflect changes on the business model. Usually this process consists of extracting information from a database to an intermediate file format, like a Comma-Separated Values (CSV) or Extensible Markup Language (XML) file that subsequently will be used to generate a corresponding spreadsheet, as described in Figure 1.8. Alternatively relational databases can be transformed into spreadsheets, and spreadsheets can be converted into databases [Cunha et al., 2009].

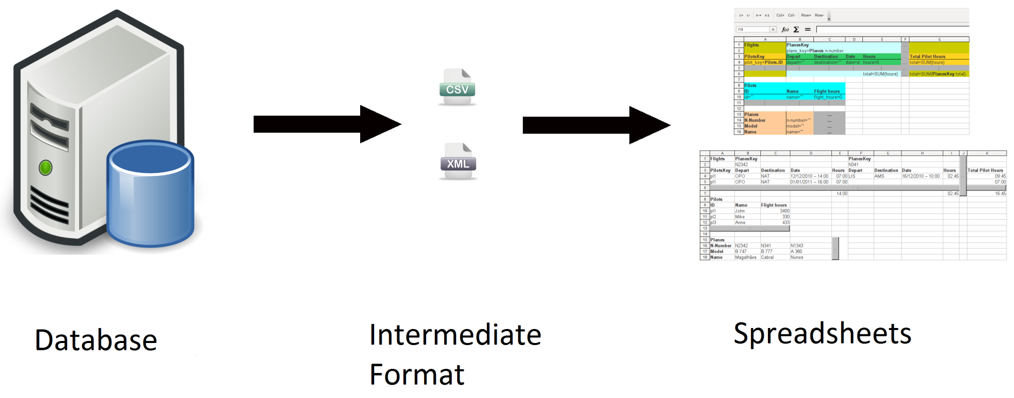


Figure 1.8: Data extraction and presentation flow.

In many real world applications of spreadsheets, different systems use different formats, so the data produced by one system has to be adapted to be consumed by a target system. Usually in these cases, both the spreadsheet initial format and final format, that is supposed to be consumed by the target system, are known in advance. The objective of this thesis is to take a step forward and propose techniques that allow the automatic evolution of spreadsheets that conform to an initial format so that they comply with a given final format.

1.5 Research Questions

The evolution method proposed, accomplishes ClassSheet model evolution at an automata level. To do this, the three following research questions must be answered.

- **Can ClassSheet models be expressed as automata?**
- **Can model evolution operations proposed in [Cunha et al., 2012a] be defined over finite automata?**

- **Can an initial ClassSheet model expressed as automaton be consecutively transformed, in a directed way, until it is equivalent to a given final automaton?**

From this point on, the focus of this thesis is to try to find affirmative answers to all these questions, as such it is divided in two major components, one theoretical and other practical.

The theoretical component is devoted to the study of model-driven evolution techniques. These techniques allow carrying out the transformation of an initial ClassSheet model to a final model known in advance. Models are represented by deterministic finite automatons, and initial work is developed to identify which transformation operations are to be applied to these automatons so that the final model can be explicitly attained.

The practical component consists on the development of a demonstration prototype capable of directed evolution of ClassSheet models and co-evolving the respective instances.

1.6 Document Structure

This thesis is structured as follows:

Chapter 2 Describes the state of the art of MDSD.

Chapter 3 Describes how ClassSheets will be represented as automata and how operations over ClassSheet models introduced in [Cunha et al., 2012a] are related to automata operations.

Chapter 4 Presents how an initial ClassSheet model can be evolved until a final model is attained, and how the data that conforms to the initial model can be evolved so it conforms to the final model.

Chapter 5 Describes the integration of the developed prototype with the MDSheet tool.

Chapter 6 Concludes this thesis mentioning the work done, and with some suggestions for future work.

Chapter 2

Model-Driven Spreadsheets

For a long time the spreadsheet research community and tool vendors have tried to mitigate spreadsheet errors by trying to introduce a set of guidelines and best practices, create spreadsheet templates, and develop specific tools to assist in spreadsheet application development, but they all center excessively on the low-level cell-oriented nature of spreadsheets [Engels and Erwig, 2005]. Spreadsheet development was still missing some well-established and proven software engineering principles like OOD [Engels and Groenewegen, 2000] and MDD [Kleppe, 2003].

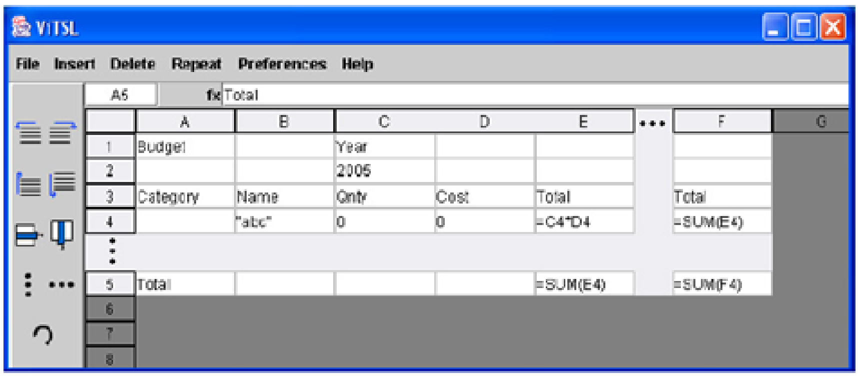
2.1 Templates

A first approach was presented to step up from the low level development previously used to a model level [Abraham et al., 2005], where a spreadsheet structure is described by grouping cells together to form blocks that can be repeated vertically or horizontally. This process consists of defining a model, designated template, by means of a visual editor called ViTSL [Abraham et al., 2005]. ViTSL offers four visual elements to describe templates:

- Cells, represented by rectangles that can contain labels, values or formulas;
- References, symbolizing concrete cell addresses;
- Vex groups, represented by vertical dots indicating the possibility of vertical expansion of a group of cells; and

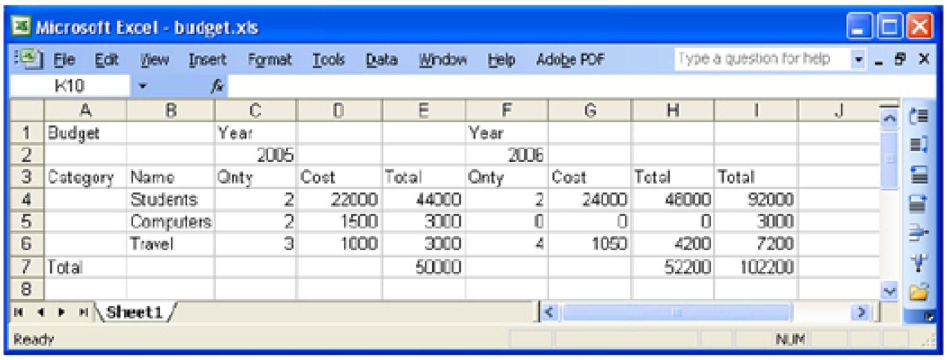
- Hex groups, represented by horizontal dots indicating the possibility of horizontal expansion of a group of cells;

The models created using ViTSL are then be passed as an argument to a tool, dubbed Gencel [Abraham et al., 2005], that generates spreadsheets according to the defined model, and constrain the type of operations allowed to make over the instances so that they always comply with the underlying model. In Figure 2.1 shows a template for a budget spreadsheet defined using the ViSTL editor. One can see that the block composed by the columns C, D and E can be repeated horizontally, and that line 4 can be repeated vertically. In Figure 2.2 an instance of that model, generated for Microsoft Excel, can be visualized. The block composed by columns C, D and E has, in fact, two occurrences, and the block composed by line 4 has three occurrences.



	A	B	C	D	E	…	F
1	Budget		Year				
2			2005				
3	Category	Name	Qnty	Cost	Total		Total
4		"abc"	0	0	=C4*D4		=SUM(E4)
⋮							
5	Total				=SUM(E4)		=SUM(F4)

Figure 2.1: A template defined using ViSTL.


Microsoft Excel - budget.xls
File Edit View Insert Format Tools Data Window Help Adobe PDF
Type a question for help
K10
Sheet1
Ready
NUM

	A	B	C	D	E	F	G	H	I	J
1	Budget		Year			Year				
2			2005			2006				
3	Category	Name	Qnty	Cost	Total	Qnty	Cost	Total	Total	
4		Students	2	22000	44000	2	24000	48000	92000	
5		Computers	2	1500	3000	0	0	0	3000	
6		Travel	3	1000	3000	4	1050	4200	7200	
7	Total				50000			52200	102200	
8										

Figure 2.2: Automatically generated Gencel spreadsheet.

Templates are an evolution over the old low-level development paradigm, and introduced model-driven development to spreadsheets, but had some limitations:

- The absence of connection between the modeling environment and the spreadsheet system prevented the synchronization between model and instances.
- Developers had to learn a different environment to be able to create the models.
- Were still error-prone because users could easily make mistakes grouping repeating blocks or wrongly introducing cell references.

2.2 ClassSheets

[Engels and Erwig, 2005] improved upon templates by introducing ClassSheets, a high-level model, based on object-oriented development, that allows describing spreadsheet business logic using concepts like classes and attributes. By using ClassSheets it is possible to group cells into logical units, identified by a name, that contain attributes with defined types, which closely match Unified Modeling Language (UML) classes. ClassSheet classes have some differences compared to UML classes though, namely, they can have labels, to help identify attribute names, formulas, and may be expandable, horizontally or vertically. In Figure 2.3 an example for an income sheet is shown. From an object-oriented stand point the ClassSheet is composed by a summation object, named **Income**, which aggregates a collection of objects containing a single data value, and named **Item**. From a layout point of view there is a list of data values extended by the header Item, representing the **Item** class, which in turn is embedded into the summation object that consists of a header Income and a footer with a label Total and an aggregation formula assigned to an attribute total. Figure 2.3 also shows the corresponding UML class diagram.

As can be observed, one great advantage over templates is that cell references are no longer used. Instead, attribute names are utilized in formulas, making ClassSheet instances a lot less error-prone than templates. In combination with the visual representation, a formal definition was also introduced.

This formal representation allows to specify ClassSheet models textually using the abstract syntax presented in Figure 2.4. Using this syntax, a ClassSheet model can be defined in the following way:

- A Sheet is a composition of fixed or horizontally composed classes.

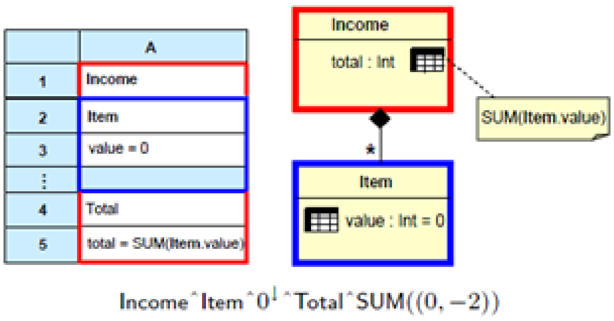


Figure 2.3: Income ClassSheet and corresponding UML class diagram.

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

Figure 2.4: ClassSheet Syntax.

- A Class is composed by fixed or vertically expandable blocks of cells.
- Each of the cells can be comprised of values and attributes or possibly empty.
- Attributes have a name (a) and a formula (f) defining its value, that can be referenced by using qualified attribute names ($n.a$).
- Each block related with a class is identified with a label (l), that can be a row (n), a column ($|n$), a table ($|\underline{n}$) or a cell class label ($.n$).

Using the ClassSheet model represented in Figure 2.3 it is possible to give a better understanding of how the formal definition can be used to describe ClassSheets.

Suppose we want to represent the **Item** class of the spreadsheet. That portion of the spreadsheet is composed by two cells, one with the value Item, used as a header, and the other one with a vertically repeated attribute named value. By labeling that two cell blocks with the value Item and marking it with a vertically repeatable label, that block is to be perceived as a vertically repeatable class, and it is represented by the following syntax:

|**Item** : Item ˆ

|**Item** : (value=0)↓

Next, the vertically expandable class **Income** is included. This class has a particularity, when compared to the **Item:**, the header cell is separated from the two cells in the bottom. To solve this problem, a vertical expandable **Income** class identifier is included before the two lowermost cells.

|**Income** : Income ˆ

|**Item** : Item ˆ

|**Item** : (value=0)↓ ˆ

|**Income** : Total ˆ

|**Income** : total=SUM(Item.value)

The above example is the complete textual representation of the example depicted in Figure 2.3. This syntax will have an important role in ClassSheet model evolution, as it will be based on it that automata will be generated and evolved (see Chapter 3).

The abstract syntax by itself has some limitations, as it does not enforce structural constraints caused by the two dimensional layout. For instance, it is possible to define the following sheet [Engels and Erwig, 2005].

|**Income** : Income ˆ

|**Item** : Item ˆ

|**Income** : (value=0)↓ ˆ

|**Item** : Total ˆ

|**Income** : total=SUM(Item.value)

This would imply that it would be possible to have an **Income** class that aggregates a class **Item** which in turn aggregates the class **Income**. This is not possible, and as such should not be considered a ClassSheet [Engels and Erwig, 2005].

To enforce a valid ClassSheet spatial structure, called tiling, a type system was formalized. With this system it is possible to define the four main tiling structures, i) non-aggregated single classes, ii) one-dimensional horizontally expandable aggregated classes, iii) one-dimensional vertically expandable aggregated classes an iv) two-dimensional aggregations. Aggregation tiles can also be nested and tiles can be horizontally or vertically composed. Tiling rules are presented in Figure 2.5.

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

Figure 2.5: ClassSheet tiling rules.

In Figure 2.6 are depicted five examples of correct tiling. From left to right, there a simple sheet, a vertical one-dimensional aggregation, a horizontal aggregation, a vertical aggregation over two vertical aggregations, and a two-dimensional aggregation.

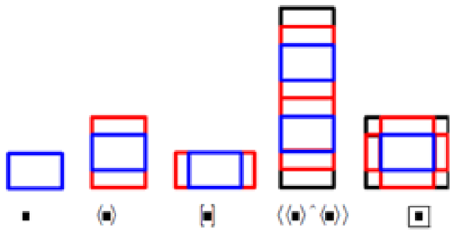


Figure 2.6: ClassSheet tiling structures.

Although ClassSheets are a great enhancement over templates, the process used to create a model, initially, was essentially the same. The ClassSheet model had to be produced using a ClassSheet editor and, after that, another tool had to be used to generate instances that comply with that model.

2.3 Relational ClassSheets to UML+OCL

A transformation method between ClassSheets and UML class diagrams, enriched with Object Constraint Language (OCL) constraints, was presented in [Cunha et al., 2012d]. This allowed the application of model validation techniques, already avaiable for UML, to spreadsheets. The aforementioned was accomplished by extending ClassSheets to support characteristics of relational databases, namely the existence of primary and foreign keys, associated with ClassSheet tables. The proposed extensions are shown in Figure 2.7 marked in red.

The new φ^{μ} extension, allows the representation of unique values within a column or row, and these values are to be perceived as primary keys in database tables. The $n.a$ extension

$$\begin{array}{llll} f \in Fml & ::= & \varphi \mid n.a \mid \varphi(f, \ldots, f) & (formulas) \\ b \in Block & ::= & \varphi \mid \varphi^{\mu} \mid a = f \mid b \shortmid b \mid b\,\hat{}\,b & (blocks) \end{array}$$

Figure 2.7: ClassSheet extensions, from [Cunha et al., 2012d].

has a stronger meaning than in the original ClassSheet syntax and it is to be perceived in the same way as a foreign key in a database model. Each time a declaration $n.a$ is defined, the existence of class n and attribute a is verified, before it is accepted. Also attribute a has to be declared as a primary key, i.e., as unique in class n. These two extensions ensure that the values in cells with both types of keys are always valid.

Mapping ClassSheets into UML classes with OCL constraints, using the proposed method in [Cunha et al., 2012d], is then a three step process. First UML classes are generated from ClassSheet classes, subsequently the associations between classes are inferred, and finally OCL code is generated to ensure that both representations possess the same properties, namely restrictions on primary and foreign keys. These specifications are generated in the notation for UML-based Specification Environment (USE), proposed in [Gogolla et al.], which is a tool with UML execution and OCL support, enabling for instant validation of spreadsheet models.

In Figure 2.9 a ClassSheet model for an airline company is displayed. Such model is composed by three main classes; **Pilots**, with primary key id, **Planes**, with primary key n-number, and **Flights**. The **Flights** class is in turn composed by three classes; **PlanesKey**, which references **Planes** by the n-number attribute, **PilotsKey** , that references **Pilots** by the id attribute, and an association class between the **PlanesKey** and **Pilots** classes. Extracting the UML class diagram using the techniques presented in [Cunha et al., 2012d], results in the diagram of Figure 2.8, as drawn by the USE tool.

The inverse process, which is generating ClassSheet models from UML class diagrams, is straightforward, and allows to broaden the range of possible evolutions. This is crucial in the context of the work carried out in this thesis.

