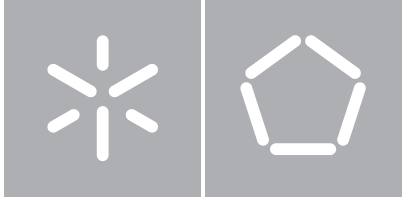
Universidade do Minho
Escola de Engenharia

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

Anexo 3

DECLARAÇÃO

Nome

André António dos Santos da Silva

Endereço electrónico: PG21001@alunos.uminho.pt Telefone: 253279037 / _____

Número do Bilhete de Identidade: 11654198

Título dissertação □/tese ☒

Directed Evolution of Model-Driven Spreadsheets

Orientador(es):

João Alexandre Baptista Vieira Saraiva

Jácome Miguel da Costa Cunha Ano de conclusão: 2013

Designação do Mestrado ou do Ramo de Conhecimento do Doutoramento:

Mestrado em Engenharia Informática

Nos exemplares das teses de doutoramento ou de mestrado ou de outros trabalhos entregues para prestação de provas públicas nas universidades ou outros estabelecimentos de ensino, e dos quais é obrigatoriamente enviado um exemplar para depósito legal na Biblioteca Nacional e, pelo menos outro para a biblioteca da universidade respectiva, deve constar uma das seguintes declarações:

1. É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA TESE/TRABALHO APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

2. É AUTORIZADA A REPRODUÇÃO PARCIAL DESTA TESE/TRABALHO (indicar, caso tal seja necessário, nº máximo de páginas, ilustrações, gráficos, etc.), APENAS PARA EFEITOS DE INVESTIGAÇÃO, , MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

3. DE ACORDO COM A LEGISLAÇÃO EM VIGOR, NÃO É PERMITIDA A REPRODUÇÃO DE QUALQUER PARTE DESTA TESE/TRABALHO

Universidade do Minho, 27/10/2013

Assinatura: André António dos Santos da Silva

Acknowledgements

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.

Abstract

Directed Evolution of Model-Driven SpreadSheets

Spreadsheets are among the most used programming languages today. The easy to use, 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 what 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 spreadsheet business logic, which offered 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 model-driven spreadsheets, it is intended to evolve an initial model, and respective instances, to a new model known in advance. The objective of this thesis is to present techniques that enable this evolution to be made automatically.

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 simples 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 o 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 ao objecto e desenvolvimento dirigido-por-modelos. Com estas técnicas passou a ser possível especificar a lógica de negócio de folhas de cálculo, o que proporcionou a estruturação dos dados nelas contidos e, ao mesmo tempo, limitar 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 folhas de cálculo dirigidas-por-modelos, pretende-se evoluir um modelo inicial, e respectivas instâncias, para um novo modelo conhecido à partida. O objectivo desta tese é apresentar um conjunto de técnicas que permitam fazer esta evolução de forma totalmente automática.

Contents

List of Figures

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].

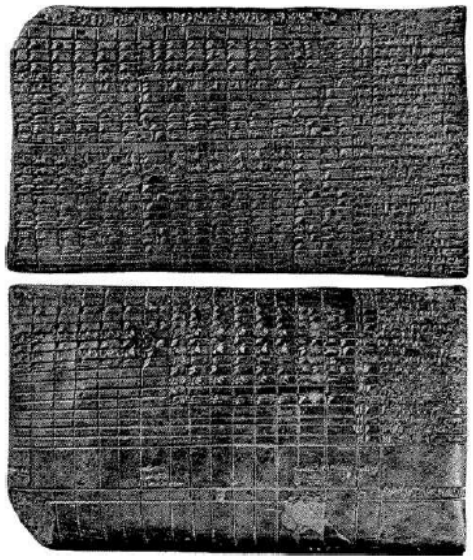


Figure 1.1: Month-by-month wage account for the Sumerian temple of Enlil at Nippur, for the year 1295 BC [Robson, 2007]

One of the first civilizations to use tables was the Sumerian. Sumerians employed tables to keep track of livestock and wage account. In figure 1.1 a tablet utilized to keep record

of monthly salaries of temple personnel for the Sumerian temple of Enlil [Robson, 2007] is shown. 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). This shows that there is still a lot of importance in displaying 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].

	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].

[1] Source: http://sites.stat.psu.edu/~mga/401/tables/binom.pdf. (19-09-2013)
[2] Source: http://www.ibrc.indiana.edu/ibr/2006/outlook/international.html. (19-09-2013)

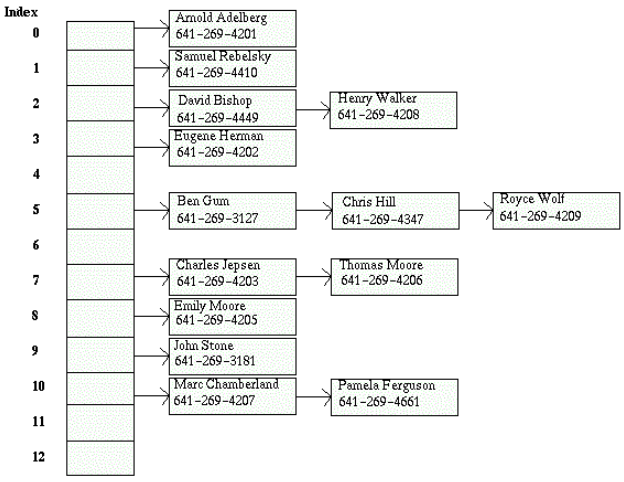


Figure 1.4: Storage of names and numbers in a Hash Table [3].

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.

1.1 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 up 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 accepted [Campbell-Kelly, 2007b].

While spreadsheets where 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

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

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

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].

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 lot of 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 on 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].

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

In 1984 Lotus 1-2-3 was released with major improvements, which included graphics generation, better performance and was more user friendly, this led it to dethrone VisiCalc as the number one spreadsheet system. It was only in 1990, when Microsoft Windows gained

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

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 over the internet were developed, like Microsoft Office 365 [7], and freeware 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 sequence of letters, 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 separated both by a colon, 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., 2012].

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

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

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]. 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%.

- 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].

EuSpRiG[11]website has many other examples were the economic impact can range from a few thousands to billions of dollars.

[11]Source:http://www.eusprig.org/horror-stories.htm(19-9-2013). (19-09-2013)

1.3 Model-Driven Spreadsheets

For a long time spreadsheet communities and tool vendors have tried to mitigate the problems previously described 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 focus 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 object-oriented [Engels and Groenewegen, 2000] and model-driven development [Kleppe, 2003].

1.3.1 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 was described by grouping cells together to form blocks that can be repeated vertically or horizontally. This process consisted of defining a model, designated as template, by means of a visual editor called ViTSL [Abraham et al., 2005]. ViTSL offered 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.
- Hex groups, represented by horizontal dots indicating the possibility of horizontal expansion of a group of cells.

The model created using ViTSL would then be passed as an argument to a tool, dubbed Gencel [Abraham et al., 2005], that would generate spreadsheets according to the defined model, and constrain the type of operations allowed to make over the instances so that they always complied with the underlying model. In figure 1.7 we can see a template for a budget spreadsheet defined using the ViSTL editor, in it, it can be observed that the block composed by the columns C, D and E can be repeated horizontally, and that line 4

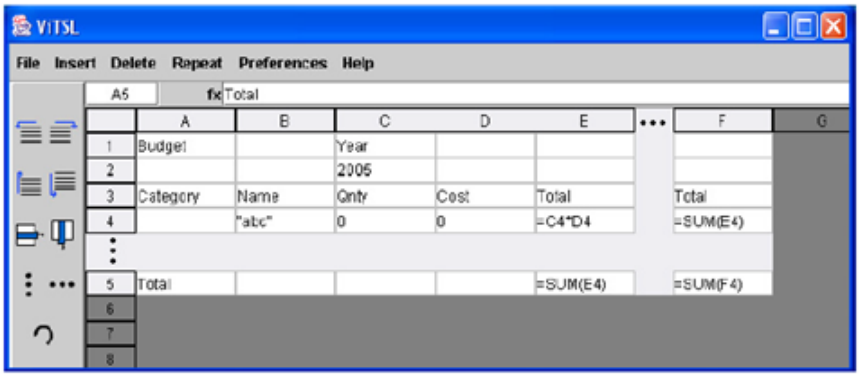


Figure 1.7: Template defined using ViSTL.

