Spreadsheet Engineering

Jácome Cunha[1,2], João Paulo Fernandes[1,3], Jorge Mendes[1,2], and João Saraiva[1](✉)

[1] HASLab/INESC TEC, Universidade do Minho, Braga, Portugal
{jacome,jpaulo,jorgemendes,jas}@di.uminho.pt
[2] CIICESI, ESTGF, Instituto Politécnico do Porto, Porto, Portugal
{jmc,jcmendes}@estgf.ipp.pt
[3] Reliable and Secure Computation Group ((rel)ease), Universidade da Beira Interior, Covilhã, Portugal
jpf@di.ubi.pt

Abstract. These tutorial notes present a methodology for spreadsheet engineering. First, we present data mining and database techniques to reason about spreadsheet data. These techniques are used to compute relationships between spreadsheet elements (cells/columns/rows), which are later used to infer a model defining the business logic of the spreadsheet. Such a model of a spreadsheet data is a visual domain specific language that we embed in a well-known spreadsheet system.

The embedded model is the building block to define techniques for model-driven spreadsheet development, where advanced techniques are used to guarantee the model-instance synchronization. In this model-driven environment, any user data update has to follow the model-instance conformance relation, thus, guiding spreadsheet users to introduce correct data. Data refinement techniques are used to synchronize models and instances after users update/evolve the model.

These notes briefly describe our model-driven spreadsheet environment, the MDSheet environment, that implements the presented methodology. To evaluate both proposed techniques and the MDSheet tool, we have conducted, in laboratory sessions, an empirical study with the summer school participants. The results of this study are presented in these notes.

1 Introduction

Spreadsheets are one of the most used software systems. Indeed, for a non-professional programmer, like for example, an accountant, an engineer, a manager, etc., the programming language of choice is a spreadsheet. These *programmers* are often referred to as *end-user programmers* [53] and their numbers are increasing

This work is part funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within projects FCOMP-01-0124-FEDER-010048, and FCOMP-01-0124-FEDER-020532. The first author was funded by FCT grant SFRH/BPD/73358/2010.

V. Zsók et al. (Eds.): CEFP 2013, LNCS 8606, pp. 246–299, 2015.
DOI: 10.1007/978-3-319-15940-9_6

rapidly. In fact, they already outnumber professional programmers [68]! The reasons for the tremendous commercial success that spreadsheets experience undergoes continuous debate, but it is almost unanimous that two key aspects are recognized. Firstly, spreadsheets are highly flexible, which inherently guarantees that they are intensively multi-purpose. Secondly, the initial learning effort associated with the use of spreadsheets is objectively low. These facts suggest that the spreadsheet is also a significant target for the application of the principles of programming languages.

As a programming language, and as noticed by Peyton-Jones et al. [45], spreadsheets can be seen as simple functional programs. For example, the following (spreadsheet) data:

```
A1 = 44
A2 = (A1-20)* 3/4
A3 = SUM(A1,A2)
```

is a functional program! If we see spreadsheets as a functional program, then it is a very simple and flat one, where there are no functions apart from the built-in ones (for example, the `SUM` function is a predefined one). A program is a single collection of equations of the form "variable = formula", with no mechanisms (like functions) to structure our code. When compared to modern (functional) programming languages, spreadsheets lack support for abstraction, testing, encapsulation, or structured programming. As a result, they are error-prone: numerous studies have shown that existing spreadsheets contain too many errors [57,58,62,63].

To overcome the lack of advanced principles of programming languages, and, consequently the alarming rate of errors in spreadsheets, several researchers proposed the use of abstraction and structuring mechanisms in spreadsheets: Peyton-Jones et al. [45] proposed the use of user-defined functions in spreadsheets. Erwig et al. [29], Hermans et al. [39], and Cunha et al. [19] introduced and advocate the use of models to abstractly represent the business logic of the spreadsheet data.

In this tutorial notes, we build upon these results and we present in detail a Model-Driven Engineering (MDE) approach for spreadsheets. First, we present the design of a Visual, Domain Specific Language (VDSL). In [29] a domain specific modeling language, named ClassSheet, was introduced in order to allow end users to reason about their spreadsheets by looking at a concise, abstract and simple model, instead of looking into large and complex spreadsheet data. In fact, ClassSheets offer to end users what API definitions offer to programmers and database schemas offer to database experts: an abstract mechanism to understand and reason about their programs/databases without having to look into large and complex implementations/data. ClassSheets have both a textual and visual representation, being the later very much like a spreadsheet! In the design of the ClassSheet language we follow a well-know approach in a functional setting: the embedding of a domain specific language in a host functional language [44,70]. To be more precise, we define the embedding of a visual, domain specific modeling language in a host spreadsheet system.

Secondly, we present the implementation of this VDSL. To provide a full MDE environment to end users we use data refinement techniques to express the type-safe evolution of a model (after an end-user update) and the automatic co-evolution of the spreadsheet data (that is, the instance) [28]. This novel implementation of the VDSL guarantees the model/instance conformance after the model evolves. Moreover, we also use principles from syntax-based editors [27,30,47] where an initial spreadsheet instance is generated from the model, that has some knowledge about the business logic off the data. Using such knowledge the spreadsheet instance guides end users introducing correct data. In fact, in these generated spreadsheets only updates that conform to the model are allowed.

Finally, we present the results of the empirical study we conducted with the school participants in order to realize whether the use of MDE approach is useful for end users, or not. In the laboratory sessions of this tutorial, we taught participants to use our model-driven spreadsheet environment. Then, the students were asked to perform a set of model-driven spreadsheet tasks, and to write small reports about the advantages/disadvantages of our approach when compared to a regular spreadsheet system.

The remaining of this paper is organized as follows. In Sect. 2 we give a brief overview of the history of spreadsheets. We also present some horror stories that recently had social and financial impact. In Sect. 3 we present data mining and database techniques that are the building blocks of our approach to build models for spreadsheets. Section 4 presents models for defining the business logic of a spreadsheet. First, we present in detail ClassSheet models. After that, we present techniques to automatically infer such a model from (legacy) spreadsheet data. Next, we show the embedding of the ClassSheet models in a widely used spreadsheet system. Section 5 presents a framework for the evolution of model-driven spreadsheets in Haskell. This framework is expressed using data refinements where by defining a model-to-model transformation we get for free the forward and backward transformations that map the data (i.e., the instance). In Sect. 6 we present MDSheet: a MDE environment for spreadsheets. Finally, in Sect. 7 we present the results of the empirical study with the school participants where we validate the use of a MDE approach in spreadsheet development. Section 8 presents the conclusions of the tutorial paper.

2 Spreadsheets: A History of Success?

The use of a tabular-like structure to organize data has been used for many years. A good example of structuring data in this way is the Plimpton 322 tablet (Fig. 1), dated from around 1800 BC [65]. The Plimpton 322 tablet is an example of a table containing four columns and fifteen rows with numerical data. For each column there is a descriptive header, and the fourth column contains a numbering of the rows from one to fifteen, written in the Babylonian number system. This tablet contains Pythagorean triples [14], but was more likely built as a list of regular reciprocal pairs [65].

A tabular layout allows a systematic analysis of the information displayed and it helps to structure values in order to perform calculations.

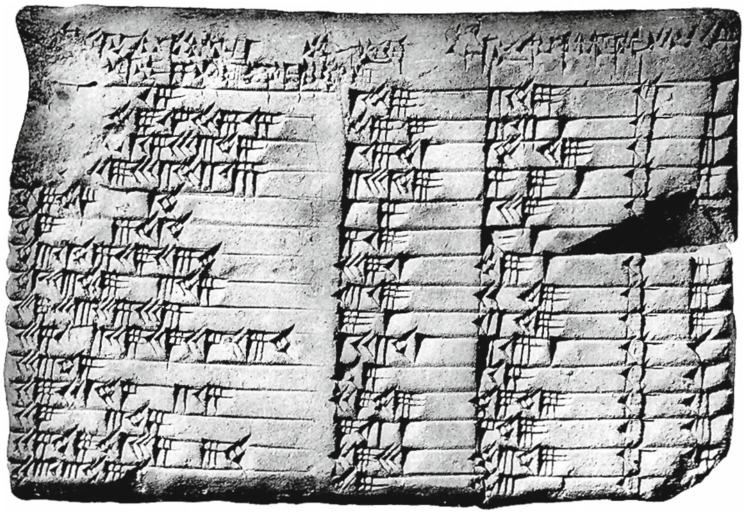
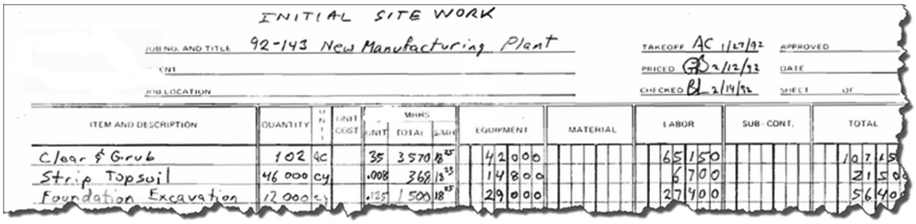


Fig. 1. Plimpton 322 – a tablet from around 1800 BC (A good explanation of the Plimpton 322 tablet is available at Bill Casselman's webpage http://www.math.ubc.ca/~cass/courses/m446-03/pl322/pl322.html).