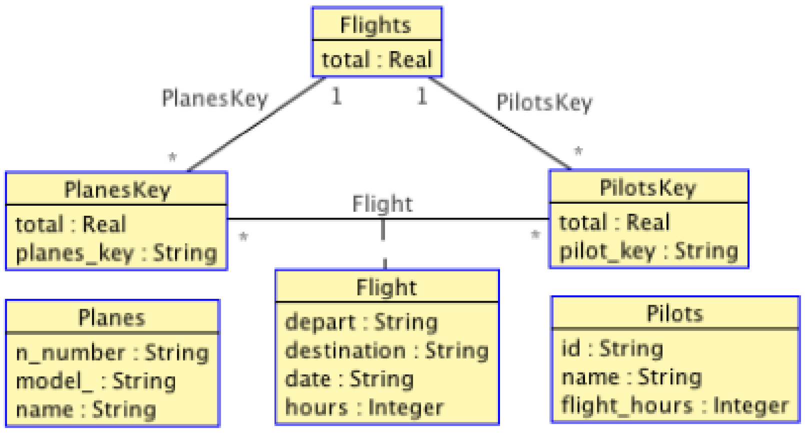


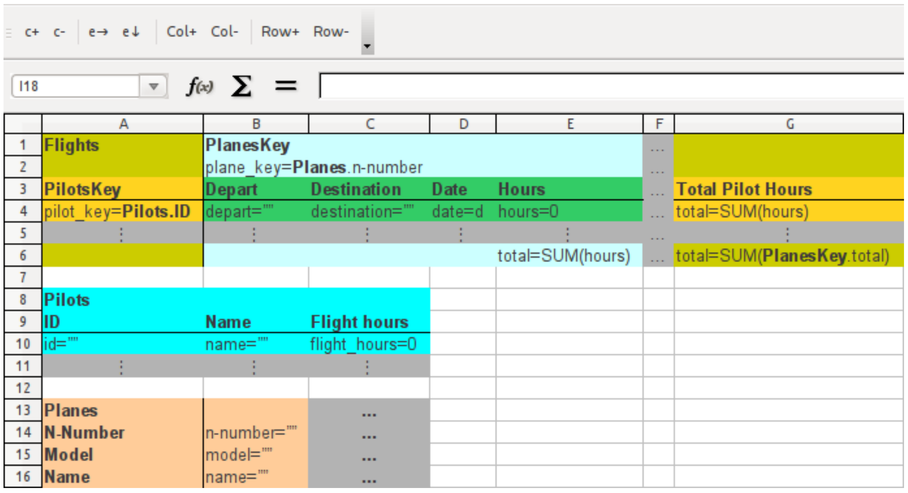
Figure 2.8: Flights UML class diagram, as presented in [Cunha et al., 2012d].

2.4 Embedding of ClassSheet Models Within Spreadsheets

In [Mendes, 2011] a method to embed ClassSheet model specifications in spreadsheet systems was proposed. Embedded ClassSheets are visually similar to normal ClassSheets, except that classes are represented with solid colors and repetition rows and columns are explicitly represented on the spreadsheet.

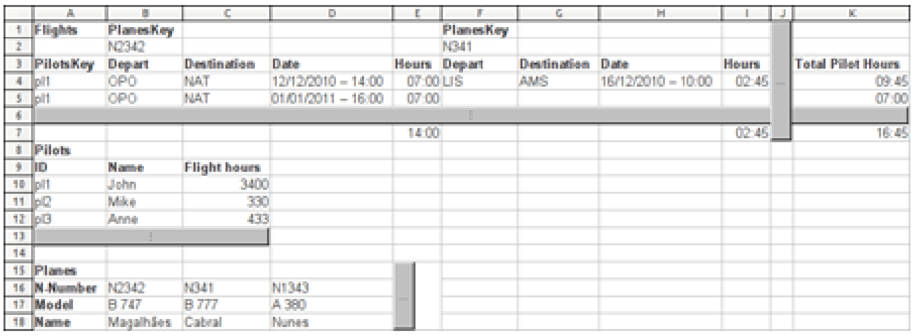
Also, new techniques were introduced to evolve ClassSheet models and instances so that they both stay synchronized when changes are applied. This brought two major advantages: (1) allowed the users of the spreadsheet system to work with both, model and instances, without having to learn a different environment, and (2) simplified the architecture of the software system responsible to keep models and instances synchronized.

In Figure 2.9 we can observe how ClassSheet models can be specified in a spreadsheet environment almost in the same way as in ViTSL. In order to create the Pilots class, a user had to group a block of cells and select an option to add a class, naming it Pilots and assign the label values **ID**, **Name** and **Flight Hours** to the respective cells. A second vertically expandable class had to be created and the id, name and flight_hours attributes had to be assigned to the respective cells. The conforming model, where we can see three occurrences of the vertically expandable class data, is shown in Figure 2.10.



	A	B	C	D	E	F	G
1	**Flights**	**PlanesKey**				…	
2		plane_key=**Planes**.n-number				…	
3	**PilotsKey**	**Depart**	**Destination**	**Date**	**Hours**	…	**Total Pilot Hours**
4	pilot_key=**Pilots.ID**	depart=""	destination=""	date=d	hours=0	…	total=SUM(hours)
5	⋮	⋮	⋮	⋮	⋮	…	⋮
6					total=SUM(hours)	…	total=SUM(**PlanesKey**.total)
7							
8	**Pilots**						
9	**ID**	**Name**	**Flight hours**				
10	id=""	name=""	flight_hours=0				
11	⋮	⋮	⋮				
12							
13	**Planes**		…				
14	**N-Number**	n-number=""	…				
15	**Model**	model=""	…				
16	**Name**	name=""	…				

Figure 2.9: Embedded ClassSheet representing a system for an airline company as proposed in [Mendes, 2011].



	A	B	C	D	E	F	G	H	I	J	K
1	**Flights**	**PlanesKey**				**PlanesKey**					
2		N2342				N341					
3	**PilotsKey**	**Depart**	**Destination**	**Date**	**Hours**	**Depart**	**Destination**	**Date**	**Hours**		**Total Pilot Hours**
4	pl1	OPO	NAT	12/12/2010 – 14:00	07:00	LIS	AMS	16/12/2010 – 10:00	02:45		09:45
5	pl1	OPO	NAT	01/01/2011 – 16:00	07:00						07:00
6											
7					14:00				02:45		16:45
8	**Pilots**										
9	**ID**	**Name**	**Flight hours**								
10	pl1	John	3400								
11	pl2	Mike	330								
12	pl3	Anne	433								
13											
14											
15	**Planes**										
16	**N-Number**	N2342	N341	N1343							
17	**Model**	B 747	B 777	A 380							
18	**Name**	Magalhães	Cabral	Nunes							

Figure 2.10: Spreadsheet instance for an airline company as proposed in [Mendes, 2011].

2.5 Bidirectional Transformation of Spreadsheets

A co-evolution technique, based on bidirectional transformations, was proposed in the context of the SSaaPP[1] project to synchronize ClassSheet models and instances [Cunha et al., 2012a]. In this approach two different sets of operations, one corresponding to editing operations over ClassSheet models, and other corresponding to editing operations over instances, were developed and bound together by means of a symmetrical bidirectional framework. The purpose of this method is to allow evolutions over models to be transformed in evolutions over instances of that model and vice-versa. The principle behind it is that every time a model or instance is changed, that change has to be reflected not only on the entity that changed but also on all of the others, in order to restore conformity, as illustrated in Figure 2.11. This bidirectional framework is implemented in MDSheet

[Cunha et al., 2012b].

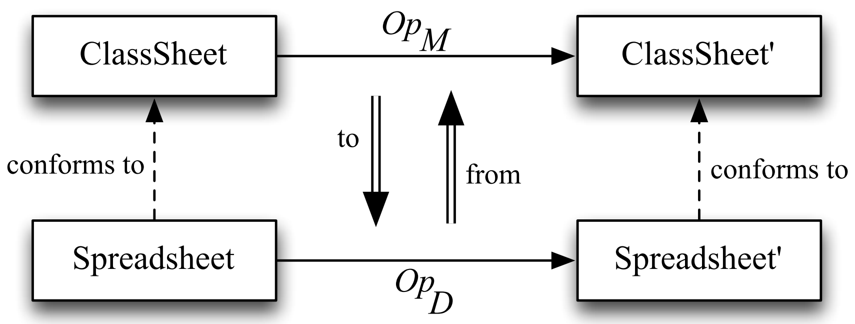


Figure 2.11: Spreadsheet bidirectional transformation system, as proposed in [Mendes, 2011].

The set of edit operations defined over models, and displayed in Figure 2.12, is of paramount importance to the work carried out in this thesis, as the same operations are implemented at automata level so that automatic evolution can be achieved.

```
data OpM : Model → Model =
    addColumnM    Where Index                                -- add a new column
  | delColumnM          Index                                -- delete a column
  | addRowM       Where Index                                -- add a new row
  | delRowM             Index                                -- delete a row
  | setLabelM     (Index, Index) Label                       -- set a label
  | setFormulaM   (Index, Index) Formula                     -- set a formula
  | replicateM    ClassName Direction Int Int                -- replicate a class
  | addClassM     ClassName (Index, Index) (Index, Index)    -- add a static class
  | addClassExpM  ClassName Direction (Index, Index) (Index, Index)
                                                             -- add an expandable class
```

Figure 2.12: Operations over ClassSheet models, as proposed in [Cunha et al., 2012a].

Even with all that was presented before, spreadsheet model evolution is still a manual process. Users still have to directly manipulate models or instances so that changes take effect, which still makes it, although to a lesser extent, error-prone. On chapters 3 and 4 a method to automatically evolve ClassSheet models and co-evolve the respective instances is proposed, building upon what was just presented.

[1]http://ssaapp.di.uminho.pt/twiki/bin/view/Main/. (23-09-2013)

2.6 Summary

In this chapter an extended overview about MDSD was made, to establish the grounds on which this thesis is based.

We started by reviewing templates, this models are a first step up from the low level cell oriented development previously used.

Afterwards ClassSheets were addressed: this method builds upon templates and allows the specification of a spreadsheet business model using OOD constructs, like classes and attributes. The fact that cell references are substituted by attribute names makes them considerably less error-prone than templates.

Finally, the embedding of ClassSheet models in the same environment used to manipulate the instances, and the bidirectional framework used to keep everything synchronized was presented.

Chapter 3

Operations Over ClassSheet Automata

Methods for model-driven spreadsheet evolution were already proposed [Cunha et al., 2012a], however this evolution had to be done manually, and a human operator had to apply successive transformations to the spreadsheet model until it reached the desired final state. This can be time consuming and lead to errors. In this chapter we present the first techniques that allow automatically evolving a ClassSheet model to a desired final model while, at the same time, migrating the data that conforms to the initial model so it conforms to the final model. The method proposed in this document to create an automatic evolution framework for ClassSheet models is based on automata transformation and equivalence, we chose automata mainly for two reasons. First, it is a formal way of representing information. Second, ClassSheets already provide the necessary tool to transition from model level to automata level evolution. Their formal, textual, description defines a regular language and thus, by formal definition, ClassSheets can be expressed by a finite automaton. One possible representation for the **Item** class in the ClassSheet presented in Figure 2.3 is shown in Figure 3.1.

This representation is translated directly from the ClassSheet formal syntax and describes the ClassSheet in the following way: the first transition works as the identification of the vertically expandable class **Item**, which indicates that the following content belongs to class **Item**, next is the content of the first cell of the ClassSheet, which holds the value Item, followed by a vertical composition (ˆ). On the second row appears once again the

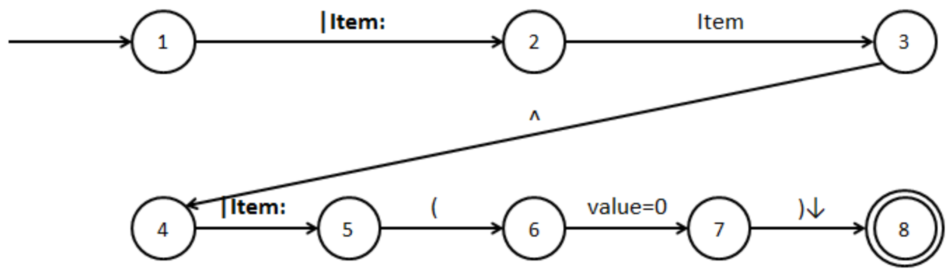
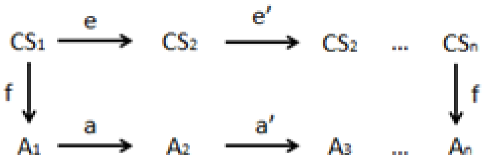


Figure 3.1: Item class expressed as an automaton.

identification of the class followed by the open parenthesis, which means that all transitions that follow, until the closing parenthesis, are considered repeatable content. In this case the value attribute is vertically repeatable, as can be witnessed by the last transition.



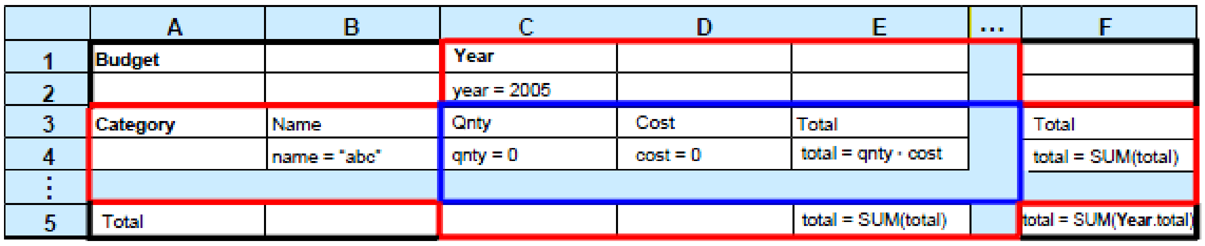
$$\begin{tikzcd} CS_1 \arrow[r, "e"] \arrow[d, "f"'] & CS_2 \arrow[r, "e'"] & CS_2 & \dots & CS_n \arrow[d, "f"] \\ A_1 \arrow[r, "a"] & A_2 \arrow[r, "a'"] & A_3 & \dots & A_n \end{tikzcd}$$

Figure 3.2: Evolution process.

To achieve our goal for automatic evolution, the first step is to establish a conversion method between ClassSheet models and deterministic finite automata, so that evolution can take place at automata level. After that, equivalent operations over ClassSheet models proposed in [Cunha et al., 2012a] have to be defined over automata. Subsequently a technique for model evolution has to be applied so that, given initial and final automata, the initial automaton can be transformed until it is equivalent to the final automaton. In Figure 3.2 one can visualize how the expected evolution process will progress. First, the initial model (CS1) has to be converted to an automaton (A1) using an appropriate conversion function (f). Transformation operations at automata level (a^x) are then continually applied until the final model is attained (An). The important thing to notice is that these operations are directly related to operations defined over ClassSheet models (e^x) in [Cunha et al., 2012a]. The main objective of this process is to identify the sequence of operations needed to apply to the first automaton, to transform it into the final one. When this sequence of operations is determined, migrating the data that complies with the initial ClassSheet model (CS1) can be done by using bidirectional transformations defined in [Cunha et al., 2012a].

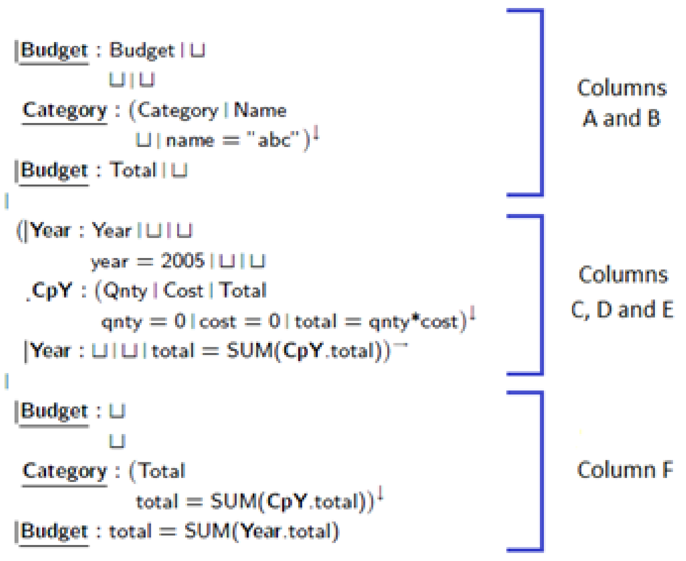
3.1 ClassSheets as Automata

As already observed, representing ClassSheets as automata can be achieved by converting the formal language syntactic elements to automata transitions; however this process has drawbacks. When representing a ClassSheet model with a horizontal aggregation over a relation, or with two-dimensional aggregations, a difficulty arises that makes it impossible to process a ClassSheet automaton in row-by-row manner. A ClassSheet with a two-dimensional aggregation, and corresponding formal definition, is presented in Figure 3.3 and Figure 3.4, respectively.



	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 3.3: Two-dimensional example of a ClassSheets, as seen in [Engels and Erwig, 2005].



```
|Budget : Budget | ⊔
          ⊔ | ⊔
 Category : (Category | Name
              ⊔ | name = "abc")↓
|Budget : Total | ⊔
|
(|Year : Year | ⊔ | ⊔
        year = 2005 | ⊔ | ⊔
 .CpY : (Qnty | Cost | Total
         qnty = 0 | cost = 0 | total = qnty*cost)↓
 |Year : ⊔ | ⊔ | total = SUM(CpY.total))→
|
|Budget : ⊔
          ⊔
 Category : (Total
              total = SUM(CpY.total))↓
|Budget : total = SUM(Year.total)
```

Columns A and B
Columns C, D and E
Column F

Figure 3.4: Textual representation of the Budget ClassSheet.

To formally define the budget ClassSheet model in Figure 3.3, one has to describe the model as horizontally composing blocks; these blocks are identified in Figure 3.4. The first block is composed by columns A and B and belongs to the root class **Budget** and an aggregation class **Category**. The middle block is composed by columns C,D and E, and belongs to the column aggregation class **Year** and **Category**, which surround the association class **.Cpy**. The final block is composed by column F and, like the first block, belongs to classes **Budget** and **Category**. The problem with this representation is that the first block is defined from top to bottom, and then the middle and third blocks are defined sequentially in the same way. For reasons that will become clear in Chapter 4 a different representation, still based on the ClassSheet formal representation, is required. This representation allows sequentially defining a ClassSheet model, row-by-row, starting on the upper-left corner and ending on the lower-right corner. In this thesis we are considering embedded ClassSheets, and as such, vertical and horizontal expansion symbols are also included on the new representation.The new representation is attained by re-writing the ClassSheet using the transformation function (τ) presented next.