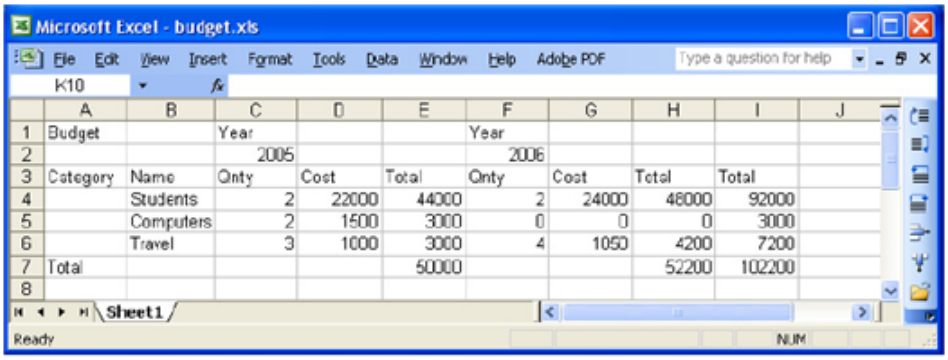


Figure 1.8: Automatically generated Gencel spreadsheet.

can be repeated vertically. In figure 1.8 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 that the block composed by line 4 has three occurrences.

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.

1.3.2 ClassSheets

Engels and Erwig improved upon templates by introducing ClassSheets [Engels and Erwig, 2005]. ClassSheets are a high-level model, based on object-oriented development, that allow 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 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 1.9 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.

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

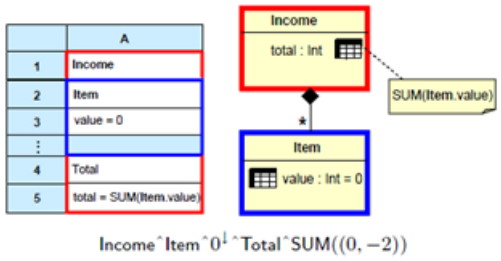
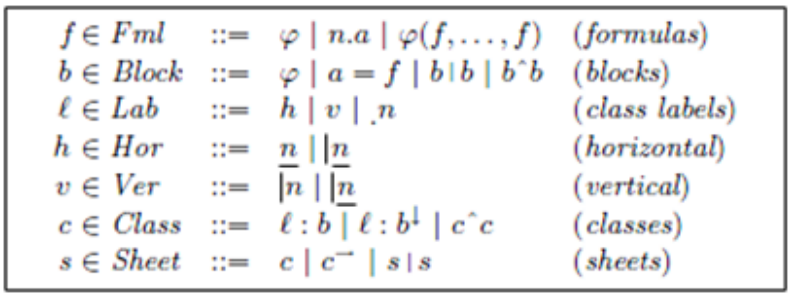


Figure 1.9: Income ClassSheet.

This formal representation allows specifying ClassSheet models textually using the abstract syntax presented in figure 1.10.

Using this syntax, a ClassSheet model can be defined in the following way:

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



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

Figure 1.10: ClassSheet Syntax.

- A Class is composed of 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 ($.n$) class label.

Using the ClassSheet model represented in figure 1.9 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 cells block with the value Item and marking it with a vertically repeatable label, that block is to be perceived as a vertically repeatable class, and is represented by the following syntax:

|**Item** : Item ^
|**Item** : (value=0)↓

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

|**Income** : Income ^
|**Item** : Item ^

|**Item** : (value=0)↓ ^
|**Income** : Total ^
|**Income** : total=SUM(Item.value)

The above example is the complete textual representation of the example depicted in figure 1.9. 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.

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 ^
|**Item** : Item ^
|**Income** : (value=0)↓ ^
|**Item** : Total ^
|**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 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 1.11.

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

Figure 1.11: ClassSheet tiling rules.

In figure 1.12 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.

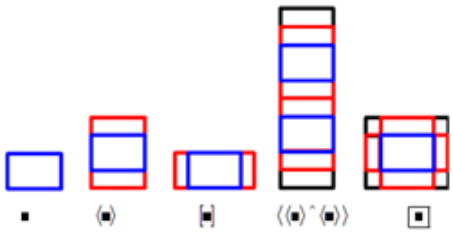


Figure 1.12: ClassSheet tiling structures.

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

1.3.3 Embedding of ClassSheet Models Within Spreadsheets

In [Mendes, 2011] a method to embed ClassSheet model specification in the same environment used to manipulate the instances was proposed. 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.

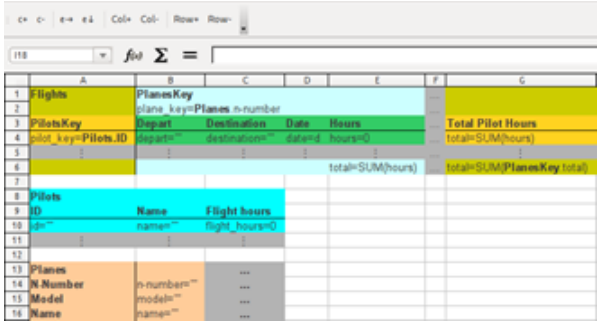
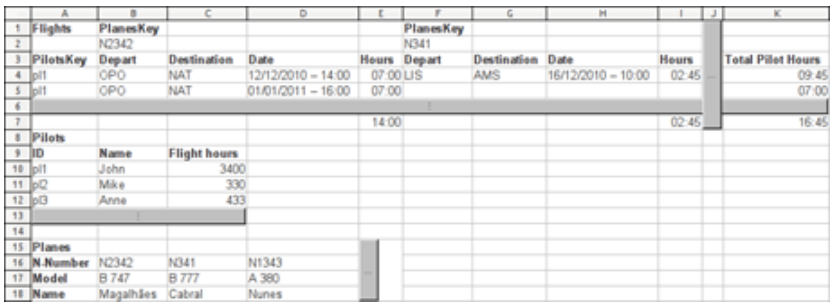


Figure 1.13: Embedded ClassSheet representing a system for an airline company as proposed in [Mendes, 2011].

In figure 1.13 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 1.14.



	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	390								
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 1.14: Spreadsheet instances for an airline company spreadsheet as proposed in [Mendes, 2011].

1.3.4 Bidirectional Transformation of Spreadsheets

A co-evolution technique, based on bidirectional transformations, was proposed in the context of the SSaaPP[12]project to synchronize ClassSheet models and instances [Cunha et al., 2012]. 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 editing operations over models to be transformed in editing operations over instances of that model and editing operations over instances to be transformed into editing operations over models. 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 1.15.

The set of edit operations defined over models, and displayed in figure 1.16, 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.

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

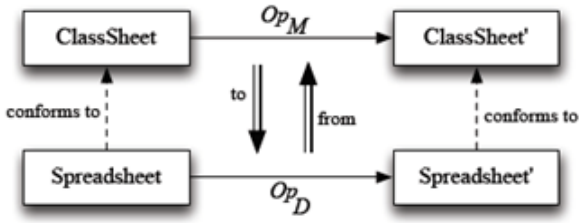


Figure 1.15: Spreadsheet bidirectional transformation system, as proposed in [Mendes, 2011].

```
data Op_M : Model → Model =
    addColumn_M   Where Index                    -- add a new column
  | delColumn_M          Index                   -- delete a column
  | addRow_M      Where Index                    -- add a new row
  | delRow_M             Index                   -- delete a row
  | setLabel_M    (Index, Index) Label           -- set a label
  | setFormula_M  (Index, Index) Formula         -- set a formula
  | replicate_M   ClassName Direction Int Int    -- replicate a class
  | addClass_M    ClassName (Index, Index) (Index, Index) – add a static class
  | addClassExp_M ClassName Direction (Index, Index) (Index, Index)
                                                 -- add an expandable class
```

Figure 1.16: Operarions over ClassSheet models, as proposed in [Cunha et al., 2012].

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.

1.4 Model-Driven Evolution

Large software companies, banking industry, 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 1.17.

In many real world applications of model-driven spreadsheets, different systems use different formats, so the data produced by one system has to be adapted to be consumed by a

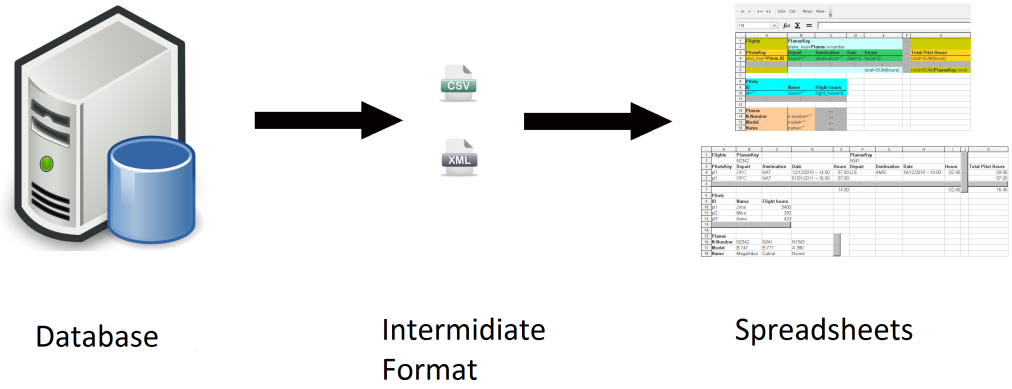


Figure 1.17: Data extraction and presentation flow.

target system. Usually in these cases, both the spreadsheet initial model and final model, that is supposed to be consumed by the target system, are known. The objective of this thesis is to take another step forward and propose techniques that allow the automatic evolution of spreadsheets that conform to the initial model so that they comply with the given final model.