The terms *spreadsheet* and *worksheet* originated in accounting even before electronic spreadsheets existed. Both had the same meaning, but the term worksheet was mostly used until 1970 [16]. Accountants used a spreadsheet or worksheet to prepare their budgets and other tasks. They would use a pencil and paper with columns and rows. They would place the accounts in one column, the corresponding amount in the next column, etc. Then they would manually total the columns and rows, as in the example shown in Fig. 2. After 1970 the term spreadsheet became more widely used [16].

This worked fine, except when the accountant needed to make a change to one of the numbers. This change would result in having to recalculate, by hand, several totals!

The benefits make (paper) tables applicable to a great variety of domains, like for example on student inquiries or exams, taxes submission, gathering and



INITIAL SITE WORK

JOB NO. AND TITLE 92-143 New Manufacturing Plant

CNT

JOB LOCATION

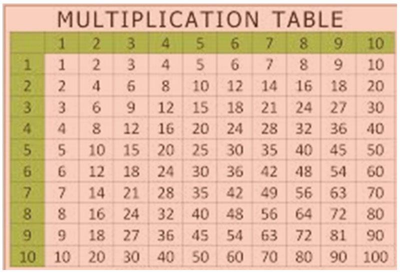
TAKEOFF AC 1/27/92 APPROVED

PRICED [illegible] 2/12/92 DATE

CHECKED [illegible] 2/14/92 SHEET OF

ITEM AND DESCRIPTION	QUANTITY	UNIT	UNIT COST	MHRS UNIT	MHRS TOTAL	MHRS $/MH	EQUIPMENT	MATERIAL	LABOR	SUB-CONT.	TOTAL
Clear & Grub	102	ac		35	3570	18^{25}	42000		65150		1071
Strip Topsoil	46 000	cy		.008	368	18^{25}	14800		6700		215
Foundation Excavation	12 000	cy		.125	1500	18^{25}	29000		27400		5640

Fig. 2. A hand-written budget spreadsheet.



MULTIPLICATION TABLE										
	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	16	20	24	28	32	36	40
5	5	10	15	20	25	30	35	40	45	50
6	6	12	18	24	30	36	42	48	54	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	48	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100

Fig. 3. Paper spreadsheet for a multiplication table.

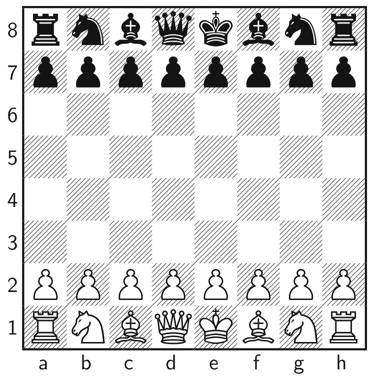


Fig. 4. Chess boards have a tabular layout, with letters identifying columns and numbers identifying rows.

analysis of sport statistics, or any purpose that requires input of data and/or performing calculations. An example of such a table used by students is the multiplication table as displayed in Fig. 3.

This spreadsheet has eleven columns and eleven rows, where the first row and column work as a header to identify the information, and the actual results of the multiplication table are shown in the other cells of the table.

Tabular layouts are also common in games. The chess game is a good example of a tabular layout game as displayed in Fig. 4.

Electronic Spreadsheets. While spreadsheets were very used on paper, they were not used electronically due to the lack of software solutions. During the 1960s and 1970s most financial software bundles were developed to run on mainframe computers and time-sharing systems. Two of the main problems of these software solutions were that they were extremely expensive and required a technical expertise to operate [16]. All that changed in 1979 when VisiCal was released for the Apple II system [13]. 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. VisiCal was the first spreadsheet software to include a textual interface composed by cells and established how the graphical interface of every other spreadsheet software that came after it would be like. It consisted of a column/row tabulation program with an WYSIWYG interface, providing cell references (format `A1`, `A3..A6`). 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 [16]. VisiCal not only made spreadsheets available to a wider audience, but also led to make personal computers more popular by introducing them to the financial and business communities and others.

In 1984, Lotus 1-2-3 was released for MS-DOS with major improvements, which included graphics generation, better performance, and user friendly interface, which led it to dethrone VisiCal as the number one spreadsheet system. It was only in 1990, when Microsoft Windows gained significant market share, that Lotus 1-2-3 lost the position as the most sold spreadsheet software. At that time only Microsoft Excel[1] was compatible with Windows, which raised sales by a huge amount making it the market leading spreadsheet system [16].

In the mid eighties the free software movement started and soon free open source alternatives can be used, namely Gnumeric[2], OpenOffice Calc[3] and derivatives like LibreOffice Calc[4].

More recently, web/cloud-based spreadsheet host systems have been developed, e.g., Google Drive[5], *Microsoft* Office 365[6], and ZoHo Sheet[7] which are making spreadsheets available in different type of mobile devices (from laptops, to tablets and mobile phones!). These systems are not dependent on any particular operating system, allow to create and edit spreadsheets in an online collaborative environment, and provide import/export of spreadsheet files for offline use.

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

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

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

Spreadsheets are a relevant research topic, as they play a pivotal role in modern society. Indeed, they are inherently multi-purpose and widely used both by individuals to cope with simple needs as well as by large companies as integrators of complex systems and as support for business decisions [40]. Also, their popularity is still growing, with an almost impossible to estimate but staggering number of spreadsheets created every year. Spreadsheet popularity is due to characteristics such as their low entry barrier, their availability on almost any computer and their simple visual interface. In fact, being a conventional language that is understood by both professional programmers and end users [53], spreadsheets are many times used as bridges between these two communities which often face communication problems. Ultimately, spreadsheets seem to hit the sweet spot between flexibility and expressiveness.

Spreadsheets have probably passed the point of no return in terms of importance. There are several studies that show the success of spreadsheets:

- it is estimated that 95 % of all U.S. firms use them for financial reporting [60];
- it is also known that 90 % of all analysts in industry perform calculations in spreadsheets [60];
- finally, studies show that 50 % of all spreadsheets are the basis for decisions [40].

This importance, however, has not been achieved together with effective mechanisms for error prevention, as shown by several studies [57,58]. Indeed, spreadsheets are known to be error-prone, a claim that is supported by the long list of real problems that were blamed on spreadsheets, which is compiled, available and frequently updated at the *European Spreadsheet Risk Interest Group (EuSpRIG)* web site[8].

One particularly sad example in this list involves our country (and other European countries), which currently undergoes a financial rescue plan based on intense austerity whose merit was co-justified upon [64]. The authors of that paper present evidence that GDP growth slows to a snail's pace once the sovereign debt of a nation exceeds 90 % of GDP, and it was precisely this evidence that was several times politically used to argue for austerity measures.

Unfortunately, the fact is that the general conclusion of [64] has been publicly questioned given that a formula range error was found in the spreadsheet supporting the authors' calculations. While the authors have later re-affirmed their original conclusions, the public pressure was so intense that a few weeks later they felt the need to publish an errata of their 2010 paper. It is furthermore unlikely that the concrete social and economical impacts of that particular spreadsheet error will ever be determined.

Given the massive use of spreadsheets and the their alarming number of errors, many researcher have been working on this topic. Burnett et al. studied the use of end-users programming principles to spreadsheets [15,36,66,67], as well as the use of software engineering techniques [35,37]. Erwig et al. applied

[8] This list of horror stories is available at: http://www.eusprig.org/horror-stories.htm.

several techniques from software engineering to spreadsheets, such as testing and debugging [3,4,7,8], model-driven approaches [5,9,29,32,33,50]. Erwig also studied the use in spreadsheets of programming languages techniques such as type systems [1,2,6,34]. Hermans et al. studied how to help users better understand the spreadsheets they use [40–42]. In this context, Cunha et al. proposed a catalog of smells for spreadsheets [21] and a tool to detect them [22]. Panko et al. have been developing very interesting work to understand the errors found in spreadsheets [56–59].