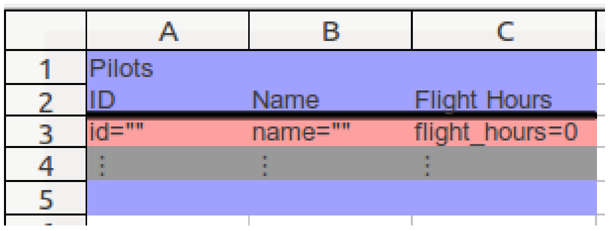
$$
\begin{array}{l}
\tau : ClassSheet \rightarrow TClassSheet \\
\tau\ f = f \\
\tau\ \varphi = \varphi \\
\tau\ (a = f) = (a = f) \\
\tau\ ((l_1 : (c_1\,\hat{}\,c_2))\,|\,(l_2 : (c_3\,\hat{}\,c_4) \rightarrow)) = \tau(l_1) : \tau(c_1)\,|\,(\tau(l_2) : \tau(c_3)\,|\,\cdots) \rightarrow |\,\tau(c_2|c_4) \\
\tau\ (b_1|b_2) = \tau(b_1)\,|\,\tau(b_2) \\
\tau\ (b_1\,\hat{}\,b_2) = \tau(b_1)\,\hat{}\,\tau(b_2) \\
\tau\ .n = .n \\
\tau\ \underline{n} = n \\
\tau\ |\underline{n} = n \\
\tau\ |n = n \\
\tau\ (l : b) = \tau(l) : \tau(b) \\
\tau\ (l : b \downarrow) = \tau(l) : (\tau(b)\,\hat{}\,\tau(l) :\ rep(\,\vdots\,, width(b))\ ) \downarrow \\
\tau\ (l : b_1\,\hat{}\,b_2) = (\tau(l) : \tau(b_1))\,\hat{}\,(\tau(l) :\ \tau(b_2)) \\
\tau\ (c_1\,\hat{}\,c_2) = \tau(c_1)\,\hat{}\,\tau(c_2) \\
\tau\ (c \rightarrow) = \tau(c) \rightarrow \\
\tau\ (s_1|\,s_2) = \tau(s_1)|\,\tau(s_2)
\end{array}
$$

The fifth and the last case look the same but as it happens in the original ClassSheet language the same vertical bar is used to compose blocks and sheets. In the case for $l : b \downarrow$

the *rep* function takes as arguments a symbol and a number of columns, and horizontally composes the symbol in that amount of times.

To better understand how this representation can be used to represent ClassSheet models as automata two simple examples will ensue.

The first example (Figure 3.5) is a ClassSheet model with a root class named **Pilots** with three labels with values, ID, Name and Flight Hours, respectively. This class aggregates a vertically expandable class, here named **PilotsA**, with three attributes; id, name and flight_hours.



	A	B	C
1	Pilots		
2	ID	Name	Flight Hours
3	id=""	name=""	flight_hours=0
4	⋮	⋮	⋮
5			

Figure 3.5: Pilots ClassSheet.

This ClassSheet formal representation is shown next.

|**Pilots:** Pilots | ⊔ | ⊔ ˆ
|**Pilots:** ID | Name | Flight Hours ˆ
|**PilotsA:** (id=”” | name=”” | flight_hours=0)↓ ˆ
|**Pilots:** ⊔ | ⊔ | ⊔

The row-by-row textual specification of this ClassSheet model, after applying the transformation function previously explained, can be realized in the following way.

Pilots: Pilots | ⊔ | ⊔ ˆ
Pilots: ID | Name | Flight Hours ˆ
PilotsA: (id=”” | name=”” | flight_hours=0 ˆ
PilotsA: ⋮ | ⋮ | ⋮)↓ ˆ
Pilots: ⊔ | ⊔ | ⊔

The first row starts with the **Pilots** class which contains a block composed by three horizontally composed (|) cells. The first cell has a label identifying the name of the class, in this case Pilots, followed by two empty cells (⊔), the row ends with the vertical composition (ˆ) of rows. The second row has a similar structure: it starts with the **Pilots** class and has a vertical composition of three cells with labels, ID, Name and Flight Hours, used to identify attributes of the **PilotsA** class; like the first one, it ends with a vertical composition. The third row has some differences, compared to the first two; it starts with the aggregated class **PilotsA**, which contains a vertically repeatable block (()↓), which can be divided into two vertically composed blocks, of these two blocks, only the first one is contained in the third row. This block is a horizontal composition of three cells that hold the id, name and flight_hours attributes of the **PilotsA** class, the row ends with a vertical composition of blocks (ˆ). The fourth row holds the second block of the previous vertical composition, containing a horizontal composition of three cells, and each one of these cells marks the vertical repeatability of the class (⁝). The fifth and final row starts with the class **Pilots** that contains a block consisting of three horizontally composed empty cells. This concludes the row-by-row description of the **Pilots** ClassSheet.

With the new representation, converting the textual definition to an automaton is a straightforward task; each language symbol can be converted into an automaton transition. To generate the complete automaton for a ClassSheet model, each transition must be sequentially connected to one another in the same order as the symbols appear in the textual definition.

The resulting automaton, attained from converting the **Pilots** ClassSheet model from textual representation, can be visualized in (Figure 3.6). As does the representation, automata representation also is a row-by-row description of a model, and all the symbols have the same meaning as in the former.

The previous example allows observing how it is possible to attain a ClassSheet automaton from its textual representation but it does not allow to understand how the new representation can be used to specify a horizontally expandable class. To address this, the ClassSheet pictured in Figure 3.7 is presented. This ClassSheet is composed by a class named **Planes** that possesses four labels with contents, Planes, N-Number, Model and Name, respectively, and aggregates a horizontally expandable class named PlanesA that holds three attributes, n-number, model and name.

The formal representation of this ClassSheet model is presented next.

<u>Planes</u> Planes ˆ
<u>Planes</u>: N-Number ˆ
<u>Planes</u>: Model ˆ

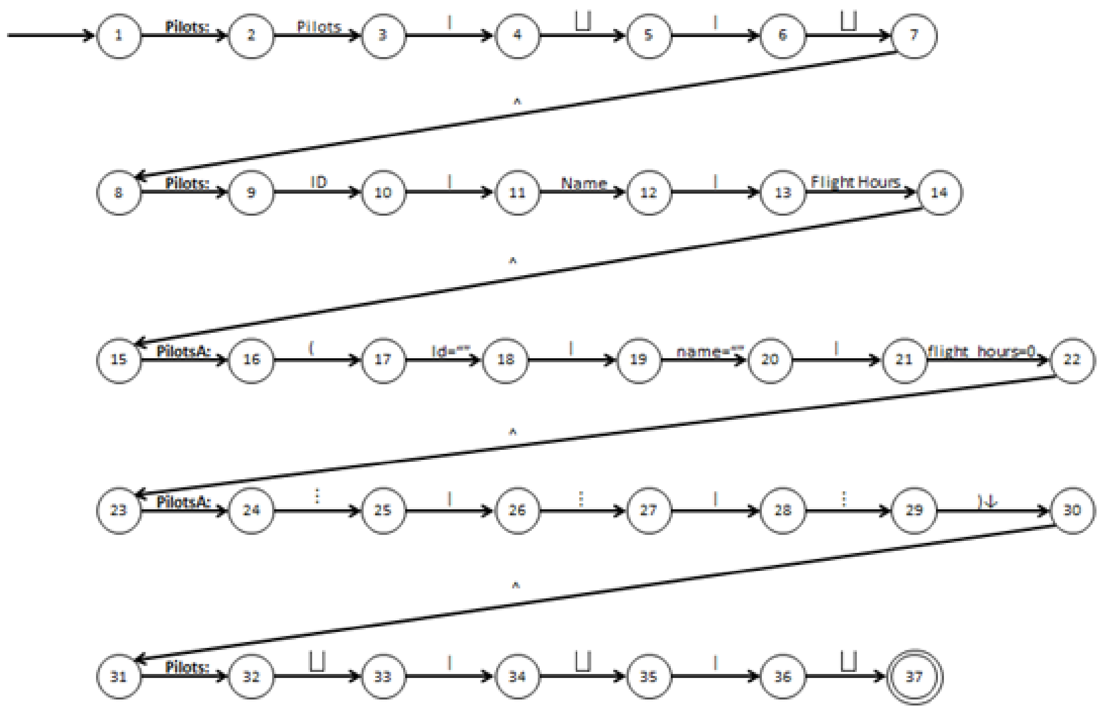
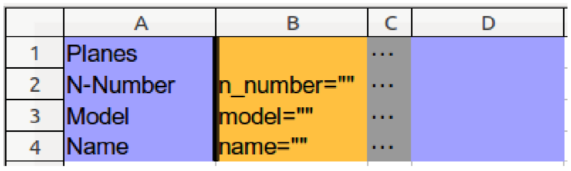


Figure 3.6: Pilots ClassSheet automaton.



	A	B	C	D
1	Planes		···	
2	N-Number	n_number=""	···	
3	Model	model=""	···	
4	Name	name=""	···	

Figure 3.7: Planes ClassSheet.

<u>Planes</u>: Name ^

|

(**<u>PlanesA</u>:** ⊔ ^

<u>PlanesA:</u> n-number="" ^

<u>PlanesA:</u> model="" ^

<u>PlanesA:</u> name="")→

|

<u>Planes:</u> ⊔ ^

<u>Planes:</u> ⊔ ^

<u>Planes:</u> ⊔ ^

<u>Planes:</u> ⊔ ^

To textually specify the **Planes** ClassSheet, in a row-by-row way, the aggregated class **PlanesA** has to be divided in multiple horizontally expandable sub-blocks, with each sub-block belonging to a single row. Below is the textual definition of the **Planes** ClassSheet.

Planes: Planes | **PlanesA:** (⊔ | · · ·)→ | **Planes:** ⊔ ˆ
Planes: N-Number | **PlanesA:** (n-number=”” | · · ·)→ | **Planes:** ⊔ ˆ
Planes: Model | **PlanesA:** (model=”” | · · ·)→ | **Planes:** ⊔ ˆ
Planes: Name | **PlanesA:** (name=”” | · · ·)→ | **Planes:** ⊔ ˆ

The first row is defined as a horizontal composition of classes; it initiates with the **Planes** class, that possesses a cell with a label Planes, this class horizontally composes with the class **PlanesA**, that holds a horizontally expandable block (()→), this block contains two cells, one empty, and another indicating the expandability of the class (· · ·), the final part of the composition is poised by the class **Planes** which holds an empty cell. The following rows are described exactly in the same way, the important thing to emphasize is that horizontally expandable classes, in this case **PlanesA**, are defined as separated horizontally expandable blocks. Like the **Pilots** ClassSheet, the automaton can be attained by simply converting the formal language symbols into automaton transitions, the result can be visualized in Figure 3.8.

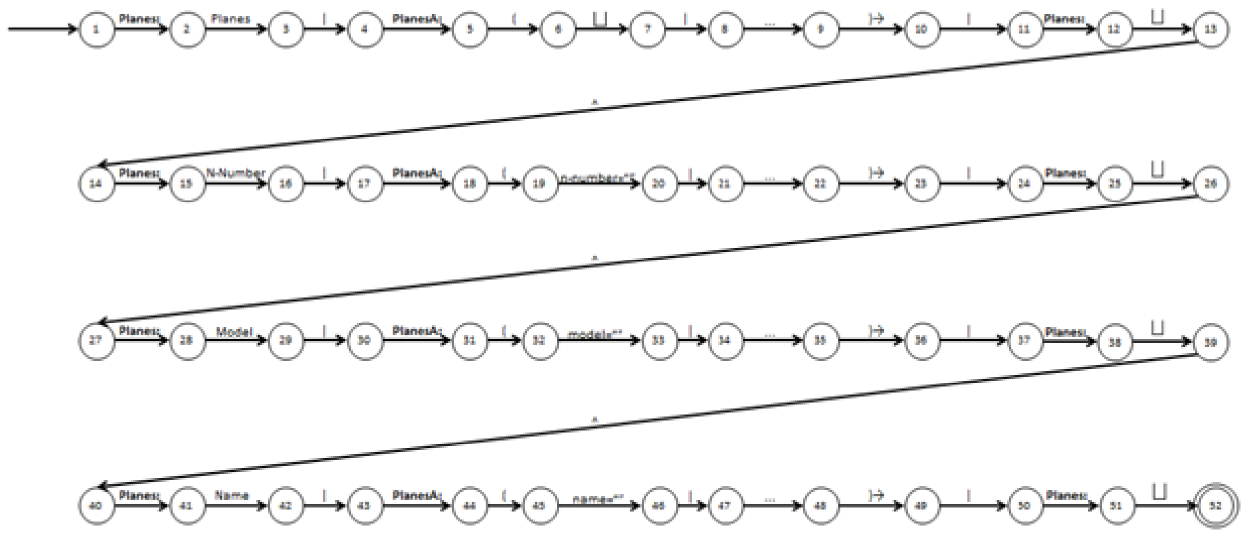


Figure 3.8: Planes ClassSheet automaton.

With the transformation function previously presented is possible to attain an automaton representing a ClassSheet model. The conversion between ClassSheet and Deterministic Finite Automata (DFA) is shown in Algorithm 1. The first step is to transform the ClassSheet textual representation to a row-by-row description. This description is then utilized to generate the vocabulary of the automaton. This is done by collecting all language symbols in the new textual representation. Next, the transitions table is generated by connecting all language symbols in the same order as they appear in the representation. Using the transitions table it is possible to determine the set of states by obtaining every state in all transitions. The following step is to identify the initial state, this is done by finding the only state not present as destiny on any transition. The set of final states is determined by finding all states not present as origin in any transition. Finally an automaton is generated using the parameters previously computed.

Algorithm 1 ClassSheet to DFA Conversion Function

```
1: function CLASSSHEETTODFA($classsheet$)
2:     $transformedCS \leftarrow \tau(classsheet)$
3:     $vocabulary \leftarrow$ getAllSymbols($transformedCS$)
4:     $transitionsTable \leftarrow$ generateTT($transformedCS$)
5:     $states \leftarrow$ getStates($transitionsTable$)
6:     $initialState \leftarrow$ getInitialState($transitionsTable$)
7:     $finalStates \leftarrow$ getFinalStates($transitionsTable$)
8:     return genDFA($vocabulary, states, initialState, finalStates, transitionsTable$)
9: end function
```

3.2 Basic Operations over Automata

Before model operations are defined over ClassSheet automata, five basic operations are necessary. These operations are used in conjunction to produce higher level, ClassSheet model operations. The description of the five basic operations follows. To aid in the description of the operations we will use as an example the automaton in Figure 3.9.

3.2.1 Add Transition

The first operation we introduce adds a transition to an automaton, given the state where the transition begins, the state where the transition ends, and the symbol that allows the transition between states to take place. The result of adding a transition between states 2 and 3, by the symbol e, to the automaton pictured in Figure 3.9 is presented in Figure 3.10.

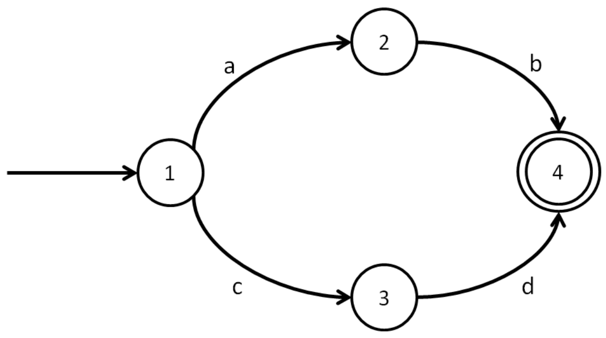


Figure 3.9: Generic automaton.

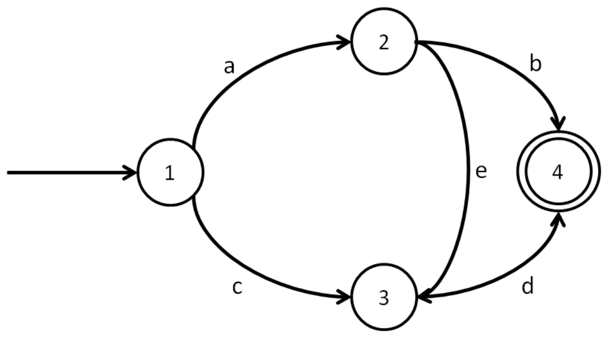


Figure 3.10: Generic automaton with a new transition.

3.2.2 Delete Transition

This operation deletes a transition from an automaton, given the state where the transition begins, the state where the transition ends and the symbol of the transition. The result of deleting the transition between states 2 and 3 by the symbol e, from the automaton in Figure 3.10 results in the automaton pictured in Figure 3.9.

3.2.3 Edit Transition

This operation allows to modify the states and symbol components of a transition. Given a transition to be modified and the new values of the three components, it updates the transition so it reflects the new values. The result of altering the transition between states 2 and 3, by the symbol e, in the automaton in Figure 3.10 to connect states 3 and 1, by symbol f, is presented in Figure 3.11.

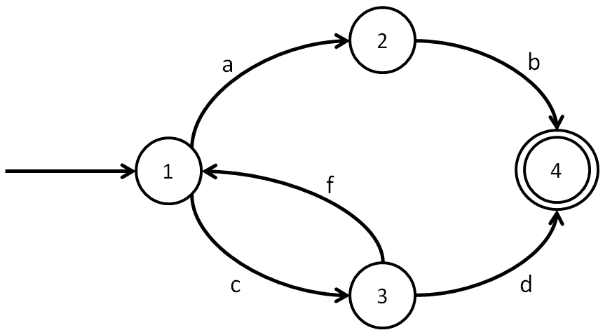


Figure 3.11: Generic automaton with an edited transitions.

3.2.4 Add State

Given a new state, this operation adds it without any transitions to an automaton. Adding a state to the automaton in Figure 3.9 results in the automaton in Figure 3.12.

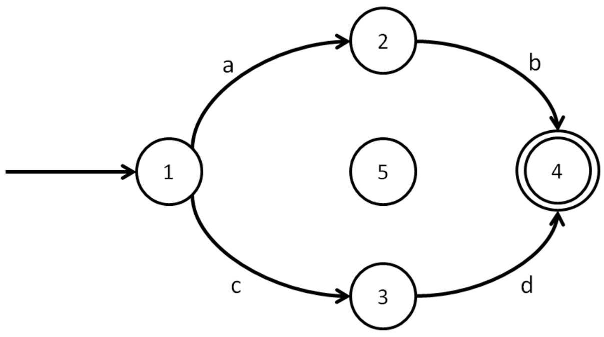


Figure 3.12: Generic automaton with a new state.