The evolution method proposed, accomplishes ClassSheet model evolution at an automata level, this means that ClassSheets have to be represented using automatons, and to do this, three things are fulfilled:

1. Transformation from ClassSheet models to deterministic finite automaton is defined.
2. All editing operations defined in (Cunha et al., 2010) are defined over deterministic finite automatons.
3. A criterion is defined to identify which operations are to be applied to the initial model so that, with each transformation, the initial automaton converges to the final automaton.

From this point on, this thesis focus on these points, and 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 instances..

1.5 Document Structure

This thesis is structured as follows:

Chapter 2 Describes how ClassSheets will be represented as automata and how operations over ClassSheet models introduced in [Cunha et al., 2012] are related to automata operations.

Chapter 3 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 4 Describes the integration of the developed prototype with the MDSheet tool, and presents some tests.

Chapter 5 Concludes this thesis mentioning the work done, and with some suggestions for future work.

Chapter 2

Operations Over ClassSheet Automata

Methods for model-driven spreadsheet evolution were already proposed [Cunha et al., 2012], 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.

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 1.9 is shown in figure 2.1.

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 (^). On the second row apears once again the

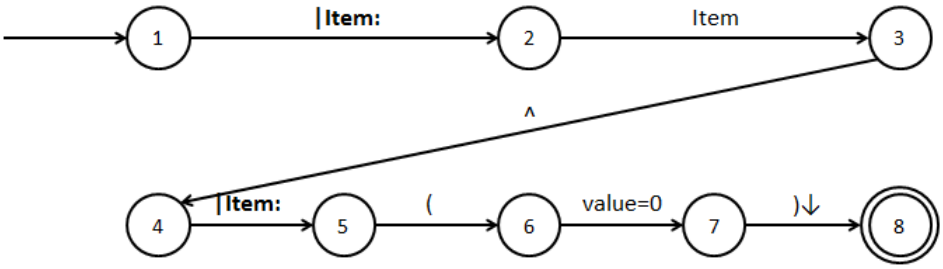


Figure 2.1: Item class expressed as automaton.

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.

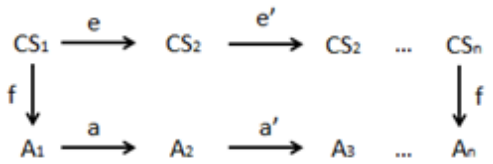


Figure 2.2: Evolution process.

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., 2012] 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 2.2 one can visualize how the expected evolution process will progress. First the initial model (CS_1) has to be converted to an automaton (A_1), using an appropriate conversion function (f), transformation operations at automata level (a^x) will ensue until the given final model is attained (A_n). The important thing to notice is that these operations are directly related to operations defined over ClassSheet models (e^x) in [Cunha et al., 2012]. 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 (CS_1) can be done by using bidirectional transformations defined in [Cunha et al., 2012].

2.1 ClassSheets as Automata

As already observed, representing ClassSheets as automata is a simple exercise of sequentially converting the formal language syntactic elements to automata transitions; however the formal language has some drawbacks. When representing a ClassSheet model with a horizontal aggregation over two horizontal aggregations, 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 2.3 and figure 2.4, 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 2.3: Two-dimensional example of a ClassSheets, as seen in [Engels and Erwig, 2005].

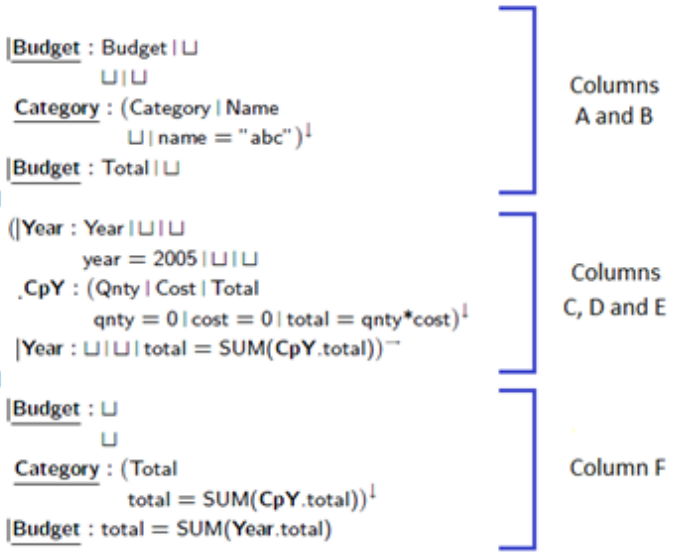


Figure 2.4: Textual representation of the Budget ClassSheet.

To formally define the budget ClassSheet model in figure 2.3, one has to describe the model as horizontally composing blocks; these blocks are duly identified in figure 2.4. 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 3 a different representation, still based on the ClassSheet formal representation, is required to allow row-by-row processing of ClassSheet models.

Another important aspect to remark is that, from now on, all ClassSheets considered in this thesis are embedded ClassSheets. 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. In figure 2.5 the embedded version of the **Budget** ClasshSeet pictured in figure 2.3 is presented, the differences are clearly visible, class colors are applied to the entire class, not only to the outlines, and in the embedded version column G and row 5 are reserved for marking horizontal and vertical repetitions, respectively, while on the non-embedded version this markings are not part of the spreadsheet. This is important to emphasize since repetition markings are also present in the new automata representation.

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

Figure 2.5: Embedded version of the Budget ClassSheet.

To address the former limitations a new abstract syntax, still based on the original ClassSheet textual syntax, is introduced. This formal representation allows sequentially defining a ClassSheet model, row-by-row, starting on the upper-left corner and ending on the lower-right corner.

A ClassSheet (s) consists of a row or a vertical composition of rows, each row (r) is comprised of a class or a horizontal composition of classes, each class (c) is identified by a name (n), and holds a block with its contents. Besides non-expandable, a block (b) can be vertically or horizontally expandable, and may be a single cell holding a value (φ), possibly empty ($\sqcup$), an attribute definition ($a = f$), a horizontal or vertical composition of cells.

$f \in Fml \quad ::= \varphi \mid n.a \mid \varphi(f,\ldots,f)$ *(formulas)*

$b \in Block ::= \varphi \mid a = f \mid b|b \mid b\hat{}b$ *(blocks)*

$l \in Label \quad ::= n$ *(class labels)*

$c \in Class ::= l{:}\,b \mid l{:}\,b^{\rightarrow} \mid l{:}\,b^{\downarrow}$ *(classes)*

$r \in Row \quad ::= c \mid r|r$ *(rows)*

$s \in Sheet ::= r \mid s\hat{}s$ *(sheets)*

Figure 2.6: Abstract syntax used to represent ClassSheet automata.

Finally, a formula (f) can be a single value (φ) or a qualified class name and attribute ($n.a$). The abstract syntax is summarized in figure 2.6.

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

The first example (figure 2.7) 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 2.7: Pilots ClassSheet.

The row-by-row textual specification of this ClassSheet model 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 (^) of rows. The second row has a similar structure to the first one, 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 ((()↓), this vertically repeatable block 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 (^). The fourth row holds the second block of the previous vertical composition, this block contains 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 syntax, 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 2.8). As does the formal 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 2.9 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.

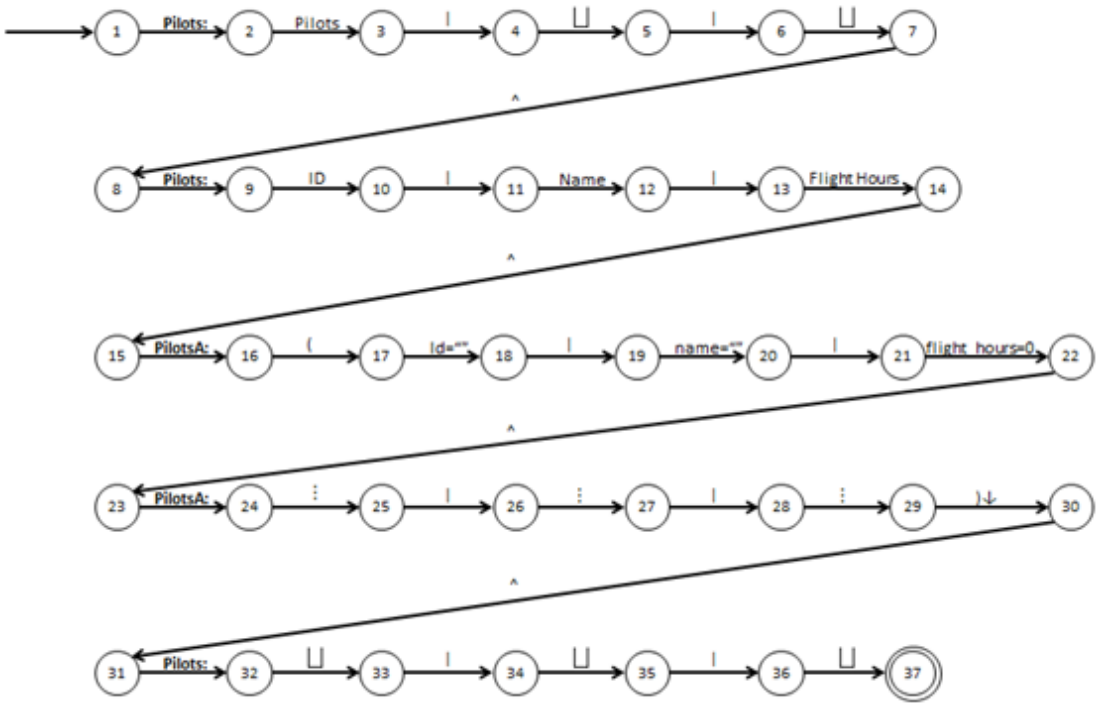


Figure 2.8: Pilots ClassSheet automaton.