3 Spreadsheet Analysis

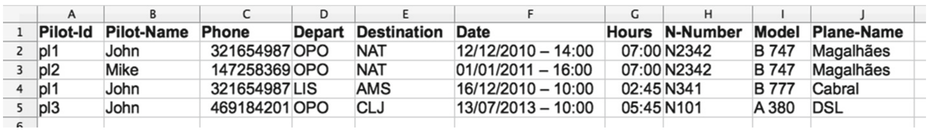
Spreadsheets, like other software systems, usually start as simple, single user software systems and rapidly evolve into complex and large systems developed by several users [40]. Such spreadsheets become hard to maintain and evolve because of the large amount of data they contain, the complex dependencies and formulas used (that very often are poor documented [41]), and finally because the developers of those spreadsheets may not be available (because they may have left the company/project). In these cases to understand the business logic defined in such legacy spreadsheets is a hard and time consuming task [40].

In this section we study techniques to analyze spreadsheet data using technology from both the data mining and the database realms. This technology is used to mine the spreadsheet data in order to automatically compute a model describing the business logic of the underlying spreadsheet data.

3.1 Spreadsheet Data Mining

Before we present these techniques, let us consider the example spreadsheet modeling an airline scheduling system which we adapted from [51] and illustrated in Fig. 5.

The labels in the first row have the following meaning: **PilotId** represents a unique identification code for each pilot, **Pilot-Name** is the name of the pilot, and column labeled **Phone** contains his phone number. Columns labeled **Depart** and **Destination** contain the departure and destination airports, respectively. The column **Date** contains the date of the flight and **Hours** defines the number of hours of the flight. Next columns define the plain used in the flight: **N-Number** is a unique number of the plain, **Model** is the model of the plane, and **Plane-Name** is the name of the plane.



	A	B	C	D	E	F	G	H	I	J
1	Pilot-Id	Pilot-Name	Phone	Depart	Destination	Date	Hours	N-Number	Model	Plane-Name
2	pl1	John	321654987	OPO	NAT	12/12/2010 – 14:00	07:00	N2342	B 747	Magalhães
3	pl2	Mike	147258369	OPO	NAT	01/01/2011 – 16:00	07:00	N2342	B 747	Magalhães
4	pl1	John	321654987	LIS	AMS	16/12/2010 – 10:00	02:45	N341	B 777	Cabral
5	pl3	John	469184201	OPO	CLJ	13/07/2013 – 10:00	05:45	N101	A 380	DSL
6										

Fig. 5. A spreadsheet representing pilots, planes and flights.

This spreadsheet defines a valid model to represent the information for scheduling flights. However, it contains *redundant information*. For example, the displayed data specifies the name of the plane *Magalhães* twice. This kind of redundancy makes the maintenance and update of the spreadsheet complex and error-prone. In fact, two well-known database problems occur when organizing our data in a non-normalized way [71]:

– Update Anomalies: this problem occurs when we change information in one tuple but leave the same information unchanged in the others. In our example, this may happen if we change the name of the plane *Magalhães* on row 2, but not on row 3. As a result the data will become inconsistent!
– Deletion Anomalies: problem happens when we delete some tuple and we lose other information as a side effect. For example, if we delete row 3 in the spreadsheet all the information concerning the pilot *Mike* is eliminated.

As a result, a mistake is easily made, for example, by mistyping a name and thus corrupting the data. The same information can be stored without redundancy. In fact, in the database community, techniques for database normalization are commonly used to minimize duplication of information and improve data integrity. Database normalization is based on the detection and exploitation of *functional dependencies* inherent in the data [51,72].

Exercise 1. *Consider the data in the following table and answer the questions.*

movieID	*title*	*language*	*renterNr*	*renterNm*	*rentStart*	*rentFinished*	*rent*	*totalToPay*
mv23	*Little Man*	*English*	*c33*	*Paul*	*01-04-2010*	*26-04-2010*	*0.5*	*12.50*
mv1	*The Ohio*	*English*	*c33*	*Paul*	*30-03-2010*	*23-04-2010*	*0.5*	*12.00*
mv21	*Edmond*	*English*	*c26*	*Smith*	*02-04-2010*	*04-04-2010*	*0.5*	*1.00*
mv102	*You, Me, D*	*English*	*c3*	*Michael*	*22-03-2010*	*03-04-2010*	*0.3*	*3.60*
mv23	*Little Man*	*English*	*c26*	*Smith*	*02-12-2009*	*04-04-2010*	*0.5*	*61.50*
mv23	*Little Man*	*English*	*c14*	*John*	*12-04-2010*	*16-04-2010*	*0.5*	*2.00*

1. *Which row(s) can be deleted without causing a deletion anomaly?*
2. *Identify two attributes that can cause update anomalies when editing the corresponding data.*

3.2 Databases Technology

In order to infer a model representing the business logic of a spreadsheet data, we need to analyze the data and define relationships between data entities. Objects that are contained in a spreadsheet and the relationships between them are reflected by the presence of *functional dependencies* between spreadsheet columns. Informally, a functional dependency between a column C and another column C' means that the values in column C determine the values in column

C', that is, there are no two rows in the spreadsheet that have the same value in column C but differ in their values in column C'.

For instance, in our running example the functional dependency between column `A` (**Pilot-Id**) and column `B` (**Pilot-Name**) exists, meaning that the identification number of a pilot determines its name. That is to say that, there are no two rows with the same id number (column `A`), but differ in their names (column `B`). A similar functional dependency occurs between identifier (i.e., number) of a plane **N-Number** and its name **Plane-Name**.

This idea can be extended to multiple columns, that is, when any two rows that agree in the values of columns $C_1, \ldots, C_n$ also agree in their value in columns $C'_1, \ldots, C'_m$, then $C'_1, \ldots, C'_m$ are said to be *functionally dependent* on $C_1, \ldots, C_n$.

In our running example, the following functional dependencies hold:

$$\mathit{Depart}, \mathit{Destination} \rightharpoonup \mathit{Hours}$$

stating that the departure and destination airports determines the number of hours of the flight.

Definition 1. *A functional dependency between two sets of attributes A and B, written $A \rightharpoonup B$, holds in a table if for any two tuples t and t' in that table $t[A] = t'[A] \implies t[B] = t'[B]$ where $t[A]$ yields the (sub)tuple of values for the attributes in A. In other words, if the tuples agree in the values for attribute set A, they agree in the values for attribute set B. The attribute set A is also called antecedent, and the attribute set B consequent.*

Our goal is to use the data in a spreadsheet to identify functional dependencies. Although we use all the data available in the spreadsheet, we consider a particular instance of the spreadsheet domain only. However, there may exist counter examples to the dependencies found, but these just happen not to be included in the spreadsheet. Thus, the dependencies we discover are always an approximation. On the other hand, depending on the data, it can happen that many "accidental" functional dependencies are detected, that is, functional dependencies that do not reflect the underlying model.

For instance, in our example we can identify the following dependency that just happens to be fulfilled for this particular data set, but that does certainly *not* reflect a constraint that should hold in general: $\mathit{Model} \rightharpoonup \mathit{Plane\_Name}$, that is to say that the model of a plane determines its name! In fact, the data contained in the spreadsheet example supports over 30 functional dependencies. Next we list a few more that hold for our example.

$\mathit{Pilot\text{-}ID} \rightharpoonup \mathit{Pilot\text{-}Name}$
$\mathit{Pilot\text{-}ID} \rightharpoonup \mathit{Phone}$
$\mathit{Pilot\text{-}ID} \rightharpoonup \mathit{Pilot\text{-}Name}, \mathit{Phone}$
$\mathit{Depart}, \mathit{Destination} \rightharpoonup \mathit{Hours}$
$\mathit{Hours} \rightharpoonup \mathit{Model}$

Exercise 2. *Consider the data in the following table.*

proj1	*John*	*New York*	*30-03-2010*	*50000*	*Long Island*	*Richy*	*34*	*USA*	*Mike*	*inst3*	*36*	*6*
proj1	*John*	*New York*	*30-03-2010*	*50000*	*Long Island*	*Tim*	*33*	*JP*	*Anthony*	*inst1*	*24*	*4*
proj1	*John*	*New York*	*30-03-2010*	*50000*	*Long Island*	*Mark*	*30*	*UK*	*Alfred*	*inst3*	*36*	*6*
proj2	*John*	*Los Angels*	*02-04-2010*	*3000*	*Los Angels*	*Richy*	*34*	*USA*	*Mike*	*inst2*	*30*	*5*
proj3	*Paul*	*Chicago*	*01-01-2009*	*12000*	*Chicago*	*Tim*	*33*	*JP*	*Anthony*	*inst1*	*24*	*4*
proj3	*Paul*	*Chicago*	*01-01-2009*	*12000*	*Chicago*	*Mark*	*30*	*UK*	*Alfred*	*inst1*	*24*	*4*

Which are the functional dependencies that hold in this case?

Because spreadsheet data may induce too many functional dependencies, the next step is therefore to filter out as many of the accidental dependencies as possible and keep the ones that are indicative of the underlying model. The process of identifying the "valid" functional dependencies is, of course, ambiguous in general. Therefore, we employ a series of heuristics for evaluating dependencies.