3.2.5 Remove State

The last operation removes a state from an automaton, and all transitions associated with that state using the remove transition operation previously defined. The result of removing state 2 from the automaton in Figure 3.9 results in the automaton in Figure 3.13.

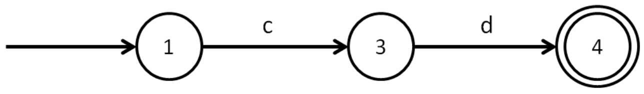


Figure 3.13: Generic automaton with a state removed.

3.3 ClassSheet Automata Operations

In this section, equivalent operations over ClassSheet models proposed in [Cunha et al., 2012a] are presented over ClassSheet models expressed as automata. These are established on top of the basic operations previously defined. The process of applying a ClassSheet model operation over an automaton is depicted in Figure 3.14. A ClassSheet automata operation (a), that transforms an automaton (A_1) in an automaton (A_n), is a sequence of basic operations (b^x) that are consecutively applied as a single, atomic, operation. On the other hand, an automata operation (a) conforms to ClassSheet operation (e) and, as such, the outcome of applying operation (a) to automaton (A_1) yields the same result as applying operation e to the ClassSheet (CS_1), and converting the result to automaton, both resulting in automaton (A_n).

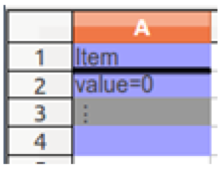
$$\begin{tikzcd}
CS_1 \arrow[rrrr, "e"] \arrow[d, "f"'] & & & & CS_n \arrow[d, "f"] \\
A_1 \arrow[rrrr, "a"] \arrow[d, "id"'] & & & & A_n \\
A_1 \arrow[r, "b"] & A_2 \arrow[r, "b'"] & A_3 & \cdots & A_n \arrow[u, "id"']
\end{tikzcd}$$

Figure 3.14: ClassSheet atomic operations as a sequence of basic operations.

Atomic operations over ClassSheet automata share a same basic strategy, each transition is traversed, while at the same time a coordinate count is kept. Each automaton starts at a given initial position. Every time a horizontal composition transition (|) is reached the x coordinate is incremented, likewise the y coordinate is incremented when a vertical composition transition (ˆ) occurs.

In Figure 3.16 the automaton that represents the **Item** ClassSheet model in Figure 3.15 is displayed, and above each state a coordinate count is shown. Starting in position (1,1) the ClassSheet automaton has a transition identifying an **Item** class followed by a cell with value Item located at the same coordinates. A transition indicating a vertical composition

(ˆ) is next, so the y coordinate has to be incremented, resulting in coordinates (1,2). On the subsequent rows the same process is repeated, the coordinates are unchanged until the vertical composition transition is reached, and at that point the y coordinate is incremented again. This simple example has only one column, but in case it had more than one, each time a horizontal transition was reached (|), the x coordinate had to be incremented instead.



	A
1	Item
2	value=0
3	⋮
4	

Figure 3.15: Embedded Item ClassSheet.

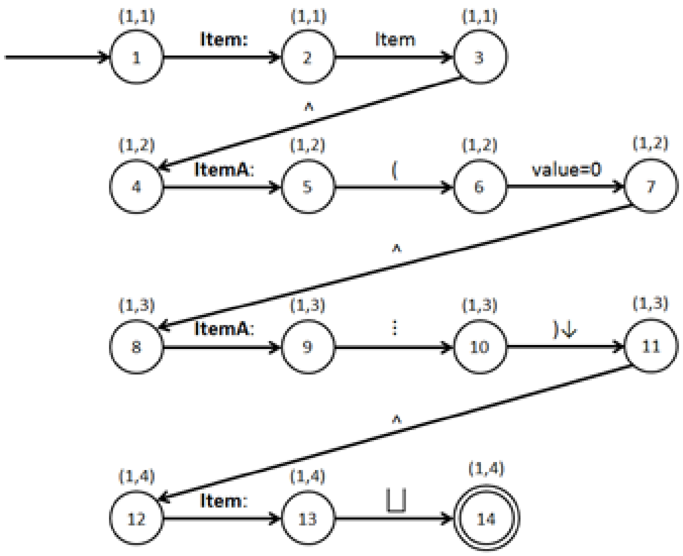


Figure 3.16: Embedded Item ClassSheet automaton.

An atomic operation is accomplished by traversing and applying basic operations to automata, based on the transition type and coordinate count. A description of each atomic operation follows.

3.3.1 Add Column

The first operation adds to an automaton the states and transitions corresponding to adding a column in a ClassSheet model. This is the automata equivalent to the following operation from [Cunha et al., 2012a].

$$addColumn_M :: Where \to Index \to Op_M$$

This operation takes as parameters the position where to insert the new column, if it should be inserted before or after that column. When defining this operation over automata, depending on the position where the new column is to be added, two different situations can occur.

The first case is the case where the new column is to be inserted before the index column (c). To do this the automaton has to be traversed, and in each occurrence of a transition of type cell that has coordinate $x = c$, the following sequence of basic operations has to be applied:

1. Add a new state ($st1$).
2. Add a new state ($st2$).
3. Edit the transition that precedes the cell transition in column (c) so that it transitions to $st1$.
4. Add a transition representing an empty cell connecting $st1$ to $st2$.
5. Add a transition connecting $st2$ to the cell transition in column (c), denoting a horizontal composition ($|$).

An example ensues to better understand how this operation is completed.

Suppose one wants to add a column before column 1 to the automaton in Figure 3.16. To do this, all the steps previously mentioned have to be taken each time a transition of type cell is located at coordinates with $x = 1$. The first transition that meets this criterion is the one that transitions from state 2 to state 3 by the symbol Item. Figure 3.17 illustrates the result of applying the basic operations to the automaton. First of all, states 15 and 16 are added, then the transition that connected states 1 and 2 by the symbol Item:, is edited so that it connects states 1 and 15, afterwards, a transition denoting an empty cell is added between states 15 and 16, and finally a transition designating a horizontal composition is added between states 16 and 2. To help visualize he result, the edited transition is shown in blue color, while states and transitions that were added appear colored in red[1].

[1]We assume colors are visible through the digital version of this document.

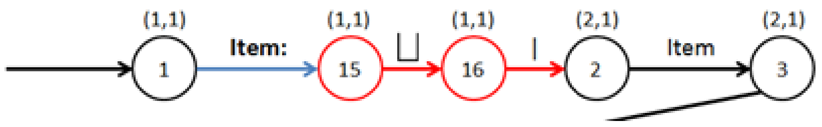


Figure 3.17: Add individual cell before column x.

Since the automaton represents a single-column ClassSheet these transformations have to be applied to every cell transition in the automaton. The final result can be seen in Figure 3.18. One important matter to mention is that cells in row 3 have a special meaning and are used to mark the vertical expandability of the class, so instead of adding an empty cell, a cell of the same type must be added.

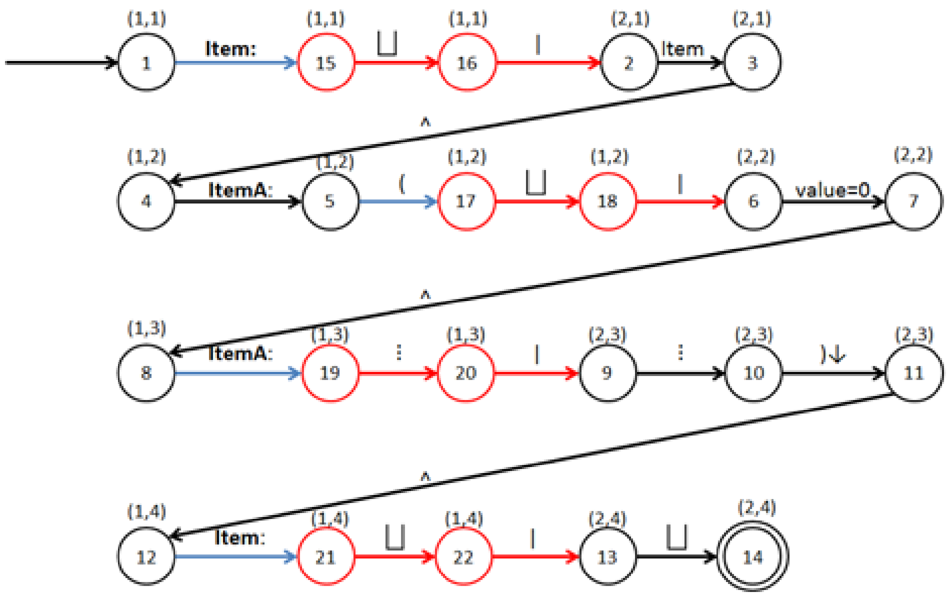


Figure 3.18: Add column before 1, operation.

A different situation occurs when a column is added after the reference column (c). In this case, each time a cell transition has coordinate $x = c$, the following sequence of basic operations is applied to the automaton:

1. Add a new state ($st1$).

2. Add a new state ($st2$).

3. Edit the transition that succeeds the cell transition at column c so that it succeeds $st2$ instead.

4. Add a new transition denoting a horizontal composition after the cell transition at column c and connect it to $st1$.
5. Add a new transition, representing an empty cell, connecting $st1$ to $st2$.

Adding a column after column 1 to the automata in Figure 3.16 shares the same principle as the previous case. On all transitions that represent cells and are located at coordinates with an $x = 1$ the sequence of basic operations is applied to the automaton. Once again, the first transition to meet this criterion is the transition from state 2 to 3, and at this stage states 15 and 16 are added. The transition that originates in state 3 is edited so it originates in state 16 instead, following, a transition denoting a horizontal composition is added between states 3 and 15, and finally a transition between states 15 and 16 is added representing an empty cell. The result can be seen in Figure 3.19, where the edited transition is blue colored and added states and transitions are red colored.

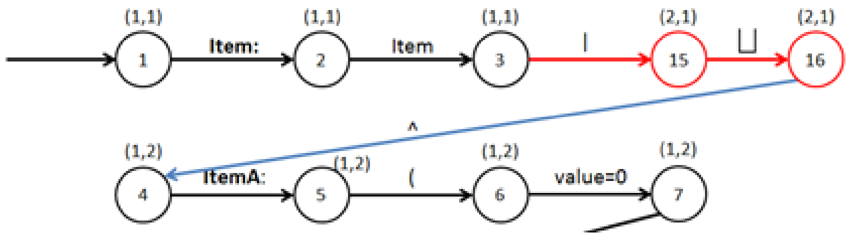


Figure 3.19: Add individual cell after x, operation.

This sequence of operations is applied on every occurrence of a cell transition resulting in the automaton displayed in Figure 3.20.

3.3.2 Delete Column

The next operation deletes the states and transitions corresponding to deleting a column from a ClassSheet model, and is the automata equivalent to the following operation from [Cunha et al., 2012a].

$$delColumn_M :: Index \rightarrow Op_M$$

This operation deletes the column located at the given index position (c). To do this, the automaton is traversed and at each occurrence of a cell transition located at coordinates with $x = c$, the following basic operations are applied.

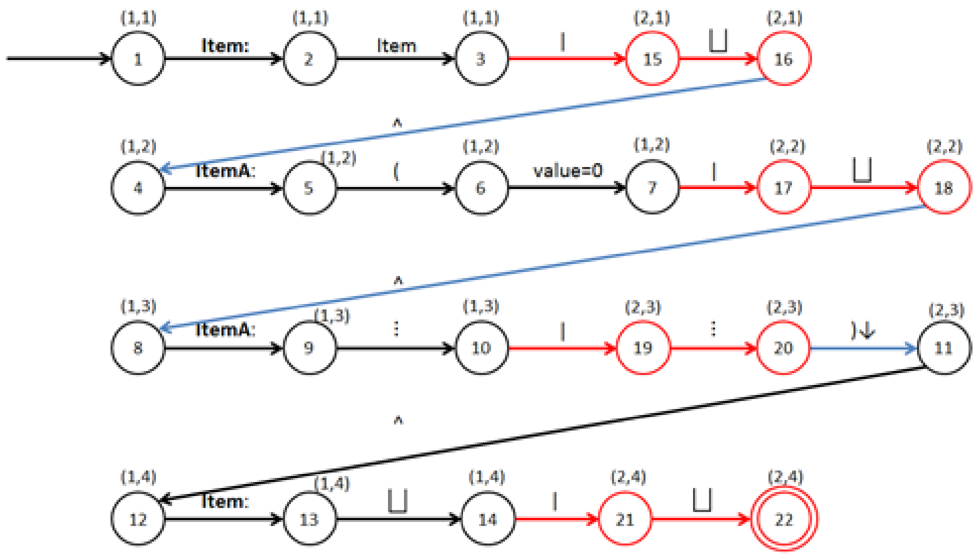


Figure 3.20: Add column after 1, operation.

1. The transition that succeeds the cell transition and respective horizontal composition transition is edited so that it originates on the transition that precedes both of them.

2. The state to where the automaton transitions by the cell symbol is deleted.

3. The state to where the automaton transitions by the horizontal composition symbol is deleted.

In the next example the operation to delete column 2 is applied to the automaton in Figure 3.20. This atomic operation consists in traversing the automata and applying the sequence of basic operations previously mentioned each time a cell transition is located at coordinates with $x = 2$. The first occurrence of such a case is at the transition that originates from state 15, and the result of applying the sequence of basic operations can be visualized in Figure 3.21. The first basic operation to apply is editing the transition that originates in state 16 so that it originates from state 3, followed by the deleting of the states 16 and 15, which are the states to where the automaton transitions by the cell symbol, and the horizontal composition symbol, respectively. The edited transition is colored in blue and the removed states and transitions are marked in gray.

The result of employing this process in all subsequent cell transitions located at column 2 is the transformation of the automaton in Figure 3.20 into the automaton in Figure 3.16.

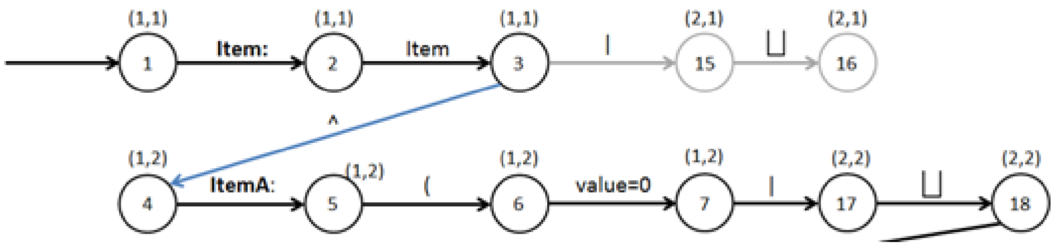


Figure 3.21: Cell removed from Item ClassSheet.

3.3.3 Add Row

The next operation adds the states and transitions corresponding to adding a row in a ClassSheet model. This is the automata equivalent to the following operation from [Cunha et al., 2012a].

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

This operation takes as input the position, before or after, and an index where the new row is to be added. The row located at the given index is used to ascertain the structure of the new row, that is, the sequence of classes and columns it possesses. As with the add column operation, when defining this operation over automata, depending on the position where the new row is to be added, two different cases can occur.

The first case arises when the new row is to be added before the index row (r). Generically, this transformation is carried out in the following steps.

1. The automaton is traversed until the first transition (t) with $y = r$.
2. The structure from the reference row is determined, and added to the automaton.
3. A vertical composition transition is added between the end of the row structure added in step 2 and the transition t.
4. The transition that precedes t is edited so it transitions to the beginning of the row structure added in step2.

As an example, suppose one wants to add a row before row 3 in the automaton in Figure 3.16. The first step is to identify the first transition with coordinate $y = 3$, which is the transition that originates in state 8 and transitions to state 9 by the symbol **ItemA:**.

The second step is to determine the reference row structure, in this case row 3 has a class **ItemA:** which possesses one column (Figure 3.22), meaning that the new row has the same structure.

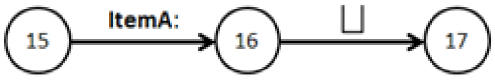


Figure 3.22: Item ClassSheet row 3 structure.

The third step is to add a transition, with a vertical composition(ˆ), connecting state 17, which is the last state in the row structure added in the previous step, to state 8, the initial state in the transition identified in the first step. Finally, the transition that originates in state 7 is edited so is transitions to state 15, the beginning of the new row. The result can be visualized in Figure 3.23.

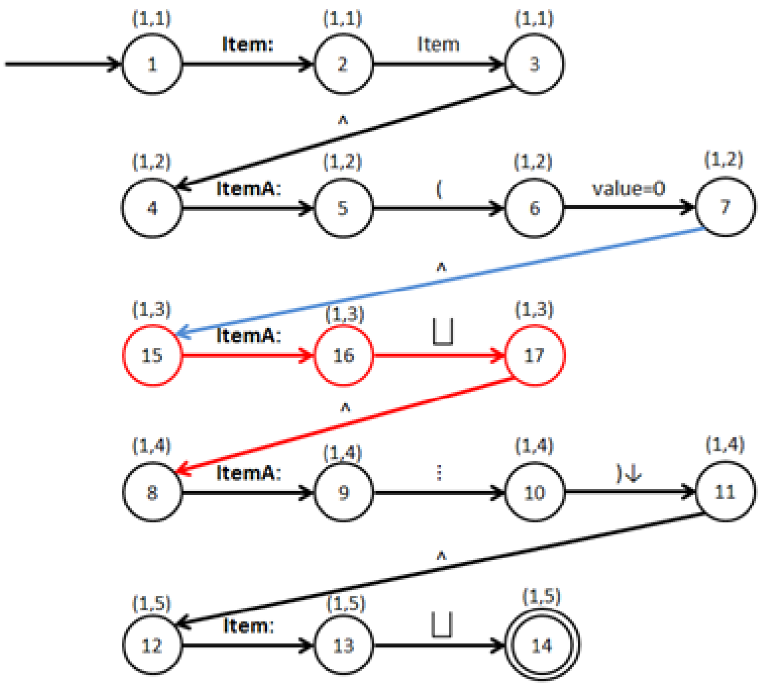


Figure 3.23: Item ClassSheet after Add Row Before 3 operation.