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

Figure 2.9: Planes ClassSheet.

To textually specify the **Planes** ClassSheet, in a row-by-row way, the aggregated class PilotsA 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 Pilots ClassSheet.

Planes: Planes | **PlanesA:** (⊔ | ⋯)→ | Planes: ⊔ ^
Planes: N-Number | **PlanesA:** (n-number="" | ⋯)→ | Planes: ⊔ ^
Planes: Model | **PlanesA:** (model="" | ⋯)→ | Planes: ⊔ ^
Planes: Name | **PlanesA:** (name="" | ⋯)→ | Planes: ⊔ ^

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 $(\cdots)$, 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 2.10.

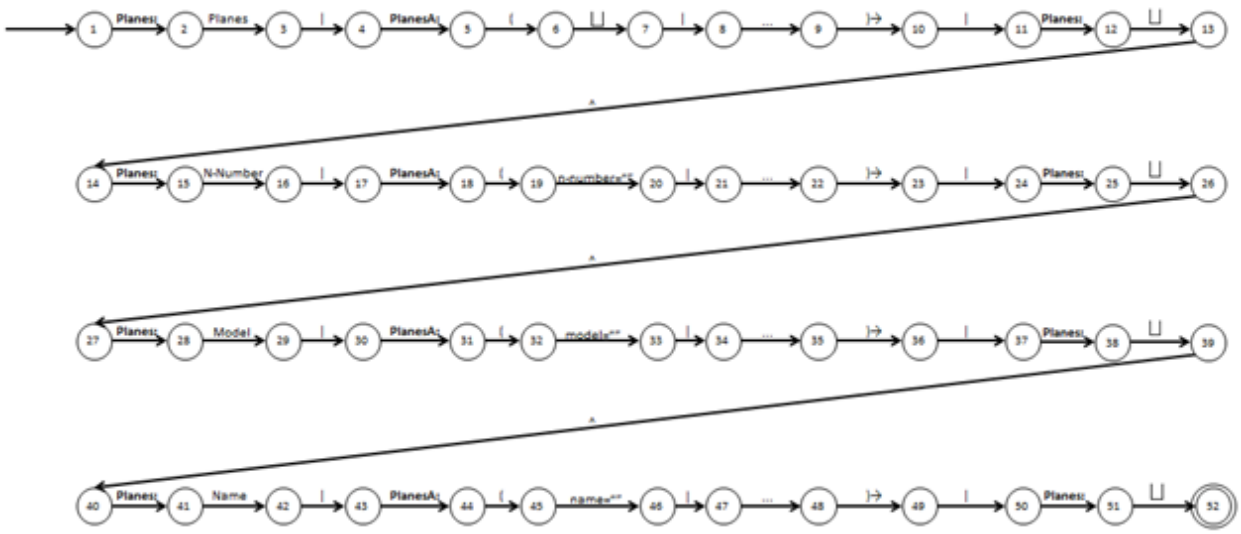


Figure 2.10: Planes ClassSheet automaton.

This transformation method between ClassSheet models and automata, establishes what task is to be executed by the conversion function (f) in figure 2.2.

2.2 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.

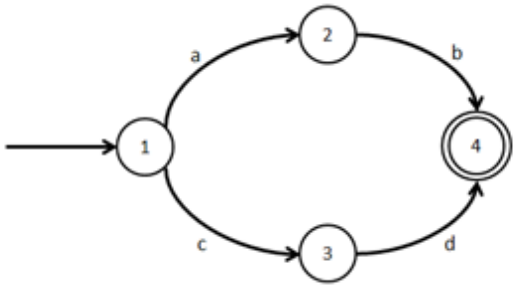


Figure 2.11: Generic automaton.

2.2.1 Add Transition

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 2.11 is presented in figure 2.12.

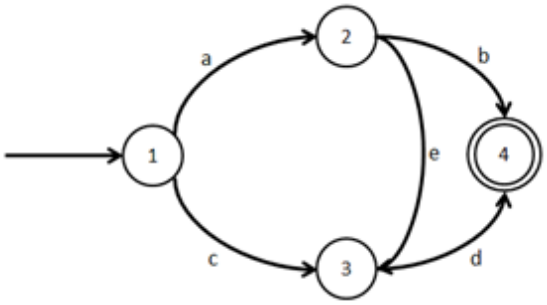


Figure 2.12: Generic automaton with a new transitions.

2.2.2 Delete Transition

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 2.12 results in the automaton pictured in figure 2.11.

2.2.3 Edit Transition

Edits a transition in an automaton, 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 2.12 to connect states 3 and 1, by symbol f, is presented in figure 2.13.

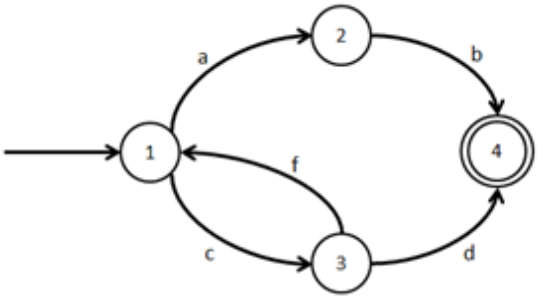


Figure 2.13: Generic automaton with an edited transitions.

2.2.4 Add State

Adds a state without any transitions to an automaton. Adding a state to the automaton in figure 2.11 results in the automaton in figure 2.14.

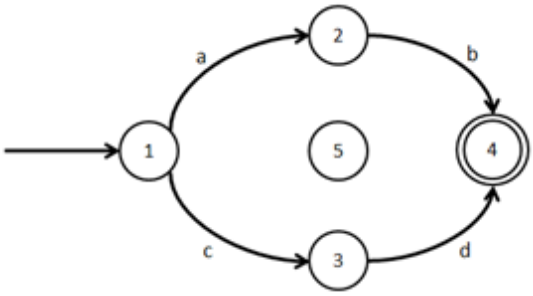


Figure 2.14: Generic automaton with a new state.

2.2.5 Remove State

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 2.11 results in the automaton in figure 2.15.

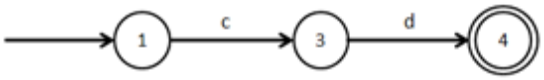


Figure 2.15: Generic automaton with a state removed.

2.3 ClassSheet Automata Atomic Operations

In this section, equivalent operations over ClassSheet models proposed in [Cunha et al., 2012] 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 2.16. 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, 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 result in automaton (A_n).

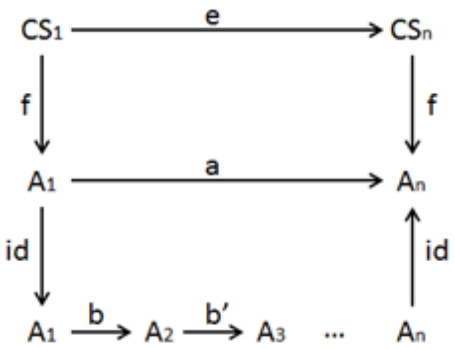


Figure 2.16: ClassSheet atomic operations as a sequence of basic operations.

Atomic operations share a same basic strategy, an automaton is traversed, transition-by-transition, while at the same time keeping track of the spreadsheet coordinates associated with each transition. 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 (^) occurs.

In figure 2.18 the automaton that represents the **Item** ClassSheet model in figure 2.17 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 (^) 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, at that point the y coordinate is incremented again. This simple example has only one column, in case it had more than one, each time a horizontal transition was reached (|), the x coordinate had to be incremented instead.

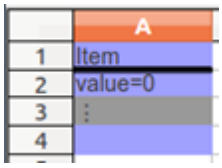


Figure 2.17: Embedded Item ClassSheet.