Note that several of these heuristics are possible only in the context of spreadsheets. This observation supports the contention that end-user software engineering can benefit greatly from the context information that is available in a specific end-user programming domain. In the spreadsheet domain rich context is provided, in particular, through the spatial arrangement of cells and through labels [31].

Next, we describe five heuristics we use to discard accidental functional dependencies. Each of these heuristics can add support to a functional dependency.

Label semantics. This heuristic is used to classify antecedents in functional dependencies. Most antecedents (recall that antecedents determine the values of consequents) are labeled as "code", "id", "nr", "no", "number", or are a combination of these labels with a label more related to the subject. functional dependency with an antecedent of this kind receives high support.

For example, in our property renting spreadsheet, we give high support to the functional dependency *N-Number* $\rightharpoonup$ *Plane-Name* than to the *Plane-Name* $\rightharpoonup$ *N-Number* one.

Label arrangement. If the functional dependency respects the original order of the attributes, this counts in favor of this dependency since very often key attributes appear to the left of non-key attributes.

In our running example, there are two functional dependencies induced by columns *N-Number* and *Plane-Name*, namely *N-Number* $\rightharpoonup$ *Plane-Name* and *Plane-Name* $\rightharpoonup$ *N-Number*. Using this heuristic we prefer the former dependency to the latter.

Antecedent size. Good primary keys often consist of a small number of attributes, that is, they are based on small antecedent sets. Therefore, the smaller the number of antecedent attributes, the higher the support for the functional dependency.

Ratio between antecedent and consequent sizes. In general, functional dependencies with smaller antecedents and larger consequents are stronger and thus more likely to be a reflection of the underlying data model. Therefore, a functional dependency receives the more support, the smaller the ratio of the number of consequent attributes is compared to the number of antecedent attributes.

Single value columns. It sometimes happens that spreadsheets have columns that contain just one and the same value. In our example, the column labeled *country* is like this. Such columns tend to appear in almost every functional dependency's consequent, which causes them to be repeated in many relations. Since in almost all cases, such dependencies are simply a consequence of the limited data (or represent redundant data entries), they are most likely not part of the underlying data model and will thus be ignored.

After having gathered support through these heuristics, we aggregate the support for each functional dependency and sort them from most to least support. We then select functional dependencies from that list in the order of their support until all the attributes of the schema are covered.

Based on these heuristics, our algorithm produces the following dependencies for the flights spreadsheet data:

$$\begin{aligned}&\mathit{Pilot\text{-}ID} \rightharpoonup \mathit{Pilot\text{-}Name}, \mathit{Phone}\\ &\mathit{N\text{-}Number} \rightharpoonup \mathit{Model}, \mathit{Plane\text{-}Name}\\ &\mathit{Pilot\text{-}ID}, \mathit{N\text{-}Number}, \mathit{Depart}, \mathit{Destination}, \mathit{Date}, \mathit{Hours} \rightharpoonup \emptyset\end{aligned}$$

Exercise 3. *Consider the data in the following table and answer the next questions.*

project_nr	*manager*	*location*	*delivery_date*	*budget*	*employee_name*	*age*	*nationality*
proj1	*John*	*New York*	*30-03-2010*	*50000*	*Richy*	*34*	*USA*
proj1	*John*	*New York*	*30-03-2010*	*50000*	*Tim*	*33*	*JP*
proj1	*John*	*New York*	*30-03-2010*	*50000*	*Mark*	*30*	*UK*
proj2	*John*	*Los Angels*	*02-04-2010*	*3000*	*Richy*	*34*	*USA*
proj3	*Paul*	*Chicago*	*01-01-2009*	*12000*	*Tim*	*33*	*JP*
proj3	*Paul*	*Chicago*	*01-01-2009*	*12000*	*Mark*	*30*	*UK*

1. *Which are the functional dependencies that hold in this case?*
2. *Was this exercise easier to complete than Exercise 2? Why do you think this happened?*

Relational Model. Knowledge about the functional dependencies in a spreadsheet provides the basis for identifying tables and their relationships in the data, which form the basis for defining models for spreadsheet. The more accurate we can make this inference step, the better the inferred models will reflect the actual business models.

It is possible to construct a relational model from a set of observed functional dependencies. Such a model consists of a set of relation schemas (each given by a set of column names) and expresses the basic business model present in the spreadsheet. Each relation schema of such a model basically results from grouping functional dependencies together.

For example, for the spreadsheet in Fig. 5 we could infer the following relational model (underlined column names indicate those columns on which the other columns are functionally dependent).

$\mathit{Pilots}\ (\underline{\mathit{Pilot\text{-}Id}}, \mathit{Pilot\text{-}Name}, \mathit{Phone})$
$\mathit{Planes}\ (\underline{\mathit{N\text{-}Number}}, \mathit{Model}, \mathit{Plane\text{-}Name}$
$\mathit{Flights}\ (\underline{\mathit{Pilot\text{-}ID}, \mathit{N\text{-}Number}, \mathit{Depart}, \mathit{Destination}, \mathit{Date}, \mathit{Hours})}$

The model has three relations: *Pilots* stores information about pilots; *Planes* contains all the information about planes, and *Flights* stores the information on flights, that is, for a particular pilot, a specific number of a plane, it stores the depart and destination airports and the data ans number of hours of the flights.

Note that several models could be created to represent this system. We have shown that the models our tool automatically generates are comparable in quality to the ones designed by database experts [19].

Although a relational model is very expressive, it is not quite suitable for spreadsheets since spreadsheets need to have a layout specification.

In contrast, the *ClassSheet* modeling framework offers high-level, object-oriented formal models to specify spreadsheets and thus present a promising alternative [29].

ClassSheets allow users to express business object structures within a spreadsheet using concepts from the Unified Modeling Language (UML). A spreadsheet application consistent with the model can be automatically generated, and thus a large variety of errors can be prevented.

We therefore employ ClassSheet as the underlying modeling approach for spreadsheets and transform the inferred relational model into a ClassSheet model.

Exercise 4. *Use the HaExcel libraries to infer the functional dependencies from the data given in Exercise 3.*[9] *For the functional dependencies computed, create the corresponding relational schema.*

4 Model-Driven Spreadsheet Engineering

The use of abstract models to reason about concrete artifacts has successfully and widespreadly been employed in science and in engineering. In fact, there are many fields for which model-driven engineering is the default, uncontested approach to follow: it is a reasonable assumption that, excluding financial or cultural limitations, no private house, let alone a bridge or a skyscraper, should be built before a model for it has been created and has been thoroughly analyzed and evolved.

[9] HaExcel can be found at http://ssaapp.di.uminho.pt.

Being itself a considerably more recent scientific field, not many decades have passed since software engineering has seriously considered the use of models. In this section, we study model-driven approaches to spreadsheet software engineering.

4.1 Spreadsheet Models

In an attempt to overcome the issue of spreadsheet errors using model-driven approaches, several techniques have been proposed, namely the creation of spreadsheet templates [9], the definition of ClassSheet [29] models and the use of class diagrams to specify spreadsheets [39]. These proposals guarantee that users may safely perform particular editing steps on their spreadsheets and they introduce a form of model-driven software development: a spreadsheet business model is defined from which a customized spreadsheet application is generated guaranteeing the consistency of the spreadsheet with the underlying model.

Despite of its huge benefits, model-driven software development is sometimes difficult to realize in practice. In the context of spreadsheets, for example, the use of model-driven software development requires that the developer is familiar both with the spreadsheet domain (business logic) and with model-driven software development. In the particular case of the use of templates, a new tool is necessary to be learned, namely ViTSL [9]. By using this tool, it is possible to generate a new spreadsheet respecting the corresponding model. This approach, however, has several drawbacks: first, in order to define a model, spreadsheet model developers will have to become familiar with a new programming environment. Second, and most important, there is no connection between the stand alone model development environment and the spreadsheet system. As a result, it is not possible to (automatically) synchronize the model and the spreadsheet data, that is, the co-evolution of the model and its instance is not possible.

The first contribution of our work is the embedding of ClassSheet spreadsheet models in spreadsheets themselves. Our approach closes the gap between creating and using a domain specific language for spreadsheet models and a totally different framework for actually editing spreadsheet data. Instead, we unify these operations within spreadsheets: in one worksheet we define the underlying model while another worksheet holds the actual data, such that the model and the data are kept synchronized by our framework. A summarized description of this work has been presented in [23,26], a description that we revise and extend in this paper, in Sect. 4.5.

ClassSheet Models. ClassSheets are a high-level, object-oriented formalism to specify the business logic of spreadsheets [29]. This formalism allows users to express business object structures within a spreadsheet using concepts from the UML [69].