The second situation happens when a row is added after the index row (r). In this case the following sequence of operations is applied to the automaton:

1. The automaton is traversed until the last transition (t) with coordinate $y = r$ is reached.

2. The structure from the reference row is determined, and added to the automaton.
3. A vertical composition transition is added between the last state in transition t and the first state in the row structure added in step 2.
4. The transition that originates in the last state of transition t is edited so it originates in the last state of the row structure added in step 2.

The ensuing example should aid in the understanding of how this atomic operation is applied.

Suppose one wants to apply the operation add row after 4 to the automaton in Figure 3.16. The first step is to identify the last transition with coordinate $y = 4$, which is the transition that originates in state 13 and transitions to state 14 by the empty cell symbol. The second step is to determine the reference row structure, in this case row 4 has a class named **Item** which possesses one column (Figure 3.24), which means that the new row has the same structure.

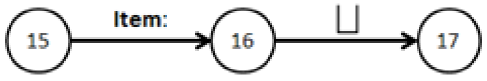


Figure 3.24: Item ClassSheet row 4 structure.

The next step is to add a vertical composition transition between state 14, which is the last state in row 4, and state 15, which is the first state in the row structure added in the previous step. Since row 14 does not have more transitions, step 4 can be ignored. The result of adding a row after row 4 to the automaton in Figure 3.16 is shown in Figure 3.25, with the added states and transitions colored in red.

3.3.4 Delete Row

The following operation deletes, from an automaton, the states and transitions corresponding to deleting a row from a ClassSheet model. It is the automata equivalent to the following operation from [Cunha et al., 2012a].

$delRow_M :: Index \to Op_M$

This operation deletes the row located at the given index position (r). In order to ensure this, the following operations are applied.

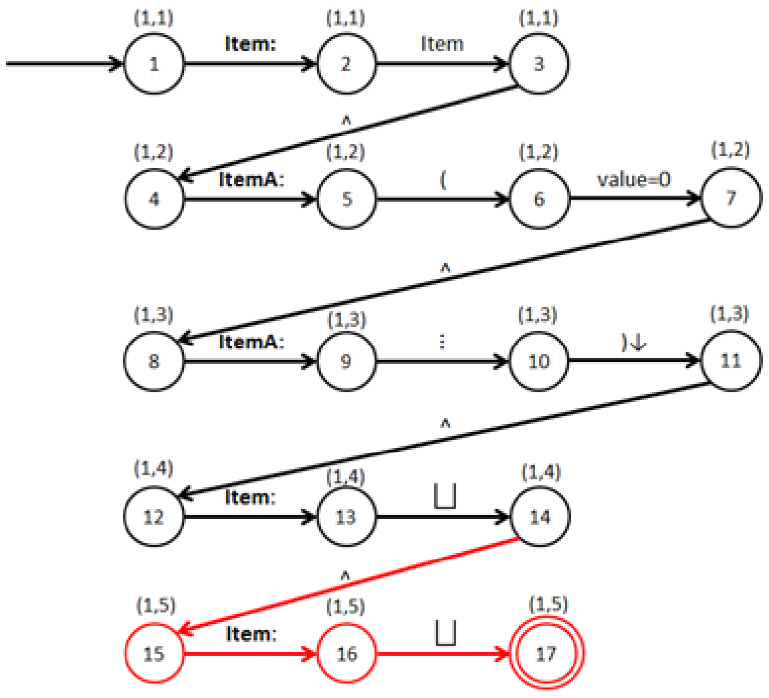


Figure 3.25: Item ClassSheet after Add Row After 4 operation.

1. The automaton is traversed until the first transition(t) with coordinate $y = r$ is reached.

2. The transition that precedes transition t, in case it exists, is edited so it transitions to the first state in row $(r + 1)$.

3. All states in row (r) are deleted.

Suppose one wants to delete row 3 in the automaton in Figure 3.23. The first step is to locate the first transition in row 3, which in this case is the one that transitions from state 15 to state 16 by the symbol **ItemA:**. Since it has a preceding transition, the one marked in blue, that transition is edited so it transitions to state 8 instead, which is the first state in row 4, thus concluding step 2. Finally all states in row 3 are removed, which comprises all the states that are colored in red. The result of removing row 3 from the automaton is pictured in Figure 3.16.

3.3.5 Set Cell

This operation changes the value of a cell type transition in an automaton, and is the automata equivalent to the following operations from [Cunha et al., 2012a].

$$setLabel_M :: (Index, Index) \rightarrow Label \rightarrow Op_M$$
$$setFormula_M :: (Index, Index) \rightarrow Formula \rightarrow Op_M$$

Which are unified in MDSheet as the following single operation.

$$setCell_M :: (Index, Index) \rightarrow String \rightarrow Op_M$$

MDSheet integration is a crucial point to take into account and as such, in this thesis, only $setCell_M$ is considered.

This operation takes as input the coordinates (c) of the cell to be modified and the new content. To apply it, the automaton has to be traversed until the transition of type cell with coordinates $(x, y) = c$ is reached. This transition is subsequently modified using an edit transition operation so it reflects the new value.

If one wants to apply the set cell (1,2) "item_value=0" operation to the automaton in Figure 3.16, the automaton has to be traversed until the transition of type cell with coordinates with value (1,2) is reached. This transition, the one that connects state 6 to state 7 by the symbol "value=0", has to be then modified to connect both states via the symbol "item_value=0". The result of using this operation is shown in Figure 3.26 with the edited transition colored in blue.

3.4 Summary

In this chapter a method for expressing ClassSheets as automata, based on their formal representation, was introduced.

A set of operations over generic automata was also defined to serve as support for higher level ClassSheet model operations over automata.

Furthermore, equivalent operations over ClassSheet models proposed in [Cunha et al., 2012a] were defined over automata. Such operations establish a base to support model evolution of ClassSheets expressed as automata, which we explore in the next chapter.

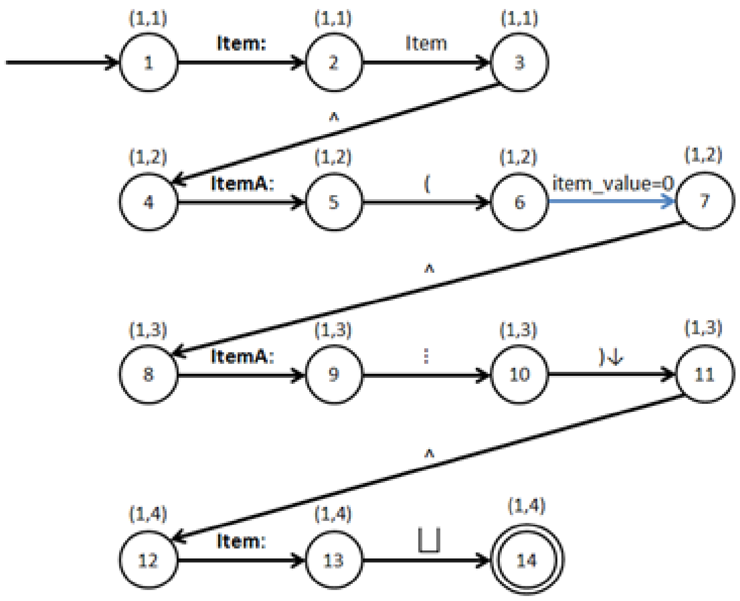


Figure 3.26: Item ClassSheet after Set Cell (1,2) item_value operation.

Chapter 4

Directed Evolution of Model-Driven Spreadsheets

Spreadsheets, like almost all software artifacts, need to be constantly updated. For numerous reasons, like reflecting changes on a business model, or in cases where spreadsheets are used to disseminate data between different systems, it might be required to change the data on a spreadsheet so that it can be compatible with the target systems. In all cases, spreadsheets have to be manually transformed by a human operator, this manual transformation has some drawbacks: as an operator modifies a spreadsheet, errors might occur, which can have significant impact on a business. On the other hand, manual transformations can also be very time consuming wich can involve substantial financial costs.

4.1 Model-Driven ClassSheet Evolution

In many real world applications where spreadsheets are used to migrate data from one system to another, both the initial and final systems' models are known in advance. This can be achieved using model-driven spreadsheets, in particular, using ClassSheets, by which both system models are specified. this can be done by inferring ClassSheets from existing data [Cunha et al., 2010] or by converting other models to ClassSheets, for instance, by applying the inverse transformation presented in [Cunha et al., 2012d]. In this chapter a technique is presented that allows to automatically evolve a given initial ClassSheet model, and co-evolve the respective instances, so that they comply with a specified final model. This technique is based on automata equivalence and transformation, using atomic operations over ClassSheet models expressed as automata defined in chapter 3, and bidirectional transformations of model-driven spreadsheets proposed in [Cunha et al., 2012a]. The method presented, and visualized in Figure 3.2, involves converting ClassSheet initial (CS_1) and final (CS_n) models to automata, identifying the sequence of atomic operations (a^x) required to be applied to the initial automaton (A_1) so that it is transformed into the final automaton (A_n). Subsequently, using the bidirectional framework defined in [Cunha et al., 2012a], the computed transformations sequence is applied to the initial model (CS_1) and propagated to the respective instances.

The evolution process proposed in this thesis implies the computation of multiple transformations sequences. In order to determine which sequence to apply we use a criterion, based on the modifications that a sequence of transformations performs on data. The sequence that requires fewer changes in cells with formulas or values is selected, this is due to the fact that this data is potentially more valuable in any decision making process. The consequence of using this criterion is that atomic operations have to be quantified so that data change cost can be calculated. This requires that for each atomic operation (a), defined on chapter 3 and illustrated in Figure 4.1, there is an operation (a') that when applied to an initial automaton (A_1) produces a pair of values, containing the same automaton (A_n) resulting from applying operation (a) to (A_1) and the respective data change cost (c).

$$\begin{tikzcd}
A_1 \arrow[r, "a"] & A_n \\
A_1 \arrow[u, "id"] \arrow[r, "a'"] & (A_n,c) \arrow[u, "\pi_1"']
\end{tikzcd}$$

Figure 4.1: Quantified atomic operations.

Operation	Cost
Insert cell transition	1
Formula change	10
Value change	10
Label change	2

Table 4.1: Costs per operation.

The values present in table 4.1 are used to calculate the cost of an operation and are explained next.

- In atomic operations where transitions representing cells are inserted, like add column and add row, each inserted cell has a cost of 1, since data on the spreadsheet is not modified.
- In operations where data might change, like delete column, delete row or set cell, the cost is determined by the type of data present in each cell. If a cell containing a label is being altered or deleted, that cost has a value of 2.
- If the data present on a cell is a formula or a value, that cost has a value of 10.

4.1.1 Computing the Sequences of Transformations

The process of ascertaining the sequence of transformations to be applied consists in traversing initial and target automata simultaneously, and consecutively comparing pairs of transitions, one from each automaton, while keeping a coordinate count for the target automaton position. When two transitions are different, a decision is taken, based on both transitions, to determine which atomic operation is applied next. The source automaton is sequentially transformed while, at the same time, a sequence of the applied transformations is kept. The automaton is transformed up until its equivalence to the target automaton is established, culminating in a valid transformation sequence.

To determine which operation has to be applied is not always possible,since in particular circumstances it is necessary to attempt different alternatives, which originates in multiple valid transformations sequences. Afterwards it is necessary to decide which sequence to apply. Such decision is based on the total cost of data change, as explained in section 4.1.

Systematically evolving a source automaton to a target automaton, in a directed way until equivalence is met, is based on a fundamental principle, that is, a transformations sequence is only pursued if at each operation applied the resulting automaton is closer to the final one. If the resulting automaton diverges, then the transformations sequence resulting from applying that operation can be discarded. Determining if the result of an operation approximates or diverges from the solution is based on the number of equal sequential transitions both automata possess, until the first difference occurs. If after an operation is applied the number of common equal sequential transitions decreases, then the result is diverging from the target automaton, otherwise it is converging. Algorithm 2 is a *naïve* approach to solve this problem. For clarity reasons the transition count is omitted, but it should be kept in mind that a transformation operation is only applied if the current transition count is equal or greater than the previous count.

Algorithm 2 Transformations Sequences Algorithm

```
 1: function COMPUTETSEQUENCES($sa$,$ta$,$ic$)
 2:     return WALK(getInitialTransition($sa$),getInitialTransition($ta$),$sa$,$ic$,$\{\}$,0)
 3: end function
 1: function WALK($st$,$tt$,$sa$,$(x, y)$,$p$,$pc$)
 2:     if hasNoMoreTransitions($st$,$tt$) then
 3:         return $(p, pc)$
 4:     end if
 5:     if $st = tt$ then
 6:         $nc \leftarrow$ updateCoordinates($st$,$(x, y)$)
 7:         return WALK(nextTransition($st$),nextTransition($tt$),$sa$,$nc$,$p$,$pc$)
 8:     end if
 9:     if targetAutomatonHasExtraColumn($st$,$tt$) then
10:         if not inLastRow($st$,$sa$) then
11:             return WALK($st$,nextTransition($tt$),$sa$,$(x, y)$,$p$,$cs$)
12:         else
13:             $(na, oc) \leftarrow$ addColumn(After $x$, $sa$)
14:             enqueue(AddColumn After $x$,$p$)
15:             return WALK(getInitialTransition($na$),getInitialTransition($ta$),$na$,$p$,$pc + oc$)
16:         end if
17:     end if
18:     if sourceAutomatonHasExtraColumn($st$,$tt$) then
19:         if not inLastRow($st$,$sa$) then
20:             return WALK(nextTransition($st$),$tt$,$sa$,$(x, y)$,$p$,$pc$)
21:         else
22:             $(na, oc) \leftarrow$ delColumn($x + 1$, $sa$)
23:             enqueue(delColumn $x + 1$,$p$)
24:             return WALK(getInitialTransition($na$),getInitialTransition($ta$),$na$,$p$,$pc + oc$)
25:         end if
26:     end if
27:     if targetAutomatonHasExtraRow($st$,$tt$) then
28:         $(na, oc) \leftarrow$ addRow(After $y$, $sa$)
29:         enqueue(addRow After $y$,$p$)
30:         return WALK(getNextTransition($st$),getNextTransition($tt$),$na$,$p$,$pc + oc$)
31:     end if
32:     if sourceAutomatonHasExtraRow($st$,$tt$) then
33:         $(na, oc) \leftarrow$ delRow($y + 1$, $sa$)
34:         enqueue(delRow $y + 1$,$p$)
```

Algorithm 2 Transformations Sequences Algorithm (continued)

```
35:         return WALK(getNextTransition($st$),getNextTransition($tt$),$na$,$p$,$pc + oc$)
36:     end if
37:     if rowsStartWithDifferentClasses($st$,$tt$) AND classIsNext($tt$,$st$,$ta$) then
38:         $(na, oc) \leftarrow$ addRow(After $y - 1$, $sa$)
39:         enqueue(addRow After $y - 1$,$p$)
40:         return WALK(getInitialTransition($na$),getInitialTransition($ta$),$na$,$p$,$pc + oc$)
41:     end if
42:     if rowsStartWithDifferentClasses($st$,$tt$) AND classIsNext($st$,$tt$,$sa$) then
43:         $(na, oc) \leftarrow$ delRow($y$, $sa$)
44:         enqueue(delRow $y$,$p$)
45:         return WALK(getNextTransition($st$),$tt$,$na$,$p$,$pc + oc$)
46:     end if
47:     if cellsHaveDifferentContent($st$,$tt$) then
48:         tryAllOperations()
49:         $all \leftarrow$ concatenateComputedTransformations()
50:         return all
51:     end if
52:     return {}
53: end function
```

The computeTSequences function takes as input a source automaton (sa), a target automaton (ta) and the source automaton initial coordinates (ic). This function invokes the walk function, which receives as input a transition from the source automaton (st), a transition from the target automaton (tt), the source automaton to be evolved (sa) the current coordinates (x, y), the transformations sequence (p) and the current transformations sequence cost (pc).

To better understand the algorithm, the cases that can occur when comparing ClassSheet models, using a row-by-row strategy, are presented. For simplicity, models are pictured in the ClassSheet visual language and the transitions being evaluated are represented as red colored cells.

The first case, displayed in Figure 4.2, arises when the transitions in the source automaton and final automaton are equal. This corresponds to the if clause in line 5 of algorithm 2.

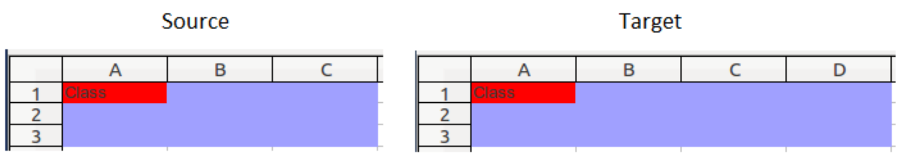


Figure 4.2: Source and target models, with equal transitions before evaluation.

When this happens the following actions have to take place.

- Update the coordinate count.
- Advance both transitions to the next ones.

The result of advancing both transitions is pictured in Figure 4.3.

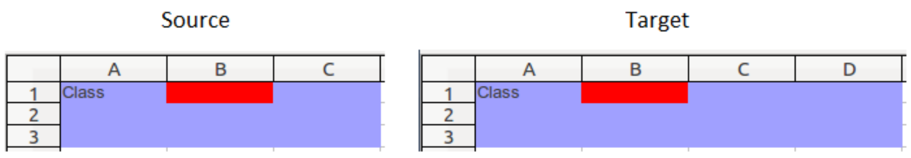


Figure 4.3: Source and target models, with equal transitions, after evaluation.

The next case, displayed in Figure 4.4, occurs when the transition in the source automaton is the last transition in a block of cells and it is not located in the final row of the model, and the target automaton has, at least, another column (if clause in line 10 of algorithm 2).