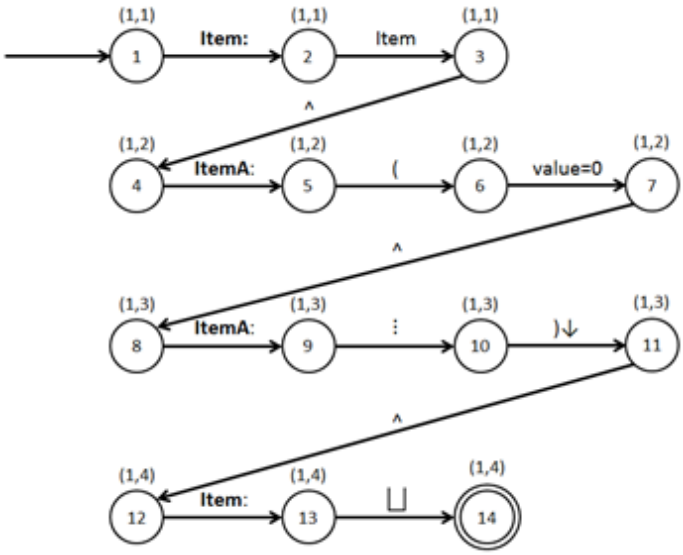


Figure 2.18: Embedded Item ClassSheet automaton.

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

2.3.1 Add Column

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., 2012].

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

This operation takes as parameters, a column that acts as a reference and the position, before or after that column, where the new one is to be added. 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 reference 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 with value c, the following sequence of basic operations has to be applied:

1. Add a new state.
2. Add a new state.
3. Edit the transition that precedes the cell transition in column (c) so that it transitions to the state added in step 1.
4. Add a transition representing an empty cell connecting the state added in step 1 to the state added in step 2.
5. Add a transition connecting the state created in step 2 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 2.18, 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 equal to 1. The first transition that meets this criterion is the one that transitions from state 2 to state 3 by the symbol Item, figure 2.19 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.

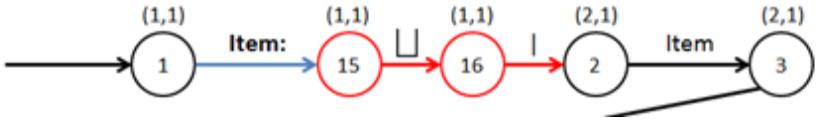


Figure 2.19: Add individual cell before column x.

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 2.20. One important matter to mention is that cells in row 3 have a special meaning and are used to mark the vertical expansibility of the class, so instead of adding an empty cell, a cell of the same type must be added.

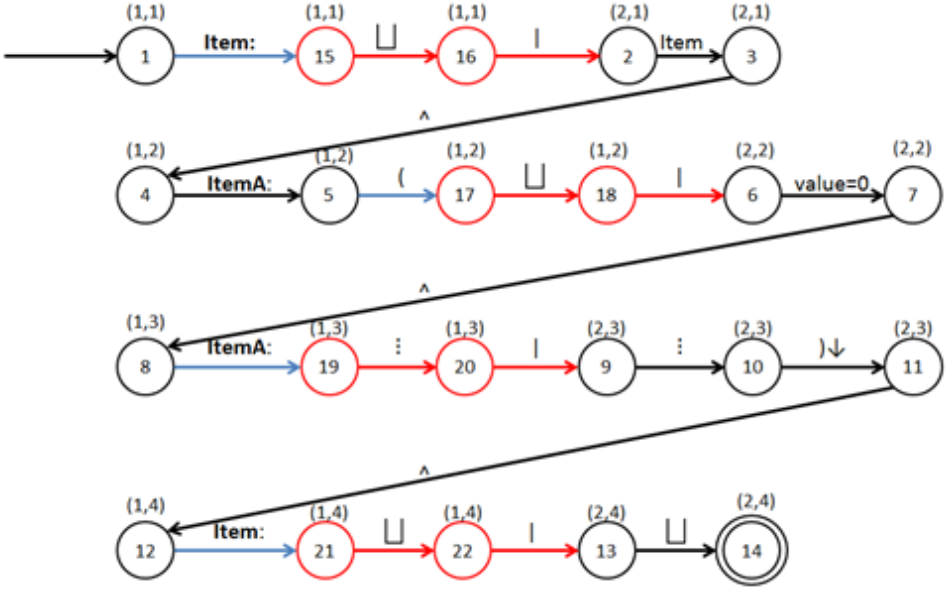


Figure 2.20: Add column before 1, operation.

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

1. Add a new state.

2. Add a new state.

3. Edit the transition that succeeds the cell transition at column c so that it succeeds the state added in step 2 instead.

4. Add a new transition denoting a horizontal composition after the cell transition at column c and connect it to the state added in step 1.

5. Add a new transition, representing an empty cell, connecting the state added in step 1 to the state added in step 2.

Adding a column after column 1 to the automata in figure 2.21 shares the same principle as the previous case, on all transitions that represent cells and are located at coordinates with an x value of 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, 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 2.21, where the edited transition is blue colored and added states and transitions are red colored.

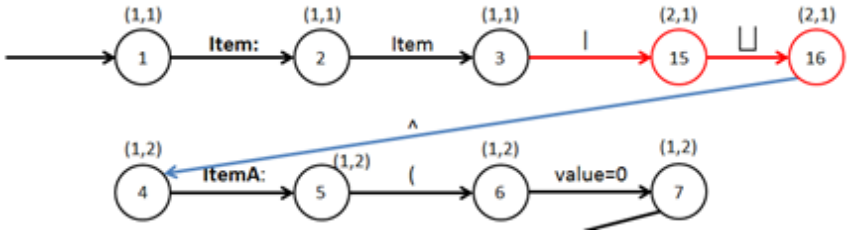


Figure 2.21: Add individual cell after x, operation.

This sequence of operations is applied on every occurrence of a cell transition resulting in the automaton displayed in figure 2.22.

2.3.2 Delete Column

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., 2012].

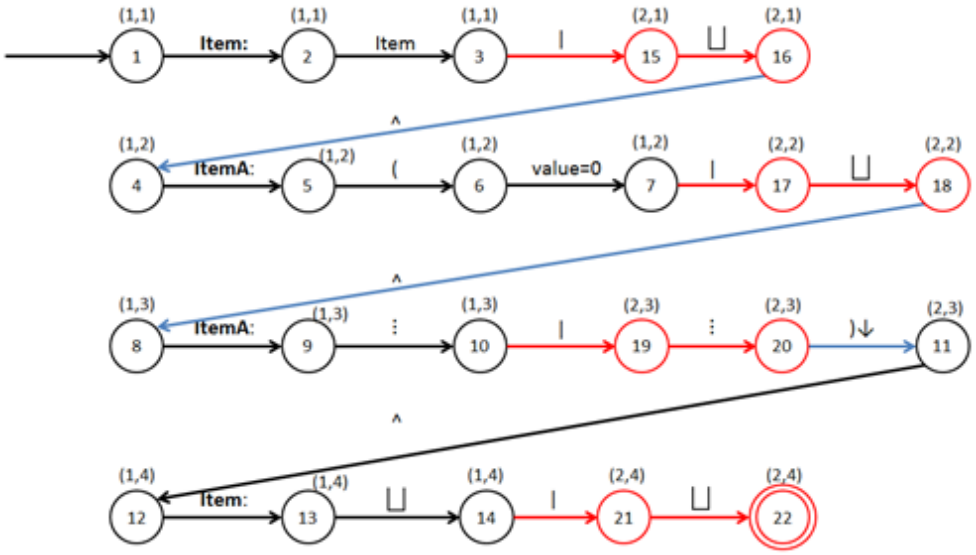


Figure 2.22: Add column after 1, operation.

$delColumn_M :: Index \rightarrow Op_M$

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 an x component with value c, the following basic operations are applied.

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 2.22. This atomic operation consists in traversing the automata and applying the sequence of basic operations previously mentioned each time a cell transition has x coordinate with value 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 2.23. 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

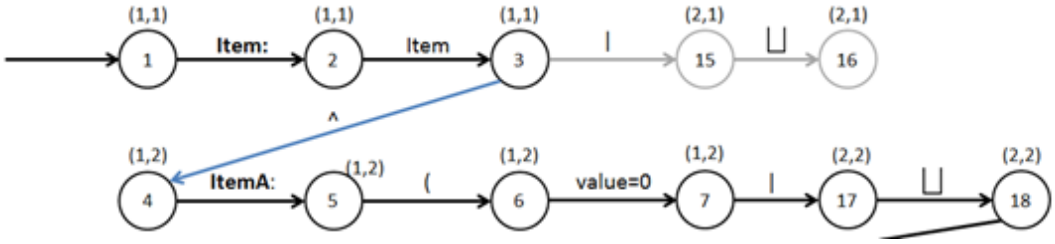


Figure 2.23: Cell removed from Item ClassSheet.

horizontal composition symbol, respectively. The edited transition is colored in blue and the removed states and transitions are marked in grey.

The result of employing this process in all subsequent cell transitions located at column 2 is the transformation of the automaton in figure 2.22 into the automaton in figure 2.18.