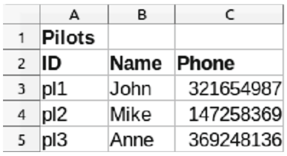
ClassSheets define (work)sheets (s) containing classes (c) formed by blocks (b). Both sheets and classes can be expandable, i.e., their instances can be repeated either horizontally ($c^{\rightarrow}$) or vertically ($b^{\downarrow}$). Classes are identified by labels (l). A block can represent in its basic form a spreadsheet cell, or it can be a composition of other blocks. When representing a cell, a block can contain a basic value (φ, e.g., a string or an integer) or an attribute ($a = f$), which is composed by an attribute name (a) and a value (f). Attributes can define three types of cells: '(1), an input value, where a default value gives that indication, *(2)*, a named reference to another attribute ($n.a$, where n is the name of the class and a the name of the attribute) or *(3)*, an expression built by applying functions to a varying number of arguments given by a formula ($\varphi(f, \ldots, f)$).

ClassSheets can be represented textually, according to the grammar presented in Fig. 6 and taken directly from [29], or visually as described further below.

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

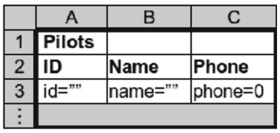
Fig. 6. Syntax of the textual representation of ClassSheets.

Vertically Expandable Tables. In order to illustrate how ClassSheets can be used in practice we shall consider the example spreadsheet defining a airline scheduling system as introduced in Sect. 3. In Fig. 7a we present a spreadsheet containing the pilot's information only. This table has a title, **Pilots**, and a row with labels, one for each of the table's column: **ID** represents a unique pilot identifier, **Name** represents the pilot's name and **Phone** represents the pilot's phone contact. Each of the subsequent rows represents a concrete pilot.



	A	B	C
1	Pilots		
2	ID	Name	Phone
3	pl1	John	321654987
4	pl2	Mike	147258369
5	pl3	Anne	369248136

(a) Pilots' table.



	A	B	C
1	Pilots		
2	ID	Name	Phone
3	id=""	name=""	phone=0
⋮			

(b) Pilots' visual ClassSheet model.

$$\begin{array}{llll}
\underline{\textbf{Pilots}} : \text{Pilots} & \mid \sqcup & \mid \sqcup & \hat{\ } \\
\underline{\textbf{Pilots}} : \text{ID} & \mid \text{Name} & \mid \text{Phone} & \hat{\ } \\
\underline{\textbf{Pilots}} : (\text{id}=\texttt{""} & \mid \text{name}=\texttt{""} & \mid \text{phone}=0)^{\downarrow} &
\end{array}$$

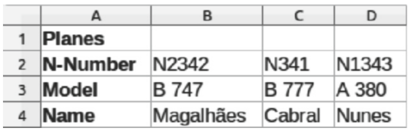
(c) Pilots' textual ClassSheet model.

Fig. 7. Pilots' example.

Tables such as the one presented in Fig. 7a are frequently used within spreadsheets, and it is fairly simple to create a model specifying them. In fact, Fig. 7b represents a visual ClassSheet model for this pilot's table, whilst Fig. 7c shows the textual ClassSheet representation. In the next paragraphs we explain such a model. To model the labels we use a textual representation and the exact same names as in the data sheet (**Pilots**, **ID**, **Name** and **Phone**). To model the actual data we abstract concrete column cell values by using a single identifier: we use the one-worded, lower-case equivalent of the corresponding column label (*id*, *name*, and *phone*). Next, a default value is associated with each column: columns `A` and `B` hold strings (denoted in the model by the empty string "" following the = sign), and column `C` holds integer values (denoted by 0 following =). Note that the last row of the model is labeled on the left hand-side with vertical ellipses. This means that it is possible for the previous block of rows to expand vertically, that is, the tables that conform to this model can have as many rows/pilots as needed. The scope of the expansion is between the ellipsis and the black line (between labels 2 and 3). Note that, by definition, ClassSheets do not allow for nested expansion blocks, and thus, there is no possible ambiguity associated with this feature. The instance shown in Fig. 7a has three pilots.

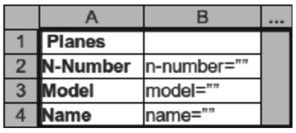
Horizontally Expandable Tables. In the lines of what we described in the previous section, airline companies must also store information on their airplanes. This is the purpose of table **Planes** in the spreadsheet illustrated in Fig. 8a, which is organized as follows: the first column holds labels that identify each row, namely, **Planes** (labeling the table itself), **N-Number**, **Model** and **Name**; cells in row **N-Number** (respectively **Model** and **Name**) contain the unique n-number identifier of a plane, (respectively the model of the plane and the name of the plane). Each of the subsequent columns contains information about one particular aircraft.

The **Planes** table can be visually modeled by the illustration in Fig. 8b and textually by the definition in Fig. 8c. This model may be constructed following the same strategy as in the previous section, but now swapping columns and



	A	B	C	D
1	Planes			
2	N-Number	N2342	N341	N1343
3	Model	B 747	B 777	A 380
4	Name	Magalhães	Cabral	Nunes

(a) Planes' table.



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

(b) Planes' visual ClassSheet model.

$$\left(\begin{array}{lll} |\mathbf{Planes}: & \text{Planes} & \hat{} \\ \underline{\mathbf{N\text{-}Number}}: & \text{N-Number} & \hat{} \\ \underline{\mathbf{Model}}: & \text{Model} & \hat{} \\ \underline{\mathbf{Name}}: & \text{Name} & \end{array}\right) \mid \left(\begin{array}{lll} |\mathbf{Planes}: & \sqcup & \hat{} \\ \underline{\mathbf{N\text{-}Number}} & \text{n-number}= \texttt{""} & \hat{} \\ \underline{\mathbf{Model}}: & \text{model}= \texttt{""} & \hat{} \\ \underline{\mathbf{Name}}: & \text{name}= \texttt{""} & \end{array}\right) \rightarrow$$

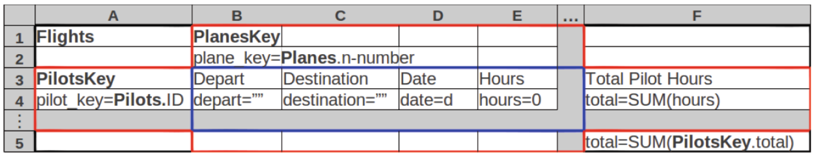
(c) Planes' textual ClassSheet model.

Fig. 8. Planes' example.

rows: the first column contains the label information and the second one the names abstracting concrete data values: again, each cell has a name and the default value of the elements in that row (in this example, all the cells have as default values empty strings); the third column is labeled not as `C` but with ellipses meaning that the immediately previous column is horizontally expandable. Note that the instance table has information about three planes.

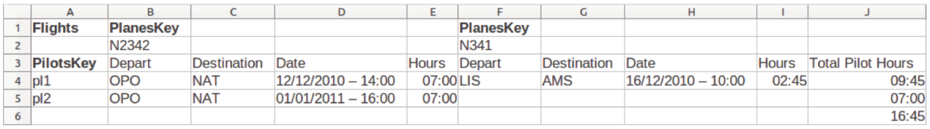
Relationship Tables. The examples used so far (the tables for pilots and planes) are useful to store the data, but another kind of table exists and can be used to relate information, being of more practical interest.

Having pilots and planes, we can set up a new table to store information from the flights that the pilots make with the planes. This new table is called a *relationship* table since it relates two entities, which are the pilots and the planes. A possible model for this example is presented in Fig. 9, which also depicts an instance of that model.



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

(a) Flights' visual ClassSheet model.



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

(b) Flights' table.

Fig. 9. Flights' table, relating pilots and planes.

The flights' table contains information from distinct entities. In the model (Fig. 9a), there is the class **Flights** that contains all the information, including:

– information about planes (class **PlanesKey**, columns `B` to `E`), namely a reference to the planes table (cell `B2`);
– information about pilots (class **PilotsKey**, rows `3` and `4`), namely a reference to the pilots table (cell `A4`);
– information about the flights (in the range `B3:E4`), namely the depart location (cell `B4`), the destination (cell `C4`), the time of departure (cell `D4`) and the duration of the flight (cell `E4`);
– the total hours flown by each pilot (cell `F4`), and also a grand total (cell `F5`). We assume that the same pilot does not appear in two different rows. In fact, we could use ClassSheet extensions to ensure this [23, 25].

For the first flight stored in the data (Fig. 9b), we know that the pilot has the identifier *pl1*, the plane has the n-number *N2342*, it departed from *OPO* in direction to *NAT* at 14:00 on December 12, 2010, with a duration of 7 h.

Note that we do not show the textual representation of this part of the model because of its complexity and because it would not improve the understandability of this document.