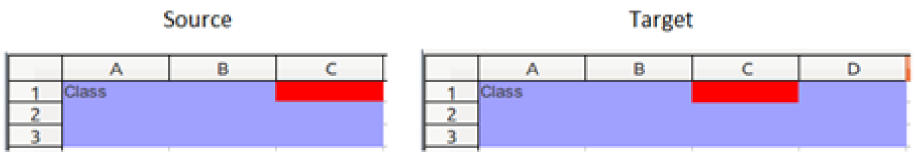


Figure 4.4: Target automaton has one more column than source automaton, before evaluation.

When this occurs the target automaton transition has to be advanced to the next one, until both automata synchronize. The result is shown in Figure 4.5.

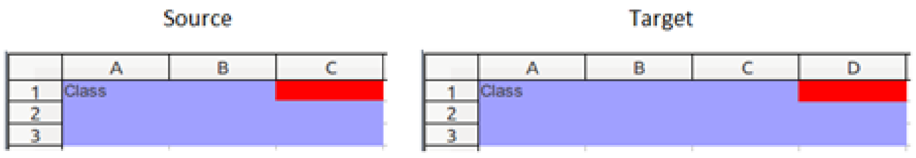


Figure 4.5: Target automaton with one more column than source automaton, before evaluation.

The following situation (Figure 4.6) arises when the source automaton transition is located at the end of a block of cells, in the last row of the automaton, and the target automaton has, at least, one more column (else clause in line 12 of algorithm 2).

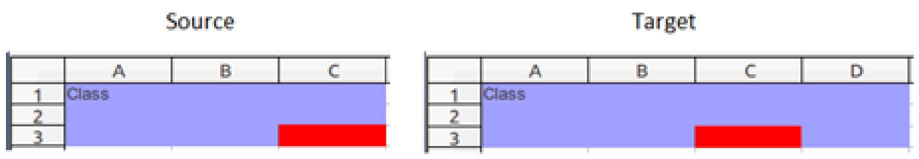


Figure 4.6: Source automaton in last row and target automata has, at least,one more column, after evaluation.

When this case occurs the following sequence of actions takes place.

- Add a column after column x to the source automaton.
- Add the transformation just applied to the sequence of transformations.
- Update the transformations sequence cost.
- Compare both automata from the beginning.

The result of applying the previous operations is shown in Figure 4.7.

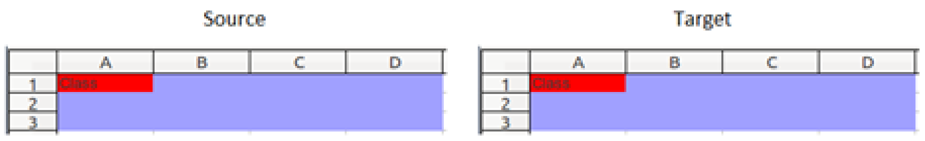


Figure 4.7: Source automaton with new added column.

The case pictured in Figure 4.8 occurs when the target automaton transition is at the end of a block of cells not located in the final row, and the source automaton has, at least, one more column (if clause in line 19 of algorithm 2).

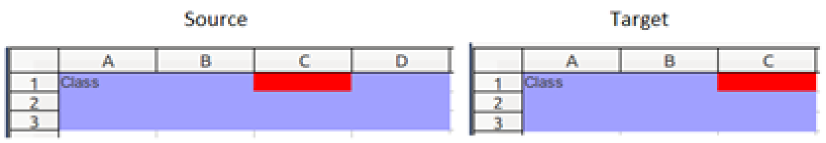


Figure 4.8: Source automaton has, at least, one more column than the target automaton.

In this case the source automaton transition is skipped to the next one. The result is displayed in Figure 4.9.

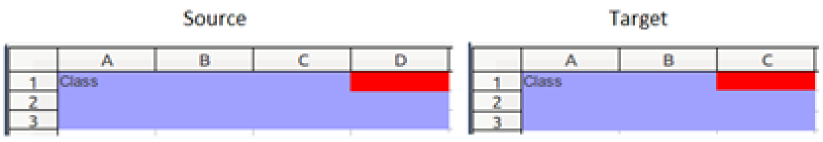


Figure 4.9: Source automaton synchronizes to target model.

The ensuing case, shown in Figure 4.10, occurs when the target automaton transition is located at the end of a block of cells and the source automaton has, at least, one more column, and the transition is located in the last row of the automaton (else clause in line 21 of algorithm 2).

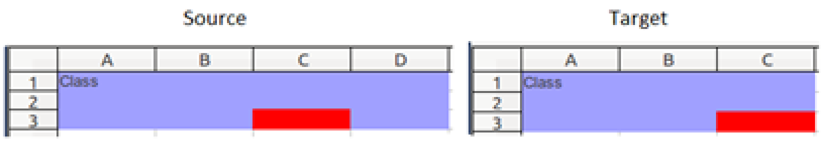


Figure 4.10: Source automaton reached the last row and has, at least, one more column.

In this situation the following steps are taken.

- Column x+1 is deleted from the initial automaton.
- The transformation applied is added to the current transformations sequence.
- The transformations sequence cost is updated.
- Both automata are compared again from the beginning.

The result is presented in Figure 4.11.

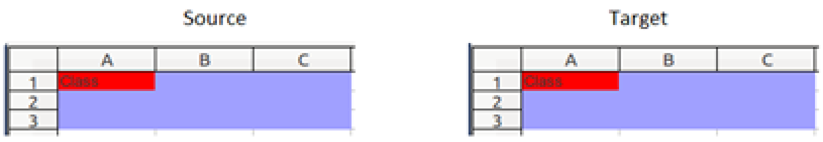


Figure 4.11: Source automaton with last column deleted.

The case depicted in Figure 4.12 occurs when the source automaton reaches its end but the target automaton still has, at least, one more row (if clause in line 27 of algorithm 2).

In this situation the following steps are taken.

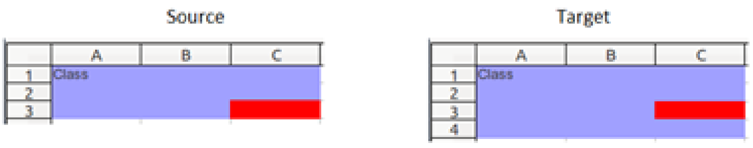


Figure 4.12: Target automaton has, at least, one more row than the source automaton.

- Add a row after y to the initial automaton.
- The transformation applied is added to the current transformations sequence.
- The transformations sequence cost is updated.
- The automata continue to be compared.

The result is shown in Figure 4.13.

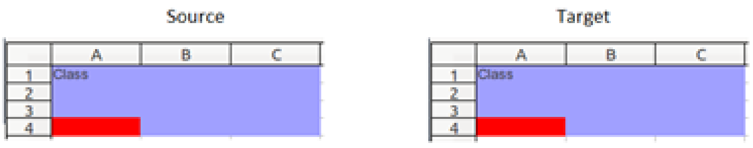


Figure 4.13: Source automaton has one more row.

The next scenario (Figure 4.14) occurs when the target automaton reaches the end the block of cells, but the source automaton has, at least, one more row (if clause in line 32 of algorithm 2).

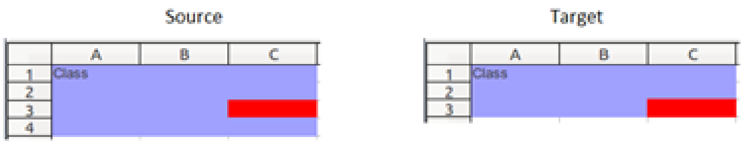


Figure 4.14: Source automaton has one more row, before evaluation.

When this happens the following steps are taken.

- Row y+1 is deleted from the source automaton.
- The transformation applied is added to the current transformations sequence.

- The transformations sequence cost is updated.
- The automata continue to be compared.

In this case, since the target automaton does not posses any more rows, equivalence is established and the algorithm computes a valid transformations sequence. If the source automaton would have any more rows, the algorithm would proceed to find any more differences. The result is shown in Figure 4.15.

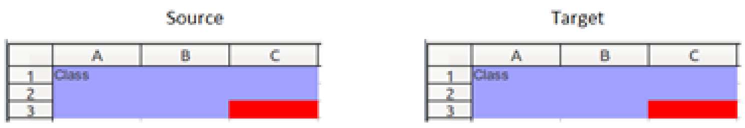


Figure 4.15: Source automaton with deleted row, after evaluation.

The next case (Figure 4.16) occurs at the beginning of both rows being processed, when each row belongs to a different class, and the current class in the source model only starts at the beginning of a subsequent row in the target model (if clause in line 37 of algorithm 2). In this example, **Class B** in the source model starts at coordinates A2, but in the target model it only starts at coordinates A3. In such situations the next steps have to e applied.

- Add a row after y-1 to the source automaton.
- The transformation applied is added to the current transformations sequence.
- The transformations sequence cost is updated.
- The automata are compared from the beginning.

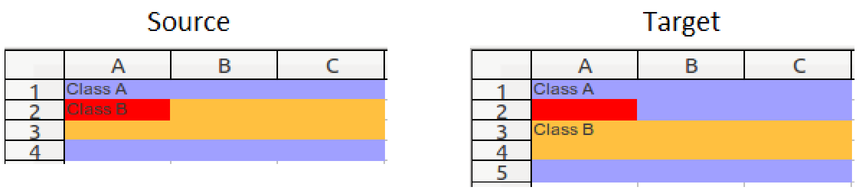


Figure 4.16: Final model has a new row preceding a class.

The result of adding the new row is shown in Figure 4.17.

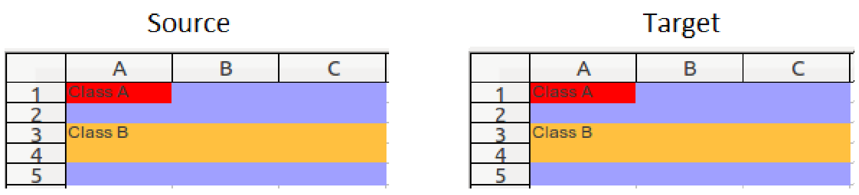


Figure 4.17: Target model after row preceding class is inserted.

The following case (Figure 4.18) arises when the two rows being processed belong to different classes, and the class in the target model starts only in a subsequent row in the source model (if clause in line 42 of algorithm 2). In this case the following steps have to be taken.

- Row y is deleted from the source automaton.
- The transformation applied is added to the current transformations sequence.
- The transformations sequence cost is updated.
- Both automata continue to be compared.

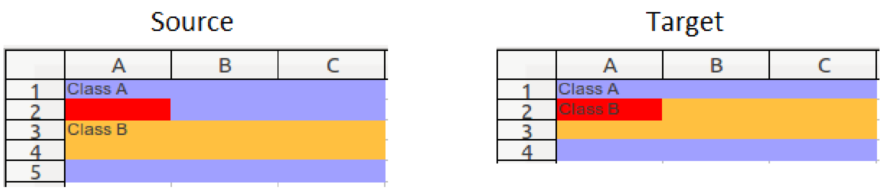


Figure 4.18: Source model has, at least, one more row preceding a class than the target model.

Figure 4.19 shows the result of removing the row from the source model.

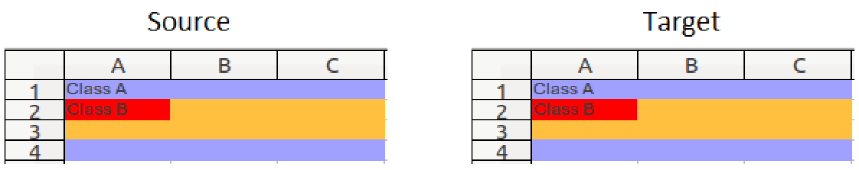


Figure 4.19: Source model after deleted row preceding class.

The subsequent case to be addressed (Figure 4.20), is the case where two cells have different content (if clause in line 47 of algorithm 2). Due to the fact that to determine the exact operation to be applied is not always a trivial task to do, multiple operations are applied to the source automaton, which potentially results in numerous transformations sequences.

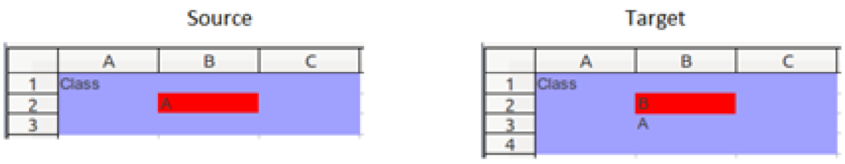


Figure 4.20: Cells holding different content.

In this particular case, the following transformation steps take place.

- Apply add column before x to the initial automaton and update the transformations sequence, and cost.
- Recursively compute the rest of the transformations sequence and respective cost.
- Repeat the previous two steps for operations add column after x-1, del column x, add row before y, add row after y-1, del row y and set cell.
- Concatenate all transformations sequences, and respective costs, resulting from the recursive calls and return the result.

The next case to address is when equivalence between the two automata is achieved. In this case the current transformations sequence and respective cost is returned.

The final case is when the number of equivalent sequential transitions in both automata decreases, after an operation is applied to the initial automaton. This means that the source automaton is diverging from the target automaton and, as such, the search for a solution with the current transformations sequence can be abandoned.

In the end the best transformations sequence, based on data cost, can be attained by simply choose the sequence with lower cost. This sequence can then be used to evolve the initial ClassSheet model, and automatically co-evolve the respective instances using the bidirectional framework defined in [Cunha et al., 2012a]. In chapter 5 some examples are presented to demonstrate how this process can be used to evolve model-driven spreadsheets.

4.2 Summary

In this chapter a method for automatic evolving of ClassSheet models, and co-evolving instances, is introduced. This method is based in automata equivalence and transformation, using atomic operations defined on Chapter 2, and bidirectional transformations.

The evolution process implies the computation of multiple transformations sequences, and, in the end, choosing the appropriate one.

Finally an algorithm is introduced, to evolve a source ClassSheet automaton to a target automaton, in a directed form, while keeping track of the atomic operations applied during that process. These atomic operations are then used to evolve ClassSheet models and instances until both comply with the target automaton.

Chapter 5

Directed Evolution of Model-Driven Spreadsheets in Practice

The validation of results of the work carried out in this thesis is done in the form of a software prototype that implements all the techniques presented previously. In this chapter several examples of evolution using this software artifact are demonstrated, where the evolution of a given source model, and the co-evolution of the respective data, is performed until both are in compliance with the new model.

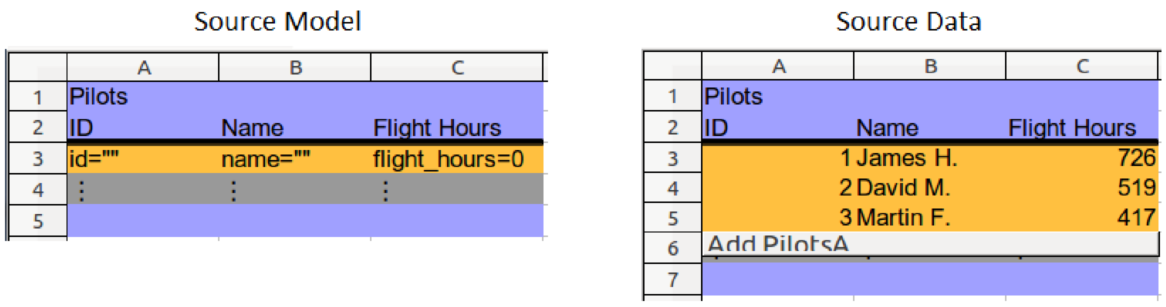
5.1 Integration with MDSheet

To illustrate the outcome of the evolution process, the software prototype is integrated into MDSheet. MDSheet is an extension for OpenOffice/LibreOffice Calc that supports the specification and manipulation of ClassSheet models and instances, using a single spreadsheet environment. To achieve this, it supports the sets of operations over ClassSheet models and instances, and the bidirectional transformations defined in [Cunha et al., 2012a], which means that whenever a model is modified, the corresponding instance is updated to bring it into conformity with the new model, and vice-versa. The developed prototype allows adding automatic evolution functionality to the range of functionalities of MDSheet.

5.2 Evolution Scenarios

This section has two purposes: i) demonstrate how automatic evolution is executed in practice, and ii) to determine how it performs. Three basic examples are presented, one for a vertically expansible class, one for a horizontally expansible class, and one for a two-dimensional expansible class. The hardware used to run the tests is a computer with an Intel Core Duo T2400 CPU, running at 1.83 GHz, and 4GB of RAM.

In order to use the automatic evolution functionality, first a model must be created in a spreadsheet, using model editing operations already supported by MDSheet. When the model is created an instance is simultaneously generated in a second worksheet. The second step is to modify that instance, using operations over instances, and fill it with data. In Figure 5.1 a Pilots ClassSheet model and respective instance, created with MDSheet can be visualized.


Source Model

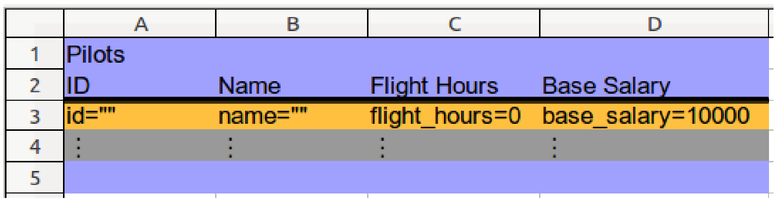
	A	B	C
1	Pilots		
2	ID	Name	Flight Hours
3	id=""	name=""	flight_hours=0
4	⋮	⋮	⋮
5			

Source Data

	A	B	C
1	Pilots		
2	ID	Name	Flight Hours
3	1	James H.	726
4	2	David M.	519
5	3	Martin F.	417
6	Add PilotsA		
7			

Figure 5.1: Pilots ClassSheet model and instance before evolution.

To evolve the model and instance previously defined, one must create the target model (Figure 5.2) in a separated sheet, select the source model and press the "Evolve To" control (Figure 5.3). In this case the target model possesses a new column, after column C, with a new attribute, *base_salary* with default value 10000.