2.3.3 Add Row

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., 2012].

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

This operation takes as input, a reference row and the position, before or after that row, where the new one is to be added. Besides being used to determine the location, the reference row is also 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 reference row (r). Generically, this transformation is carried out in the following steps.

1. The automaton is traversed until the first transition with y coordinate with value 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 end of the row structure added in step 2 and the transition identified in step 1.

4. The transition that precedes the one identified in step 1 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 2.18. The first step is to identify the first transition with y coordinate with value 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 2.24), this means that the new row has the same structure.

Figure 2.24: Item ClassSheet row 3 structure.

The third step is to add a transition, with a vertical composition, 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 2.25.

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

1. The automaton is traversed until the last transition with y coordinate with value 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 the transition identified in step 1 and the first state in the row structure added in step 2.

4. The transition that originates in the last state of the transition identified in step 1 is edited so it originates in the last state of the row structure added in step 2.

The ensuing example allows for a precise understanding of how this atomic operation is applied.

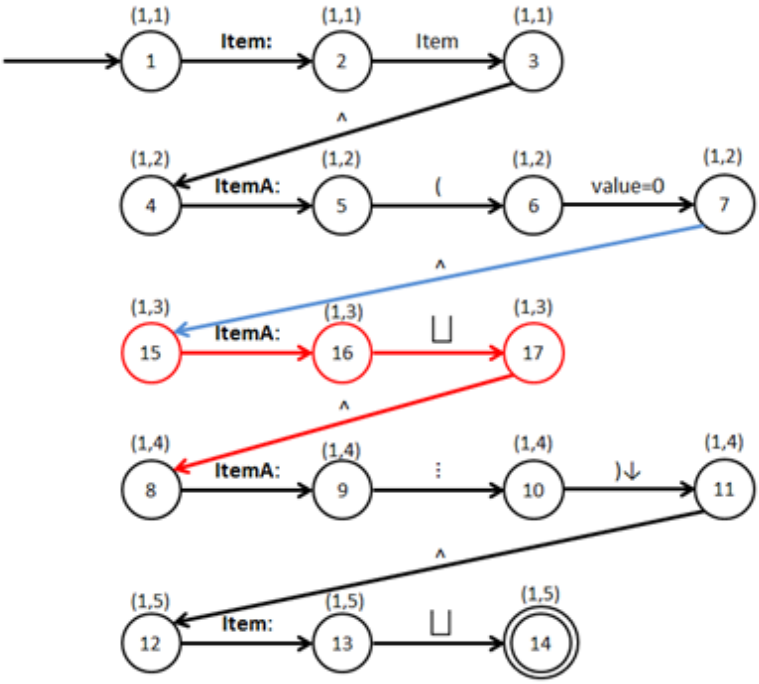


Figure 2.25: Item ClassSheet after Add Row Before 3 operation.

Suppose one wants to apply the operation add row after 4 to the automaton in figure 2.18. The first step is to identify the last transition with y coordinate with value 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 2.26), this means that the new row has the same structure.

Figure 2.26: 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 2.18 is shown in figure 2.27, with the added states and transitions colored in red.

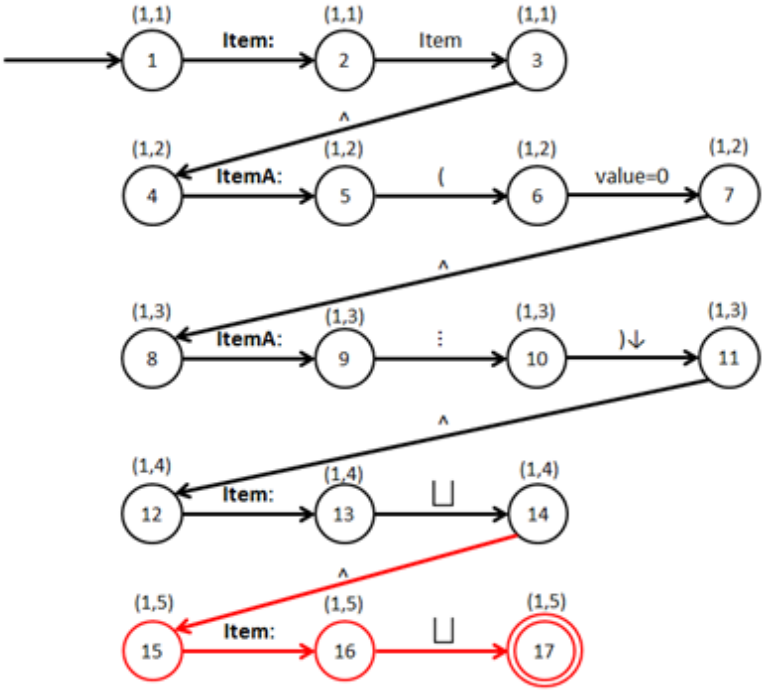


Figure 2.27: Item ClassSheet after Add Row After 4 operation.

2.3.4 Delete Row

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

$delRow_M :: Index \rightarrow Op_M$

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

1. The automaton is traversed until the first transition with y coordinate with value r is reached.

2. The transition that precedes the transition found in step 1, 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 2.25. The first step is to locate the first transition in row 3, 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, this concludes step 2. Finally all states in row 3 are removed, that comprises all the states that are colored in red. The result of removing row 3 from the automaton is pictured in figure 2.18.

2.3.5 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., 2012].

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

Part of the work done in this thesis was to develop a prototype that supports automatic evolution of ClassSheet models and instances, with the objective of being integrated in MDSheet. In MDSheet both ClassSheet model operations are unified 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 that has coordinates with value 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 2.18, the automaton has to be traversed until the transition of type cell with coordinates with value c 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 2.28 with the edited transition colored in blue.

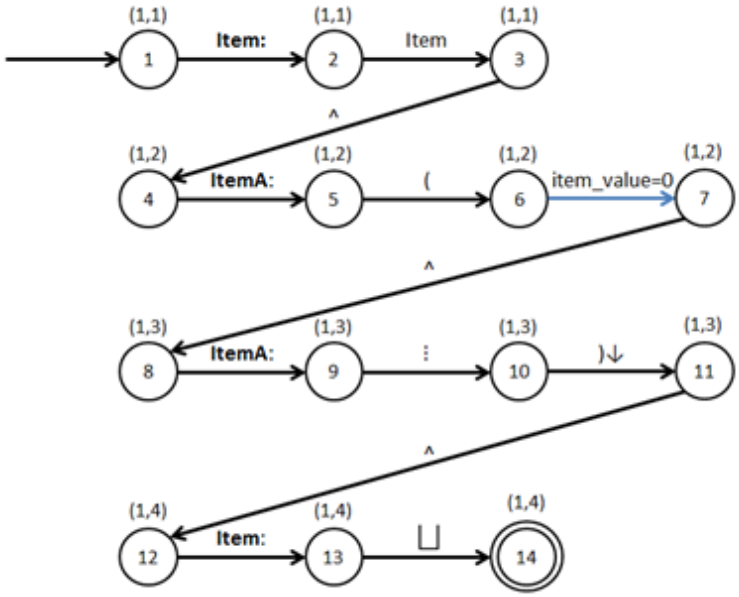


Figure 2.28: Item ClassSheet after Set Cell (1,2) item_value operation.

2.4 Summary

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

A set of operations over generic automata is 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., 2012] are defined over automata, these operations establish a base to support model evolution of ClassSheets expressed as automata.

Chapter 3

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, and that can involve substantial financial costs.

3.1 Model-Driven ClassSheet Evolution

In many real world applications of model-driven spreadsheets, both the initial and final models are known in advance. In this chapter a technique is presented that allows automatically evolving a given initial ClassSheet model, and co-evolving 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 2, and bidirectional transformations of model-driven spreadsheets proposed in [Cunha et al., 2012]. The method presented, and visualized in figure 2.2,

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., 2012], 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 be applied a criterion, based on the modifications that a sequence of transformations performs on data, is used. The sequence that requires fewer changes in cells with formulas or values is selected. 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 2 and illustrated in figure 3.1, 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).

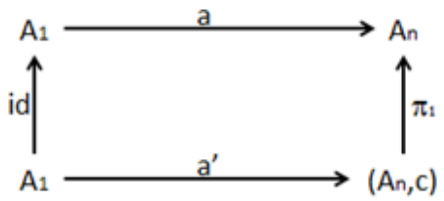


Figure 3.1: Quantified atomic operations.

The values present in table 3.1 are used to calculate the cost of an operation.

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

Table 3.1: Costs per operation.

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. On the other hand, if the data present on a cell is a formula or a value, that cost has a value of 10, this is due to the fact that this data is potentially more valuable in any decision making process.

3.1.1 Transformations Sequences Algorithm

The process of ascertaining the sequence of transformations to be applied consists in traversing both 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, based on both transitions, is taken 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, this culminates in a valid transformation sequence.