Exercise 5. *Consider we would like to construct a spreadsheet to handle a school budget. This budget should consider different categories of expenses such as personnel, books, maintenance, etc. These different items should be laid along the rows of the spreadsheet. The budget must also consider the expenses for different years. Each year must have information about the number of items bought, the price per unit, and the total amount of money spent. Each year should be created after the previous one in an horizontal displacement.*

1. *Define a standard spreadsheet that contains data at least for two years and several expenses.*
2. *Define now a ClassSheet defining the business logic of the school budget. Please note that the spreadsheet data defined in the previous item should be an instance of this model.*

Exercise 6. *Consider the spreadsheets given in all previous exercises. Define a ClassSheet that implements the business logic of the spreadsheet data.*

4.2 Inferring Spreadsheet Models

In this section we explain in detail the steps to automatically extract a ClassSheet model from a spreadsheet [19]. Essentially, our method involves the following steps:

1. Detect all functional dependencies and identify model-relevant functional dependencies;
2. Determine relational schemas with candidate, foreign, and primary keys;
3. Generate and refactor a relational graph;
4. Translate the relational graph into a ClassSheet.

We have already introduced steps 1 and 2 in Sect. 3. In the following subsections we will explain the steps 3 and 4.

The Relational Intermediate Directed Graph. In this sub-section we explain how to produce a *Relational Intermediate Directed (RID) Graph* [11]. This graph includes all the relationships between a given set of schemas. Nodes in the graph represent schemas and directed edges represent foreign keys between those schemas. For each schema, a node in the graph is created, and for each foreign key, an edge with cardinality “*” at both ends is added to the graph.

Figure 10 represents the RID graph for the flights scheduling. This graph can generally be improved in several ways. For example, the information about foreign

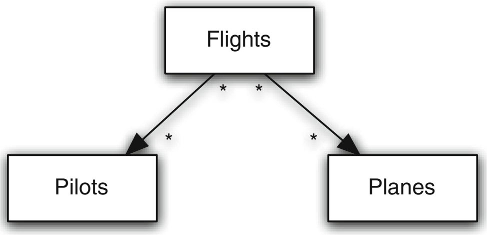


Fig. 10. RID graph for our running example.

keys may lead to additional links in the RID graph. If two relations reference each other, their relationship is said to be *symmetric* [11]. One of the foreign keys can then be removed. In our example there are no symmetric references.

Another improvement to the RID graph is the detection of relationships, that is, whether a schema is a relationship connecting other schemas. In such cases, the schema is transformed into a relationship. The details of this algorithm are not so important and left out for brevity.

Since the only candidate key of the schema *Flights* is the combination of all the other schemas' primary keys, it is a relationship between all the other schemas and is therefore transformed into a relationship. The improved RID graph can be seen in Fig. 11.

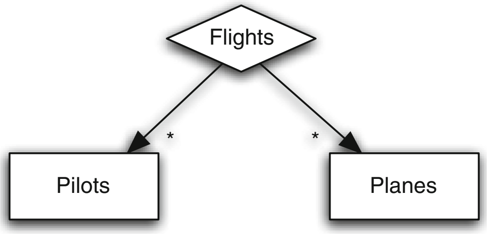


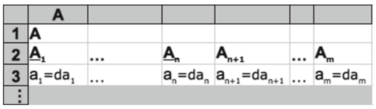
Fig. 11. Refactored RID graph.

Generating ClassSheets. The RID graph generated in Sect. 4.2 can be directly translated into a ClassSheet diagram. By default, each node is translated into a class with the same name as the relation and a vertically expanding block. In general, for a relation of the form

$$\underline{A_1}, \ldots, \underline{A_n}, A_{n+1}, \ldots, A_m$$

and default values $da_1, \ldots, da_n, d_{n+1}, \ldots, d_m$, a ClassSheet class/table is generated as shown in Fig. 12[10]. From now on this rule is termed *rule 1*.

[10] We omit here the column labels, whose names depend on the number of columns in the generated table.



	A					
1	A					
2	$\underline{A_1}$	...	$\underline{A_n}$	A_{n+1}	...	A_m
3	a_1=da_1	...	a_n=da_n	a_{n+1}=da_{n+1}	...	a_m=da_m
⋮						

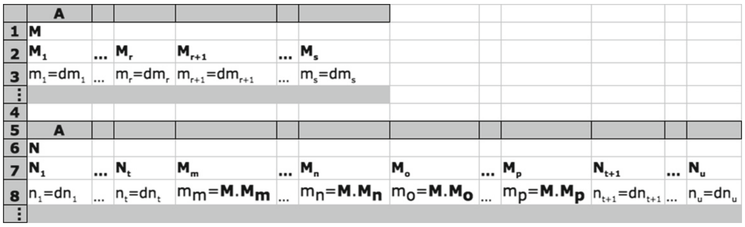
Fig. 12. Generated class for a relation A.

This ClassSheet represents a spreadsheet "table" with name **A**. For each attribute, a column is created and is labeled with the attribute's name. The default values depend on the attribute's domain. This table expands vertically, as indicated by the ellipses. The key attributes become underlined labels.

A special case occurs when there is a foreign key from one relation to another. The two relations are created basically as described above but the attributes that compose the foreign key do not have default values, but references to the corresponding attributes in the other class. Let us use the following generic relations:

$$M(\underline{M_1, \ldots, M_r}, M_{r+1}, \ldots, M_s)$$
$$N(\underline{N_1, \ldots, N_t, M_m, \ldots, M_n}, M_o, \ldots, M_p, N_{t+1}, \ldots, N_u)$$

Note that $M_n, \ldots, M_m, M_o, \ldots, M_p$ are foreign keys from the relation N to the relation M, where $1 \leqslant n, m, o, p \leqslant r$, $n \leqslant m$, and $o \leqslant p$. This means that the foreign key attributes in N can only reference key attributes in the M. The corresponding ClassSheet is illustrated in Fig. 13. This rule is termed *rule 2*.



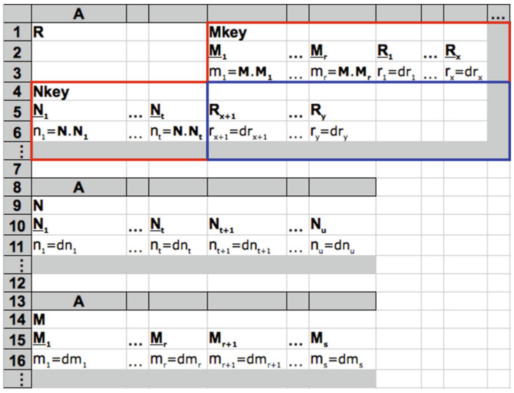
	A											
1	M											
2	M_1	...	M_r	M_{r+1}	...	M_s						
3	m_1=dm_1	...	m_r=dm_r	m_{r+1}=dm_{r+1}	...	m_s=dm_s						
⋮												
4												
5	A											
6	N											
7	N_1	...	N_t	M_m	...	M_n	M_o	...	M_p	N_{t+1}	...	N_u
8	n_1=dn_1	...	n_t=dn_t	m_m=**M.M$_m$**	...	m_n=**M.M$_n$**	m_o=**M.M$_o$**	...	m_p=**M.M$_p$**	n_{t+1}=dn_{t+1}	...	n_u=dn_u
⋮												

Fig. 13. Generated ClassSheet for relations with foreign keys.

Relationships are treated differently and will be translated into cell classes. We distinguish between two cases: *(A)* relationships between two schemas, and *(B)* relationships between more than two schemas.

For case *(A)*, let us consider the following set of schemas:

$$M(\underline{M_1, \ldots, M_r}, M_{r+1}, \ldots, M_s)$$
$$N(\underline{N_1, \ldots, N_t}, N_{t+1}, \ldots, N_u)$$
$$R(\underline{M_1, \ldots, M_r, N_1, \ldots, N_t, R_1, \ldots, R_x}, R_{x+1}, \ldots, R_y)$$



	A									...
1	R			Mkey						
2				M_1	...	M_r	R_1	...	R_x	
3				m_1=M.M_1	...	m_r=M.M_r	r_1=dr_1	...	r_x=dr_x	
4	Nkey									
5	N_1	...	N_t	R_{x+1}	...	R_y				
6	n_1=N.N_1	...	n_t=N.N_t	r_{x+1}=dr_{x+1}	...	r_y=dr_y				
⋮										
7										
8	A									
9	N									
10	N_1	...	N_t	N_{t+1}	...	N_u				
11	n_1=dn_1	...	n_t=dn_t	n_{t+1}=dn_{t+1}	...	n_u=dn_u				
⋮										
12										
13	A									
14	M									
15	M_1	...	M_r	M_{r+1}	...	M_s				
16	m_1=dm_1	...	m_r=dm_r	m_{r+1}=dm_{r+1}	...	m_s=dm_s				
⋮										

Fig. 14. ClassSheet of a relationship connecting two relations.