	A	B	C	D
1	Pilots			
2	ID	Name	Flight Hours	Base Salary
3	id=""	name=""	flight_hours=0	base_salary=10000
4	⋮	⋮	⋮	⋮
5				

Figure 5.2: Pilots target model.

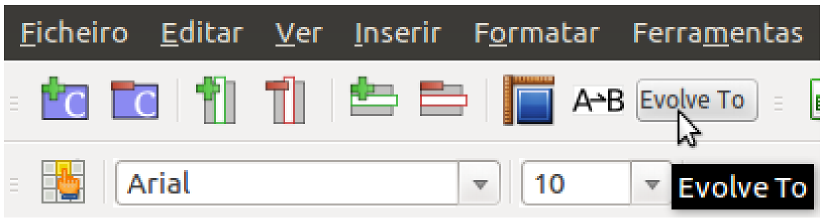
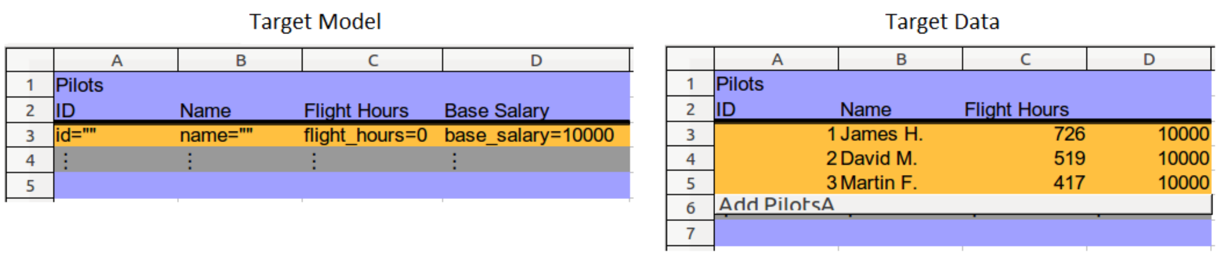


Figure 5.3: Evolve To control.

Due to the fact that multiple transformations sequences are calculated during an evolution process, the solution can take some time to be achieved. This particular example takes 6 seconds to complete, including the time it takes to compute the transformations sequence, and evolving the source model and instance until both conform to the target model. The result can be seen in Figure 5.4.


Target Model

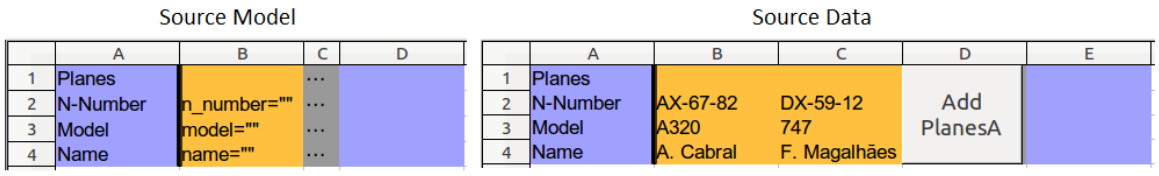
	A	B	C	D
1	Pilots			
2	ID	Name	Flight Hours	Base Salary
3	id=""	name=""	flight_hours=0	base_salary=10000
4	⋮	⋮	⋮	⋮
5				

Target Data

	A	B	C	D
1	Pilots			
2	ID	Name	Flight Hours	
3	1	James H.	726	10000
4	2	David M.	519	10000
5	3	Martin F.	417	10000
6	Add PilotsA			
7				

Figure 5.4: Pilots ClassSheet model and instance after evolution.

In the next example a Planes ClassSheet model is presented alongside with a conforming instance (see Figure 5.5).


Source Model

	A	B	C	D
1	Planes		⋯	
2	N-Number	n_number=""	⋯	
3	Model	model=""	⋯	
4	Name	name=""	⋯	

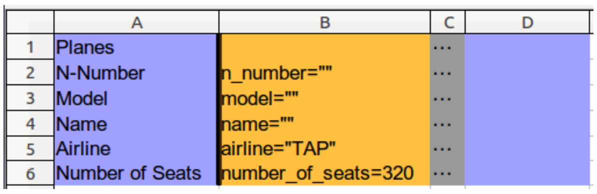
Source Data

	A	B	C	D	E
1	Planes				
2	N-Number	AX-67-82	DX-59-12	Add PlanesA	
3	Model	A320	747		
4	Name	A. Cabral	F. Magalhães		

Figure 5.5: Planes ClassSheet model and instance after evolution.

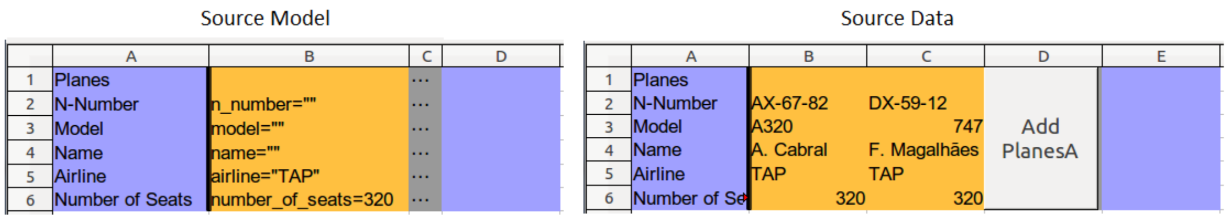
Both model and instance are to be evolved so that both comply with the model in Figure 5.6, which possesses two new rows with attributes *airline* and *number_of_seats* respectively.

The total computational time to evolve this example is 1 minute and 17 seconds. The result, with both model an instance conforming to the target model, is presented in Figure 5.7.



	A	B	C	D
1	Planes		…	
2	N-Number	n_number=""	…	
3	Model	model=""	…	
4	Name	name=""	…	
5	Airline	airline="TAP"	…	
6	Number of Seats	number_of_seats=320	…	

Figure 5.6: Planes target model.



Source Model

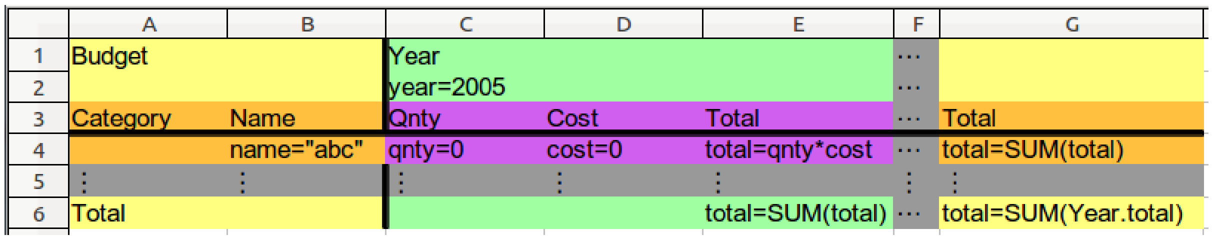
	A	B	C	D
1	Planes		…	
2	N-Number	n_number=""	…	
3	Model	model=""	…	
4	Name	name=""	…	
5	Airline	airline="TAP"	…	
6	Number of Seats	number_of_seats=320	…	

Source Data

	A	B	C	D	E
1	Planes			Add PlanesA	
2	N-Number	AX-67-82	DX-59-12		
3	Model	A320	747		
4	Name	A. Cabral	F. Magalhães		
5	Airline	TAP	TAP		
6	Number of Se	320	320		

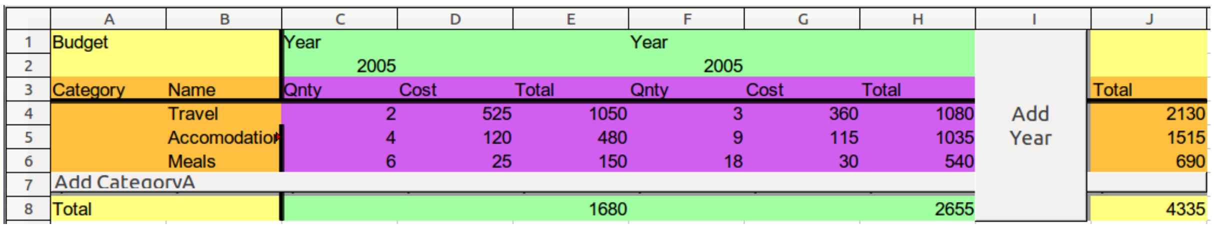
Figure 5.7: Planes ClassSheet model and instance after evolution.

The last example is a more complex, two-dimensionally expansible, Budget ClassSheet. The source model and instance are exhibited in Figure 5.8 and Figure 5.9, respectively.



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

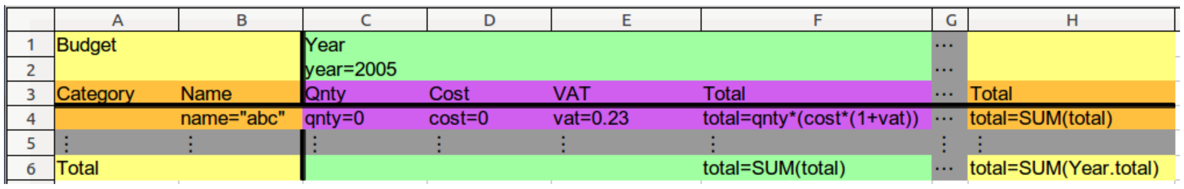
Figure 5.8: Budget source model before evolution.



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

Figure 5.9: Budget source instance before evolution.

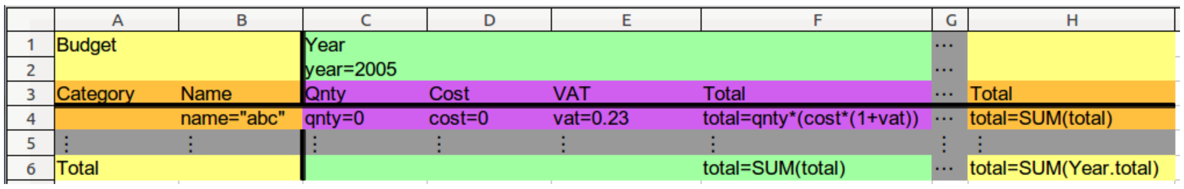
Both model and instance are to be evolved until they comply with the target model pictured in Figure 5.10. This model has an extra column between columns D and E, which possesses a new attribute named *vat* with default value 0.23. The formula in the total attribute, in the association class, is also updated to take in to account vat when calculating the total cost.



	A	B	C	D	E	F	G	H
1	Budget		Year				⋯	
2			year=2005				⋯	
3	Category	Name	Qnty	Cost	VAT	Total	⋯	Total
4		name="abc"	qnty=0	cost=0	vat=0.23	total=qnty*(cost*(1+vat))	⋯	total=SUM(total)
5	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
6	Total					total=SUM(total)	⋯	total=SUM(Year.total)

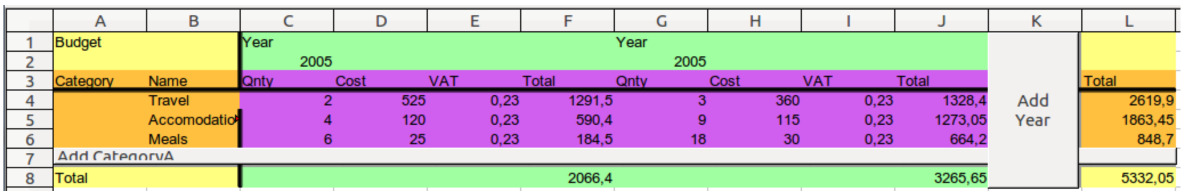
Figure 5.10: Budget target model.

This example is one of the more computational intensives, taking 7 minutes and 26 seconds to calculate the transformations sequence and evolve the model and instance. The resulting model is shown in Figure 5.11, and the resulting instance is show in Figure 5.12.



	A	B	C	D	E	F	G	H
1	Budget		Year				⋯	
2			year=2005				⋯	
3	Category	Name	Qnty	Cost	VAT	Total	⋯	Total
4		name="abc"	qnty=0	cost=0	vat=0.23	total=qnty*(cost*(1+vat))	⋯	total=SUM(total)
5	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
6	Total					total=SUM(total)	⋯	total=SUM(Year.total)

Figure 5.11: Budget source model after evolution.



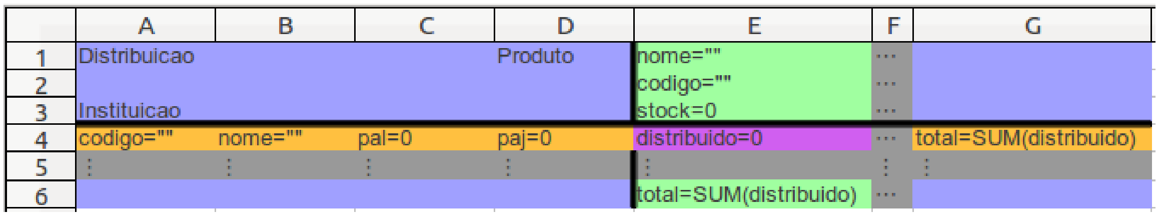
	A	B	C	D	E	F	G	H	I	J	K	L
1	Budget		Year				Year					
2				2005				2005				
3	Category	Name	Qnty	Cost	VAT	Total	Qnty	Cost	VAT	Total		Total
4		Travel	2	525	0,23	1291,5	3	360	0,23	1328,4	Add Year	2619,9
5		Accomodatio	4	120	0,23	590,4	9	115	0,23	1273,05		1863,45
6		Meals	6	25	0,23	184,5	18	30	0,23	664,2		848,7
7	Add CategoryA											
8	Total					2066,4				3265,65		5332,05

Figure 5.12: Budget source instance after evolution.

This case yields some interesting results, changing the formula of the total attribute, in the association class, has the consequence of also changing the values of the total attributes of the classes Category and Year, as can be seen in the resulting instance.

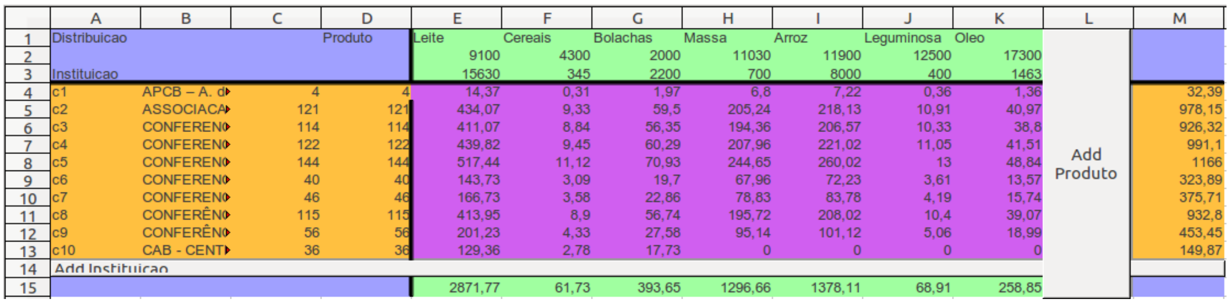
5.2.1 Food Bank: Directed Evolution In Real World

At this point we are able to model the business logic of a large spreadsheet using ClassSheets and we can automatically evolve an initial model to a target model, but evolution between ClassSheets can be inserted in a broader setting. To give some context a real world spreadsheet is utilized. This spreadsheet is taken from Banco Alimentar de Braga (BA)[1] institution, a social solidarity institution devoted to helping people in need with food products. In Figure 5.13 the spreadsheet model, defined using MDSheet, is displayed. This model specifies the structure of a spreadsheet used to store products and respective quantities to be distributed to various other social solidarity institutions. Figure 5.14 shows an instance conforming to that model as generated by MDSheet. Note that the data here used is not real data, as this would reveal private information.



	A	B	C	D	E	F	G
1	Distribuicao			Produto	nome=""	⋯	
2					codigo=""	⋯	
3	Instituicao				stock=0	⋯	
4	codigo=""	nome=""	pal=0	paj=0	distribuido=0	⋯	total=SUM(distribuido)
5	⋮	⋮	⋮	⋮	⋮	⋮	⋮
6					total=SUM(distribuido)	⋯	

Figure 5.13: Banco Alimentar de Braga ClassSheet model.



	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Distribuicao			Produto	Leite	Cereais	Bolachas	Massa	Arroz	Leguminosa	Oleo	Add Produto	
2					9100	4300	2000	11030	11900	12500	17300		
3	Instituicao				15630	345	2200	700	8000	400	1463		
4	c1	APCB – A. d	4	4	14,37	0,31	1,97	6,8	7,22	0,36	1,36		32,39
5	c2	ASSOCIACA	121	121	434,07	9,33	59,5	205,24	218,13	10,91	40,97		978,15
6	c3	CONFEREN	114	114	411,07	8,84	56,35	194,36	206,57	10,33	38,8		926,32
7	c4	CONFEREN	122	122	439,82	9,45	60,29	207,96	221,02	11,05	41,51		991,1
8	c5	CONFEREN	144	144	517,44	11,12	70,93	244,65	260,02	13	48,84		1166
9	c6	CONFEREN	40	40	143,73	3,09	19,7	67,96	72,23	3,61	13,57		323,89
10	c7	CONFEREN	46	46	166,73	3,58	22,86	78,83	83,78	4,19	15,74		375,71
11	c8	CONFERÊN	115	115	413,95	8,9	56,74	195,72	208,02	10,4	39,07		932,8
12	c9	CONFERÊN	56	56	201,23	4,33	27,58	95,14	101,12	5,06	18,99		453,45
13	c10	CAB - CENT	36	36	129,36	2,78	17,73	0	0	0	0		149,87
14	Add Instituicao												
15					2871,77	61,73	393,65	1296,66	1378,11	68,91	258,85		

Figure 5.14: Banco Alimentar de Braga ClassSheet instance.