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

Systematically evolving a source automaton to a target automaton, in a directed way until equivalence is met, is based on a fundamental principle, 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 disregarded. 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 is converging.

Now the cases that can occur when comparing two 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 3.2, arises when the transitions in the source automaton and final automaton are equal.

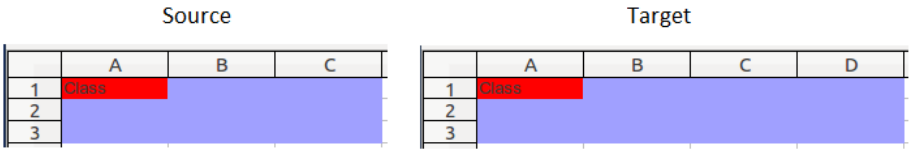


Figure 3.2: Source and target models, with equal transitions before evaluation.

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 3.3.

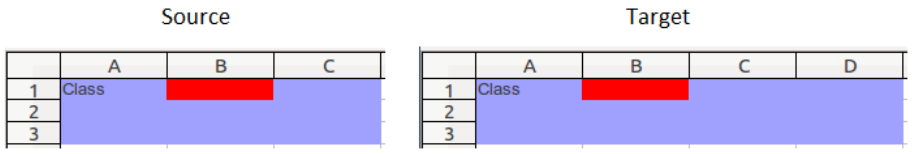


Figure 3.3: Source and target models, with equal transitions, after evaluation.

The next case, displayed in figure 3.4, 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.

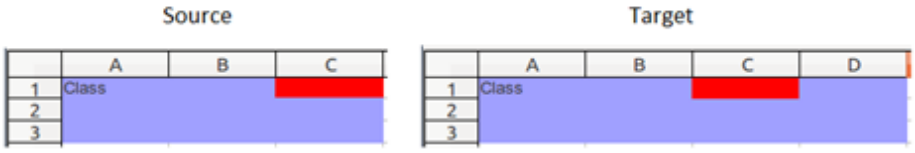


Figure 3.4: Target automaton has one more column than source automaton, before evaluation.

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 3.5.

The following situation (figure 3.6) 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.

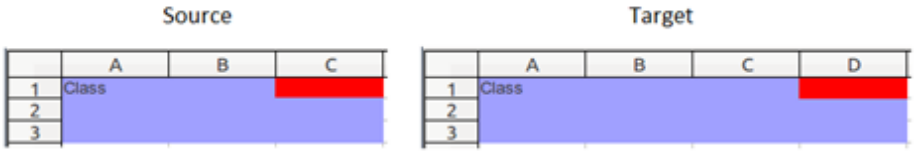


Figure 3.5: Target automaton with one more column than source automaton, before evaluation.

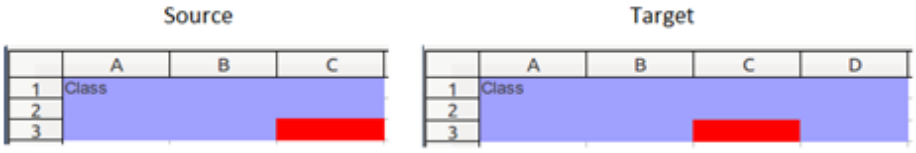


Figure 3.6: Source automaton in last row and target automata has, at least,one more column, after evaluation.

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 3.7.

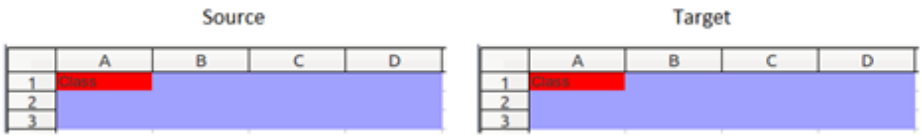


Figure 3.7: Source automaton with new added column.

The case pictured in figure 3.8 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.

In this case the source automaton transition is skipped to the next one. The result is displayed in figure 3.9.

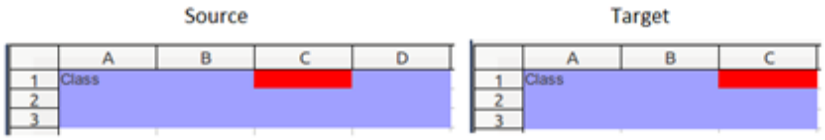


Figure 3.8: Source automaton has, at least, one more column than the target automaton.

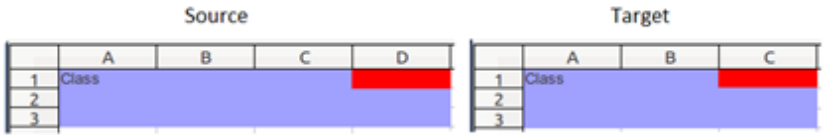


Figure 3.9: Source automaton synchronizes to target model.

The ensuing case, shown in figure 3.11, 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.

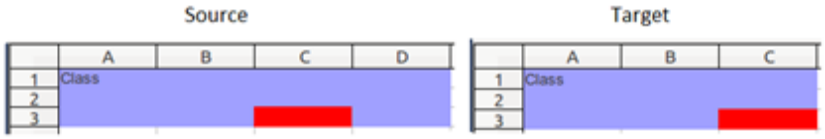


Figure 3.10: Source automaton reached last row and has, at least, one more column.

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.
- The automata are compared again.

The result is presented in figure 3.11.

The case depicted in figure 3.12 occurs when the source automaton reaches its end but the target automaton still has, at least, one more row.

In this situation the following steps are taken.

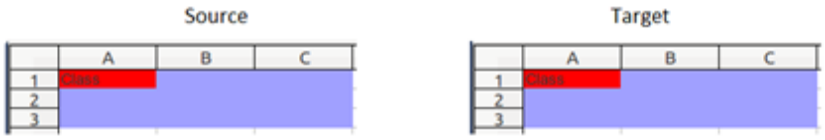


Figure 3.11: Source automaton with last column deleted.

Figure 3.12: Target automaton has, at least, one more row than the source automaton.

- Add 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 3.13.

Figure 3.13: Source automaton has one more row.

The next scenario (figure 3.14) occurs when the target automaton reaches the end the block of cells, but the source automaton has, at least, one more row.

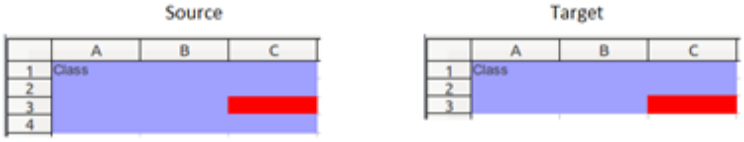


Figure 3.14: Source automaton has one more row, before evaluation.

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 had any more rows, the algorithm would proceed to find any more differences. The result is shown in figure 3.15.

Figure 3.15: Source automaton with deleted row, after evaluation.

The subsequent case to be addressed (figure 3.16), is the case where two cells have different content. Due to the fact that to determine the exact operation to be applied is not always a trivial thing to do, multiple operations are applied to the source automaton, which potentially results in numerous transformations sequences.

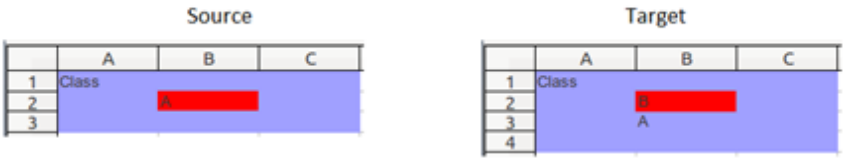


Figure 3.16: Cells holding different content.

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

- Apply the add column before x to the initial automaton and update the transformations sequence, and cost.
- Recursively compute the rest of the tranformations sequence and respective cost.

- Repeat the process 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.

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.

Finally is the case where 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 chose 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., 2012]. In chapter 4 some examples are presented to demonstrate how this process can be used to evolve model-driven spreadsheets.

3.2 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. To this to be possible a criterion, based on data modification effort, is defined.

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 4

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.

4.1 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., 2012], 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.

4.2 Tests

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[1] 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 sheet, using model editing operations already supported by MDSheet. When the model is created an instance is simultaneously generated in a second sheet, the second step is to modify that instance, using operations over instances, and fill it with data. In figure 4.1 a Pilots ClassSheet model and respective instance, created with MDSheet can be visualized.

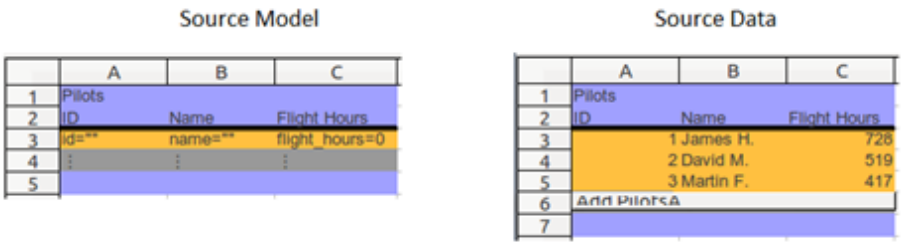


Figure 4.1: PilotsClassSheet model and instance before evolution.