The ClassSheet that is produced by this translation is shown in Fig. 14 and explained next.

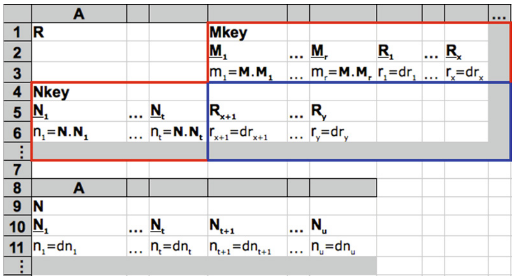
For both nodes M and N a class is created as explained before (lower part of the ClassSheet). The top part of the ClassSheet is divided in two classes and one cell class. The first class, **NKey**, is created using the key attributes from the **N** class. All its values are references to **N**. For example, $n1 = \mathbf{N.N1}$ references the values in column `A` in class **N**. This makes the spreadsheet easier to maintain while avoiding insertion, modification and deletion anomalies [17]. Class **Mkey** is created using the key attributes of the class **M** and the rest of the key attributes of the relationship R. The cell class (with blue border) is created using the rest of the attributes of the relationship R.

In principle, the positions of **M** and **N** are interchangeable and we have to choose which one expands vertically and which one expands horizontally. We choose whichever combination minimizes the number of empty cells created by the cell class, that is, the number of key attributes from **M** and **R** should be similar to the number of non-key attributes of **R**. This rule is named *rule A*. Three special cases can occur with this configuration.

Case 1. The first case occurs when one of the relations **M** or **N** might have only key attributes. Let us assume that **M** is in this situation:

$$M(\underline{M_1, \ldots, M_r})$$
$$N(\underline{N_1, \ldots, N_t}, N_{t+1}, \ldots, N_u)$$
$$R(\underline{M_1, \ldots, M_r, N_1, \ldots, N_t, R_1, \ldots, R_x}, R_{x+1}, \ldots, R_y)$$

In this case, and since all the attributes of that class are already included in the class **MKey** or **NKey**, no separated class is created for it. The resultant ClassSheet would be similar to the one presented in Fig. 14, but a separated class would not be created for **M** or for **N** or for both. Figure 15 illustrates this situation. This rule is from now on termed *rule A1*.



	A									...
1	R			Mkey						
2				$\underline{M_1}$	...	$\underline{M_r}$	$\underline{R_1}$	...	$\underline{R_x}$	
3				m_1=**M.M**$_1$	...	m_r=**M.M**$_r$	r_1=dr_1	...	r_x=dr_x	
4	Nkey									
5	$\underline{N_1}$	...	$\underline{N_t}$	R_{x+1}	...	R_y				
6	n_1=**N.N**$_1$	...	n_t=**N.N**$_t$	r_{x+1}=dr_{x+1}	...	r_y=dr_y				
⋮										
7										
8	A									
9	N									
10	$\underline{N_1}$	...	$\underline{N_t}$	N_{t+1}	...	N_u				
11	n_1=dn_1	...	n_t=dn_t	n_{t+1}=dn_{t+1}	...	n_u=dn_u				
⋮										

Fig. 15. ClassSheet where one entity has only key attributes.

Case 2. The second case occurs when the key of the relationship R is only composed by the keys of M and N (defined as before), that is, R is defined as follows:

$$\begin{array}{l} M(\underline{M_1, \ldots, M_r}, M_{r+1}, \ldots, M_s) \\ N(\underline{N_1, \ldots, N_t}, N_{t+1}, \ldots, N_u) \\ R(\underline{M_1, \ldots, M_r, N_1, \ldots, N_t}, R_1, \ldots, R_x) \end{array}$$

The resultant ClassSheet is shown in Fig. 16.

The difference between this ClassSheet model and the general one is that the **MKey** class on the top does not contain any attribute from R: all its attributes are contained in the cell class. This rule is from now on named *rule A2*.

Case 3. Finally, the third case occurs when the relationship is composed only by key attributes as illustrated next:

$$\begin{array}{l} M(\underline{M_1, \ldots, M_r}, M_{r+1}, \ldots, M_s) \\ N(\underline{N_1, \ldots, N_t}, N_{t+1}, \ldots, N_u) \\ R(\underline{M_1, \ldots, M_r, N_1, \ldots, N_t}) \end{array}$$

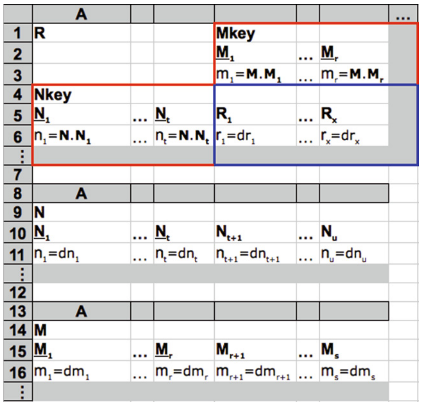
In this situation, the attributes that appear in the cell class are the non-key attributes of **N** and no class is created for **N**. Figure 17 illustrates this case. From now on this rule is named *rule A3*.

For case *(B)*, that is, for relationships between more than two tables, we choose between the candidates to span the cell class using the following criteria:

1. **M** and **N** should have small keys;
2. the number of empty cells created by the cell class should be minimal.

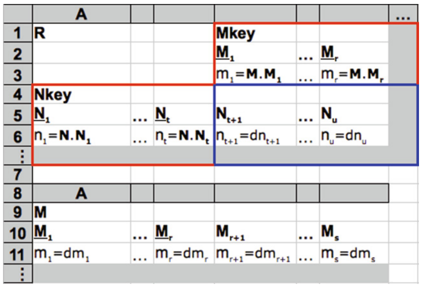
This rule is from now on named *rule B*.

After having chosen the two relations (and the relationship), the generation proceeds as described above. The remaining relations are created as explained in the beginning of this section.



	A						...
1	**R**			**Mkey**			
2				$\underline{M_1}$	...	$\underline{M_r}$	
3				m_1=**M.M$_1$**	...	m_r=**M.M$_r$**	
4	**Nkey**						
5	$\underline{N_1}$	...	$\underline{N_t}$	R_1	...	R_x	
6	n_1=**N.N$_1$**	...	n_t=**N.N$_t$**	$r_1=dr_1$	...	$r_x=dr_x$	
⋮							
7							
8	A						
9	**N**						
10	$\underline{N_1}$	...	$\underline{N_t}$	N_{t+1}	...	N_u	
11	$n_1=dn_1$	...	$n_t=dn_t$	$n_{t+1}=dn_{t+1}$	...	$n_u=dn_u$	
⋮							
12							
13	A						
14	**M**						
15	$\underline{M_1}$	...	$\underline{M_r}$	M_{r+1}	...	M_s	
16	$m_1=dm_1$	...	$m_r=dm_r$	$m_{r+1}=dm_{r+1}$	...	$m_s=dm_s$	
⋮							

Fig. 16. ClassSheet of a relationship with all the key attributes being foreign keys.



	A						...
1	**R**			**Mkey**			
2				$\underline{M_1}$	...	$\underline{M_r}$	
3				m_1=**M.M$_1$**	...	m_r=**M.M$_r$**	
4	**Nkey**						
5	$\underline{N_1}$	...	$\underline{N_t}$	N_{t+1}	...	N_u	
6	n_1=**N.N$_1$**	...	n_t=**N.N$_t$**	$n_{t+1}=dn_{t+1}$	...	$n_u=dn_u$	
⋮							
7							
8	A						
9	**M**						
10	$\underline{M_1}$	...	$\underline{M_r}$	M_{r+1}	...	M_s	
11	$m_1=dm_1$	...	$m_r=dm_r$	$m_{r+1}=dm_{r+1}$	...	$m_s=dm_s$	
⋮							

Fig. 17. ClassSheet of a relationship composed only by key attributes.