Suppose that BA now has a database and its administrator specifies a new schema that conforms to the UML class diagram in Figure 5.15. This results in the necessity to migrate the spreadsheet instance in Figure 5.14 so that the data can be stored in the database. To do this two steps have to be taken: i) a ClassSheet model has to be inferred from the UML class diagram, using a reverse technique to the one proposed in [Cunha et al., 2012d], after infering the ClassSheet from data [Cunha et al., 2010], and, ii) the initial model (Figure 5.13) has to be evolved until equivalence to the new model is met, while at the same time co-evolving the respective instance (Figure 5.14) so it conforms to the new model. The resulting instance (Figure 5.17) can then be used to store the information in the database using techniques introduced in [Cunha et al., 2009]. The entire process can be visualized in Figure 5.18.

[1]`http://www.braga.bancoalimentar.pt/` (19-9-2013).

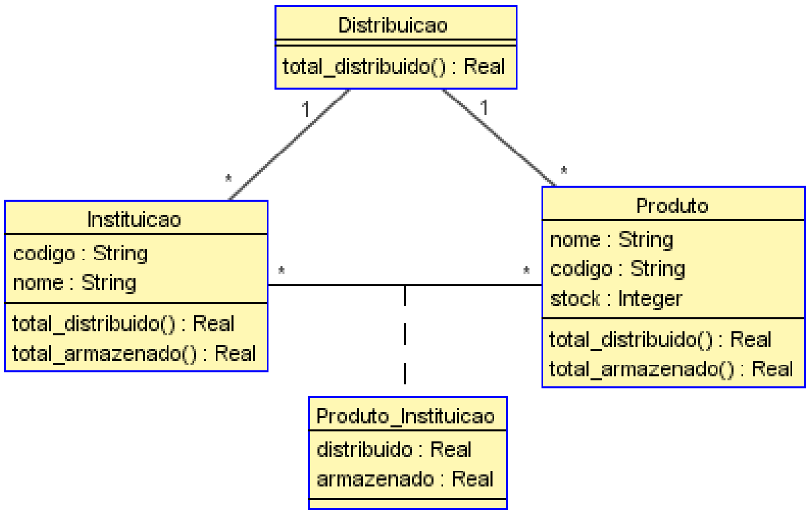
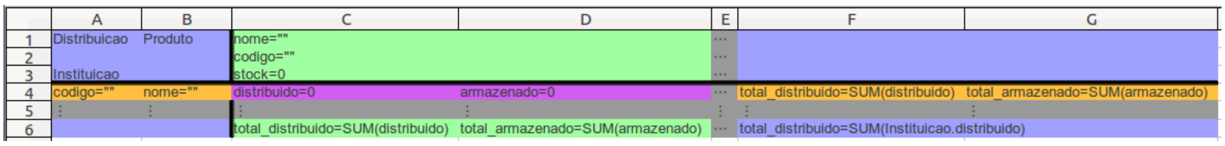
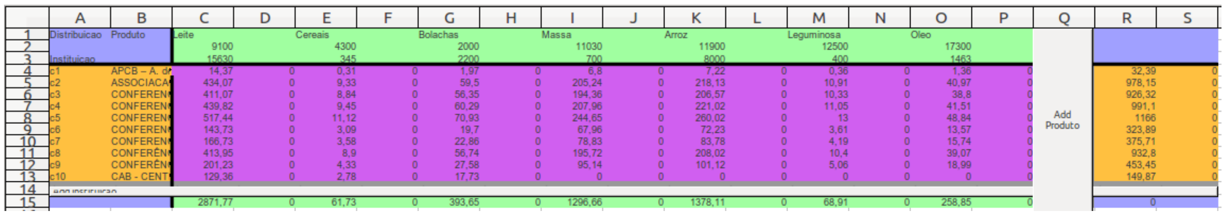


Figure 5.15: Banco Alimentar new UML class diagram.



	A	B	C	D	E	F	G
1	Distribuicao	Produto	nome=""		⋯		
2			codigo=""		⋯		
3	Instituicao		stock=0		⋯		
4	codigo=""	nome=""	distribuido=0	armazenado=0	⋯	total_distribuido=SUM(distribuido)	total_armazenado=SUM(armazenado)
5	⋮	⋮	⋮	⋮	⋮	⋮	⋮
6			total_distribuido=SUM(distribuido)	total_armazenado=SUM(armazenado)	⋯	total_distribuido=SUM(Instituicao.distribuido)	

Figure 5.16: Banco Alimentar new ClassSheet Model.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
1	Distribuicao	Produto	Leite		Cereais		Bolachas		Massa		Arroz		Leguminosa		Oleo				
2			9100		4300		2000		11030		11900		12500		17300				
3	Instituicao		15630		345		2200		700		8000		400		1463				
4	c1	APCB – A. d	14,37	0	0,31	0	1,97	0	6,8	0	7,22	0	0,36	0	1,36	0		32,39	0
5	c2	ASSOCIACA	434,07	0	9,33	0	59,5	0	205,24	0	218,13	0	10,91	0	40,97	0		978,15	0
6	c3	CONFEREN	411,07	0	8,84	0	56,35	0	194,36	0	206,57	0	10,33	0	38,8	0		926,32	0
7	c4	CONFEREN	439,82	0	9,45	0	60,29	0	207,96	0	221,02	0	11,05	0	41,51	0		991,1	0
8	c5	CONFEREN	517,44	0	11,12	0	70,93	0	244,65	0	260,02	0	13	0	48,84	0	Add Produto	1166	0
9	c6	CONFEREN	143,73	0	3,09	0	19,7	0	67,96	0	72,23	0	3,61	0	13,57	0		323,89	0
10	c7	CONFEREN	166,73	0	3,58	0	22,86	0	78,83	0	83,78	0	4,19	0	15,74	0		375,71	0
11	c8	CONFERÊN	413,95	0	8,9	0	56,74	0	195,72	0	208,02	0	10,4	0	39,07	0		932,8	0
12	c9	CONFERÊN	201,23	0	4,33	0	27,58	0	95,14	0	101,12	0	5,06	0	18,99	0		453,45	0
13	c10	CAB - CENT	129,36	0	2,78	0	17,73	0	0	0	0	0	0	0	0	0		149,87	0
14	add instituicao																		
15			2871,77	0	61,73	0	393,65	0	1296,66	0	1378,11	0	68,91	0	258,85	0		0	

Figure 5.17: Banco Alimentar new ClassSheet instance.

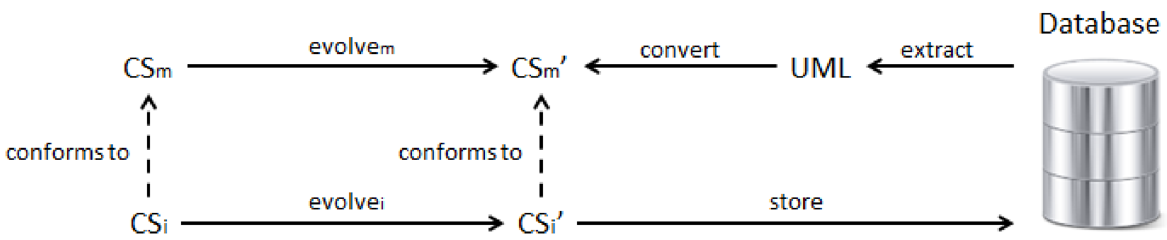


Figure 5.18: Evolution in real world.

5.3 Limitations

Although the prototype already supports a significant number of operations it still has the following limitations.

- Only supports one class or table. Our algorithm supports evolution of individual ClassSheet models, and at this stage our front end is only able of single ClassSheet model evolution. For example, the airline spreadsheet in Figure 2.9 is not yet fully supported.
- At this stage add/remove class operations are not supported.

5.4 Summary

In this chapter a prototype, with the implementation of the techniques proposed in this thesis, is presented.

A description of how the prototype is integrated with MDSheet, and can be used to evolve ClassSheet models and instances, is made.

Some tests are exhibited, showing the result of evolving a source model and instance until they conform to a target model. We also present the time it takes to compute that evolution.

Finally, some limitations of the prototype are referred.

Chapter 6

Conclusion

Besides being used by human operators, spreadsheets are also used to bond different systems. Data produced by one system is transferred, as a spreadsheet, to be consumed by a target system. What happens in some cases is that the data produced by the source system is not totally compatible with the endpoint system, and data has to be adapted so it can be processed. Generally this job is assigned to a human operator. This person modifies the source spreadsheet until the format conforms to the system that is going to process it. Due to the human factor sometimes errors are inadvertently introduced.

Much research has been done on Spreadsheets in recent years, with the aim to reduce errors typically present in them. Tools were introduced by spreadsheet software vendors to try to decrease mistakes made by users.

Model-driven spreadsheet development was introduced to try to mitigate this problem. One of the most accepted methods by the scientific community was proposed in [Engels and Erwig, 2005]. This approach is based on model-driven engineering and allows specifying spreadsheet business logic using software engineering constructs, like classes and attributes.

The work done in this thesis provides a framework for automatic evolution of ClassSheet models and instances, based on automata equivalence and transformation. ClassSheet model operations were implemented as automata operations and used in the evolution process to determine which operations have to be applied to a source model, expressed as automaton, until equivalence to a final automaton is achieved.

The presented framework offers the possibility to solve two kinds of problems: reduce the errors introduced by users, by allowing them to specify a final model and only then migrate the data, and second, allows seamlessly bridging two systems, in a fully automatic way, i.e., removing the need of human intervention.

6.1 Answers To The Research Questions

We are now in conditions to answer the research questions asked at the end of chapter 1.

Q: Can ClassSheet models be expressed as automata?

A: ClassSheet models can be expressed as automata, using their textual representation and the transformation method proposed in section 3.1.

Q: Can model evolution operations proposed in [Cunha et al., 2012a] be defined over finite automata?

A: Equivalent operations defined over ClassSheet models in [Cunha et al., 2012a] can be defined over finite automata, as presented on chapter 3.

Q: Can an initial ClassSheet model, expressed as automaton, be consecutively transformed, in a directed way, until it is equivalent to a given final automaton?

A: An initial ClassSheet model expressed as automaton can be consecutively transformed, in a directed way, until equivalence to a given final automaton is achieved. In Chapter 4 we developed an algorithm that allows this directed evolution to take place while, at the same time, keeping track of the sequence of transformations applied to the initial automaton.

6.2 Future Work

Furthermore, we have integrated automatic evolution support into a tool, that allows manipulating ClassSheet models and instances, to validate the work done in this thesis.

At this stage the developed prototype only supports evolution of models with different cell contents or different physical dimensions, by adding or removing columns, or adding or removing rows. As future work functionalities to add and remove classes are planned. Also the possibility to use ClassSheet evolution to evolve other types of models and instances is left open. Other types of models can be converted to ClassSheets, evolved and, in the end, the evolution mechanism can be used to evolve the original models and instances, as with spreadsheets. Selecting the transformation sequence to be applied is based on data change, in the future a different criterion based on spreadsheet [Cunha et al., 2012c] or ClassSheet model quality [Cunha et al., 2013] can be used.

Also a paper is being submited to be released in the near future with the findings resulting from the work carried out in this thesis.

References

ABC (2005). Accountants make AUD$30M mistake. http://www.abc.net.au/news/newsitems/200506/s1394937.htm (last retrieved: 19-09-2013). 6

Abraham, R., Erwig, M., Kollmansberger, S., and Seifert, E. (2005). Visual Specifications of Correct Spreadsheets. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC '05, pages 189–196. IEEE Computer Society. 7, 11, 12

Beckwith, L., Cunha, J., ao Paolo Fernandes, J., and Saraiva, J. (2011a). End-users productivity in model-based spreadsheets: An empirical study. In *Proceedings of the Third International Symposium on End-User Development*, IS-EUD '11, pages 282–288. 8

Beckwith, L., Cunha, J., Fernandes, J. P., and Saraiva, J. (2011b). An empirical study on end-users productivity using model-based spreadsheets. In Thorne, S. and Croll, G., editors, *Proceedings of the European Spreadsheet Risks Interest Group*, EuSpRIG '11, pages 87–100. 8

Blade, T. (2004). University of Toledo loses $2.4M in projected revenue. http://www.toledoblade.com/Education/2004/05/01/University-of-Toledo-loses-2-4M-in-projected-revenue.html (last retrieved: 19-09-2013). 6

Bricklin, D. VisiCalc: Information from its creators, Dan Bricklin and Bob Frankston. http://www.bricklin.com/visicalc.htm (last retrieved: 19-09-2013). 4

Campbell-Kelly, M., C. M. F. R. R. E. (2007a). Introduction. In *The History of Mathematical Tables From Summer to Spreadsheets*, pages 1–15. Oxford University Press. 1

Campbell-Kelly, M. (2007b). The rise and rise of the spreadsheet. In *The History of Mathematical Tables From Summer to Spreadsheets*, pages 323–346. Oxford University Press. 2, 3, 4

Catless (1995). The Risks Digest Volume 16: Issue 72 - Computing error at Fidelity's Magellan fund. `http://catless.ncl.ac.uk/Risks/16.72.html` (last retrieved: 19-09-2013). 6

Cunha, A., Oliveira, J. N., and Visser, J. (2006). Type-Safe Two-Level Data Transformation. In Misra, J., Nipkow, T., and Sekerinski, E., editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 284–299. Springer-Verlag. 8

Cunha, A. and Visser, J. (2007). Strongly Typed Rewriting for Coupled Software Transformation. *Electronic Notes in Theoretical Computer Science*, 174(1):17–34. Elsevier Science. 8

Cunha, J. (2011). *Model-based Spreadsheet Engineering*. PhD thesis, University of Minho. 8

Cunha, J., Erwig, M., and Saraiva, J. a. (2010). Automatically inferring classsheet models from spreadsheets. In *Proceedings of the 2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, VLHCC '10, pages 93–100, Washington, DC, USA. IEEE Computer Society. 47, 66

Cunha, J., Fernandes, J. P., Mendes, J., Pacheco, H., and Saraiva, J. (2012a). Bidirectional Transformation of Model-Driven Spreadsheets. In Hu, Z. and de Lara, J., editors, *Theory and Practice of Model Transformations – ICMT 2012*, volume 7307 of *Lecture Notes in Computer Science*, pages 105–120. Springer-Verlag. 5, 9, 10, 19, 20, 23, 24, 34, 35, 38, 40, 42, 44, 48, 58, 61, 70

Cunha, J., Fernandes, J. P., Mendes, J., and Saraiva, J. (2012b). MDSheet: A Framework for Model-driven Spreadsheet Engineering. In *Proceedings of the 34rd International Conference on Software Engineering*, ICSE'12, pages 1412–1415. ACM. 8, 20

Cunha, J., Fernandes, J. P., Mendes, J., and Saraiva, J. (2013). Complexity Metrics for Spreadsheet Models. In *The 13th International Conference on Computational Science and Its Applications*, ICCSA'13. LNCS. to appear. 71

Cunha, J., Fernandes, J. P., Peixoto, C., and Saraiva, J. (2012c). A quality model for spreadsheets. In *Proceedings of the 8th International Conference on the Quality of Information and Communications Technology, Quality in ICT Evolution Track*, pages 231–236. 71

Cunha, J., Fernandes, J. P., and Saraiva, J. (2012d). From Relational ClassSheets to UML+OCL. In *Proceedings of the Software Engineering Track at the 27th Annual ACM Symposium On Applied Computing (SAC 2012)*, pages 1151–1158. ACM. 16, 17, 18, 47, 66

Cunha, J., Saraiva, J., and Visser, J. (2009). From spreadsheets to relational databases and back. In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipolation*, PEPM '09, pages 179–188, New York, NY, USA. ACM. 9, 66

Cunha, J., Visser, J., Alves, T., and Saraiva, J. (2011). Type-Safe Evolution of Spreadsheets. In Giannakopoulou, D. and Orejas, F., editors, *Fundamental Approaches to Software Engineering – FASE '11/ETAPS '11*, volume 6603 of *Lecture Notes in Computer Science*, pages 186–201. Springer-Verlag. 8

Engels, G. and Erwig, M. (2005). Classsheets: Automatic Generation of Spreadsheet Applications from Object-Oriented Specifications. In *Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering*, ASE '05, pages 124–133. ACM. 7, 8, 11, 13, 15, 25, 69

Engels, G. and Groenewegen, L. (2000). Object-Oriented Modeling: A Roadmap. In *Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering*, ICSE'00, pages 103–116. ACM. 11

Erwig, M., Abraham, R., Cooperstein, I., and Kollmansberger, S. (2005). Automatic Generation and Maintenance of Correct Spreadsheets. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 136–145. ACM. 7

EuSpRiG. European Spreadsheet Risks Interest Group. `http://www.eusprig.org/` (last retrieved: 19-09-2013). 6

Gogolla, M., Büttner, F., and Richters, M. Use: A UML-based specification environment for validating UML and OCL. 17

Hermans, F., Pinzger, M., and van Deursen, A. (2010). Automatically Extracting Class Diagrams from Spreadsheets. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, ECOOP '10, pages 52–75. Springer-Verlag. 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 '97. 7

Kleppe, A., W. J. (2003). *MDA Explained: The Model Driven Architecture Practice and Promise*. Addison-Wesley. 11

Mendes, J. (2011). Classsheet-driven Spreadsheet Environments. In *Proceedings of the 2011 IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC '11, pages 235–236. 18, 19, 20

Panko, R. R. (2008). Spreadsheet errors: What we know. what we think we can do. *CoRR*, abs/0802.3457. 6

Panko, R. R. and Ordway, N. (2008). Sarbanes-Oxley: What About all the Spreadsheets? *CoRR*, abs/0804.0797. 5, 6

Powell, S. and Baker, K. (2011). *Management Science: The Art of Modeling with Spreadsheets*. John Wiley & Sons. 6

Register, T. (2003). Excel snafu costs firm$24M. `http://www.theregister.co.uk/2003/06/19/excel_snafu_costs_firm_24m/` (last retrieved: 19-09-2013). 6

Robson, E. (2007). Tables and tabular formatting in Sumer, Babylonia, and Assyria, 2500 BCE-50 CE. In *The History of Mathematical Tables From Summer to Spreadsheets*, pages 19–48. Oxford University Press. 1, 2

Telegraph, T. (2012). London 2012 Olympics: lucky few to get 100m final tickets after synchronised swimming was overbooked by 10,000. `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` (last retrieved: 19-09-2013). 6