To evolve the model and instance previously defined, one must create the target model (figure 4.2) in a separated sheet, select the source model and press the "Evolve To" control (figure 4.3). In this case the target model possesses a new column, after column C, with a new attribute, base_salary with default value 10000.

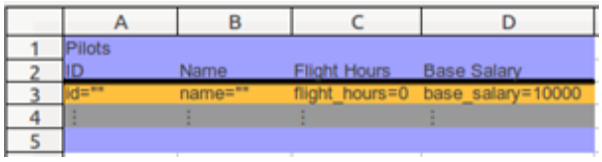


Figure 4.2: Pilots target model.

[1]Source:http://ark.intel.com/products/27235/Intel-Core-Duo-Processor-T2400-2M-Cache-1_83-ghz-667-mhz-fsb. (27-10-2013)

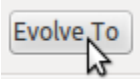


Figure 4.3: Evolve To control.

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, this includes 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 4.4.

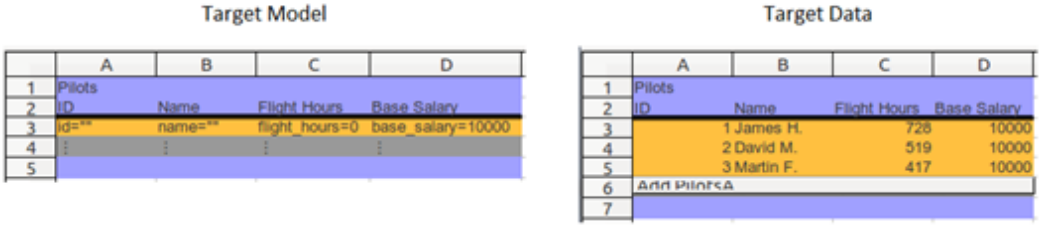


Figure 4.4: Pilots ClassSheet model and instance after evolution.

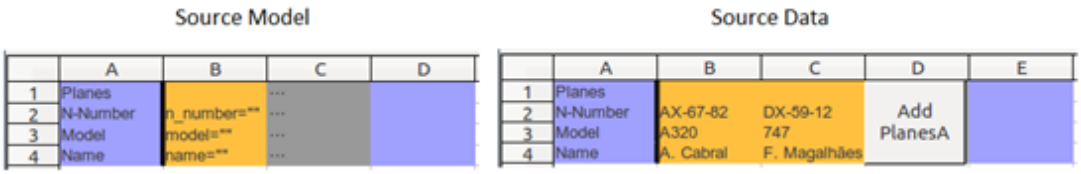


Figure 4.5: Planes ClassSheet model and instance after evolution.

In the next example a Planes ClassSheet model is presented alongside with a conforming instance (see figure 4.5). Both model and instance are to be evolved so that both comply with the model in figure 4.6, this model possesses two new rows with attributes airline and number_of_seats respectively.

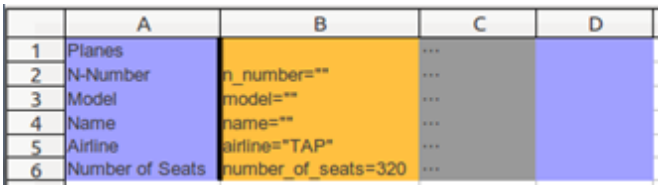


Figure 4.6: Planes target model.

The total computational time to evolve this example is 1 minute and 17 seconds, and the result, with both model an instance conforming to the target model, is presented in figure 4.7.

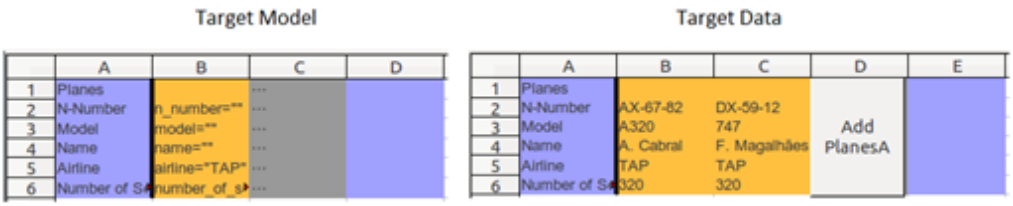


Figure 4.7: Planes ClassSheet model and instance after evolution.

The last example is a more complex, two-dimensionally expansible, Budget ClassSheet. The source model and instance are exhibited in figure 4.8 and Figure 4.9, respectively.

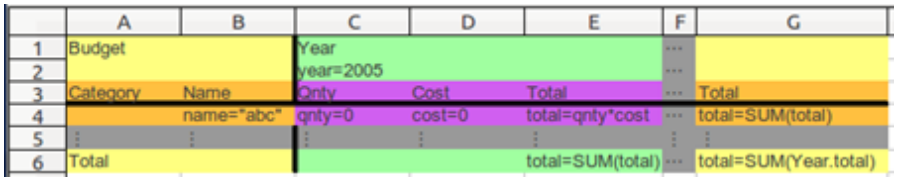


Figure 4.8: Budget source model before evolution.

Figure 4.9: Budget source instance before evolution.

Both, model and instance are to be evolved until they comply with the target model pictured in figure 4.10 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.

This example is one of the more computational intensives, the total time taken to calculate the transformations sequence and evolve model and instance is 7 minutes and 26 seconds. The resulting model is shown in figure 4.11, and the resulting instance is show in figure 4.12.

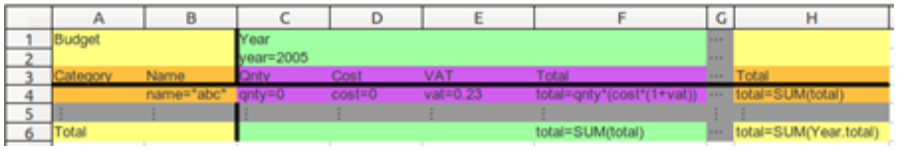


Figure 4.10: Budget target model.

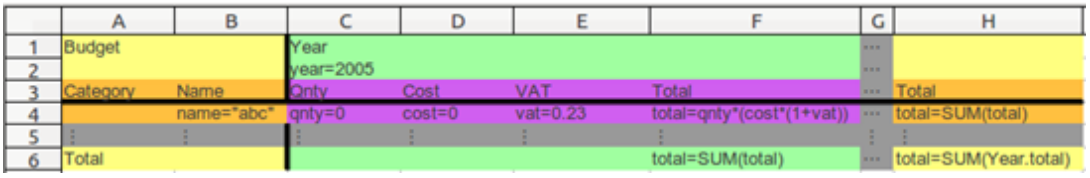


Figure 4.11: Budget source model after evolution.

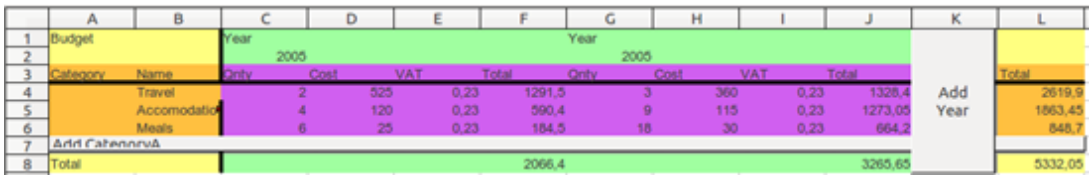


Figure 4.12: Budget source instance after evolution.

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.

4.3 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.

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

Chapter 5

Conclusion

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, but with no relevant success.

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.

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, due to the human factor sometimes errors are inadvertently introduced.

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 allows seamlessly bridging two systems, in a fully automatic way, i.e., removing the need of human intervention.

At this stage the developed prototype only supports evolution of models with different cell contents or different physical dimensions, be it 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.

For more information about spreadsheet development please visit the SSaaPP – Spread-Sheets as a Programming Paradigm web page:

http://ssaapp.di.uminho.pt

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

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, 5

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, J., Fernandes, J. P., Mendes, J., Pacheco, H., and Saraiva, J. (2012). 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, 13, 14, 16, 17, 18, 27, 29, 31, 33, 36, 37, 38, 39, 40, 47, 49

Engels, G. and Erwig, M. (2005). Classsheets: Automatic Generation of Spreadsheet Applications from Object-Oriented Specifications. In *Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering*, ASE '05, pages 124–133. ACM. 7, 9, 11, 19, 55

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. 7

Kleppe, A., W. J. (2003). *MDA Explained: The Model Driven Architecture Practice and Promise*. Addison-Wesley. 7

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. 12, 13, 14

Panko, R. R. and Ordway, N. (2008). Sarbanes-Oxley: What About all the Spreadsheets? *CoRR*, abs/0804.0797. 5, 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-swimmi
html (last retrieved: 19-09-2013). 6