4.3 Mapping Strategy

In this section we present the mapping function between RID graphs and ClassSheets, which builds on the rules presented before. For that, we use the common strategic combinators listed below [48,73,74]:

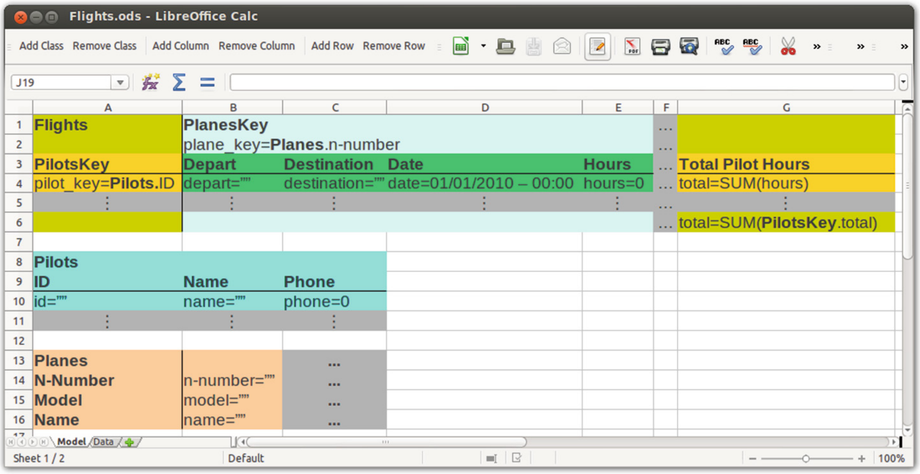
nop	$:: Rule$	-- identity
$\triangleright$	$:: Rule \to Rule \to Rule$	-- sequential composition
$\oslash$	$:: Rule \to Rule \to Rule$	-- left-biased choice
$many$	$:: Rule \to Rule$	-- repetition
$once$	$:: Rule \to Rule$	-- arbitrary depth rule application

In this context, *Rule* encodes a transformation from RID graphs to ClassSheets.

Using the rules defined in the previous section and the combinators listed above, we can construct a strategy that generates a ClassSheet:

$$
\begin{aligned}
&genCS = \\
&\quad many\ (once\ (rule\ B)) \triangleright \\
&\quad many\ (once\ (rule\ A)) \triangleright
\end{aligned}
$$

$$
\begin{array}{l}
\textit{many}\ (\textit{once}\ (\textit{rule A1})\ \oslash\ \textit{once}\ (\textit{rule A2})\ \oslash\ \textit{once}\ (\textit{rule A3}))\ \triangleright \\
\textit{many}\ (\textit{once}\ (\textit{rule}\ 2))\ \triangleright \\
\textit{many}\ (\textit{once}\ (\textit{rule}\ 1))
\end{array}
$$



	A	B	C	D	E	F	G
1	Flights	PlanesKey				...	
2		plane_key=Planes.n-number				...	
3	PilotsKey	Depart	Destination	Date	Hours	...	Total Pilot Hours
4	pilot_key=Pilots.ID	depart=""	destination=""	date=01/01/2010 – 00:00	hours=0	...	total=SUM(hours)
5	⋮	⋮	⋮	⋮	⋮	...	⋮
6						...	total=SUM(PilotsKey.total)
7							
8	Pilots						
9	ID	Name	Phone				
10	id=""	name=""	phone=0				
11	⋮	⋮	⋮				
12							
13	Planes		...				
14	N-Number	n-number=""	...				
15	Model	model=""	...				
16	Name	name=""	...				

Fig. 18. The ClassSheet generated by our algorithm for the running example.

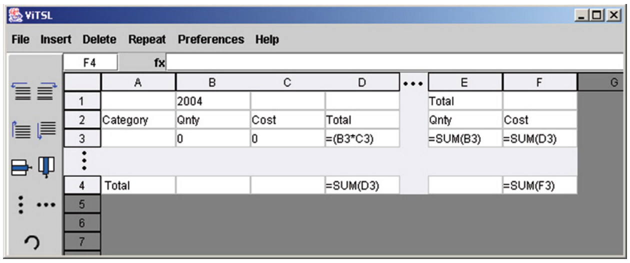
The strategy works as follows: it tries to apply *rule B* as many times as possible, consuming all the relationships with more than two relations; it then tries to apply *rule A* as many times as possible, consuming relationships with two relations; next the three sub-cases of *rule A* are applied as many times as possible consuming all the relationships with two relations that match some of the sub-rules; after consuming all the relationships and corresponding relations, the strategy consumes all the relations that are connected through a foreign key using *rule 2*; finally, all the remaining relations are mapped using *rule 1*.

In Fig. 18 we present the ClassSheet model that is generated by our tool for the flight scheduling spreadsheet.

4.4 Generation of Model-Driven Spreadsheets

Together with the definition of ClassSheet models, Erwig et al. developed a visual tool, ViTSL, to allow the easy creation and manipulation of the visual representation of ClassSheet models [9]. The visual and domain specific modeling language used by ViTSL is visually similar to spreadsheets (see Fig. 19).

The approach proposed by Erwig et al. follows a traditional compiler construction architecture [10] and generative approach [49]: first a language is defined (a visual domain specific language, in this case). Then a specific tool/compiler (the ViTSL tool, in this case) compiles it into a target lower level representation: an *Excel* spreadsheet. This generated representation is then interpreted by a different software system: the *Excel* spreadsheet system through the *Gencel* extension [33]. Given that model representation, *Gencel* generates an initial spreadsheet instance



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

Fig. 19. Screen shot of the ViTSL editor, taken from [9].

(conforming to the model) with embedded (spreadsheet) operations that express the underlying business logic. The architecture of these tools is shown in Fig. 20.

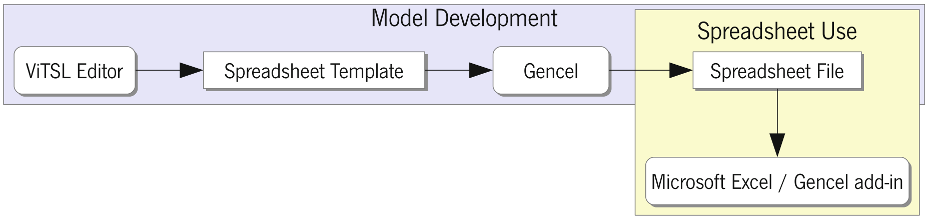


Fig. 20. ViTSL/*Gencel*-based environment for spreadsheet development.

The idea is that, when using such generated spreadsheets, end users are restricted to only perform operations that are logically and technically correct for that model. The generated spreadsheet not only guides end users to introduce correct data, but it also provides operations to perform some repetitive tasks like the repetition of a set of columns with some default values.

In fact, this approach provides a form of model-driven software development for spreadsheet users. Unfortunately, it provides a very limited form of model-driven spreadsheet development: it does not support model/instance synchronization. Indeed, if the user needs to evolve the model, then he has to do it using the ViTSL tool. Then, the tool compiles this new model to a new *Excel* spreadsheet instance. However, there are no techniques to co-evolve the spreadsheet data from the new instance to the newly generated one. In the next sections, we present embedded spreadsheet models and data refinement techniques that provide a full model-driven spreadsheet development setting.

4.5 Embedding ClassSheet Models in Spreadsheets

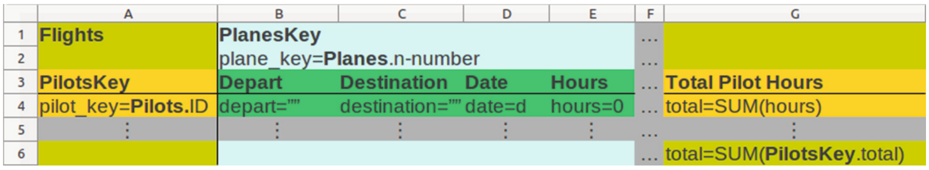
The ClassSheet language is a domain specific language to represent the business model of spreadsheet data. Furthermore, as we have seen in the previous section, the visual representation of ClassSheets very much resembles spreadsheets themselves. Indeed, the visual representation of ClassSheet models is a Visual Domain Specific Language. These two facts combined motivated the use of spreadsheet systems to define ClassSheet models [26], i.e., to natively embed ClassSheets in a spreadsheet host system. In this line, we have adopted the well-known techniques to embed Domain Specific Languages (DSL) in a host general purpose language [38,44,70]. In this way, both the model and the spreadsheet can be stored in the same file, and model creation along with data editing can be handled in the same environment that users are familiar with.

The embedding of ClassSheets within spreadsheets is not direct, since ClassSheets were not meant to be embedded inside spreadsheets. Their resemblance helps, but some limitations arise due to syntactic restrictions imposed by spreadsheet host systems. Several options are available to overcome the syntactic restrictions, like writing a new spreadsheet host system from start, modifying an existing one, or adapting the ClassSheet visual language. The two first options are not viable to distribute Model-Driven Spreadsheet Engineering (MDSE) widely, since both require users to switch their system, which can be inconvenient. Also, to accomplish the first option would be a tremendous effort and would change the focus of the work from the embedding to building a tool.

The solution adopted modifies slightly the ClassSheet visual language so it can be embedded in a worksheet without doing major changes on a spreadsheet host system (see Fig. 21). The modifications are:

1. identify expansion using cells (in the ClassSheet language, this identification is done between columns/rows letters/numbers);
2. draw an expansion limitation black line in the spreadsheet (originally this is done between column/row letters/numbers);
3. fill classes with a background color (instead of using lines as in the original ClassSheets).

The last change (3) is not mandatory, but it is easier to identify the classes and, along with the first change (2), eases the identification of classes' parts. This way, users do not need to think which role the line is playing (expansion limitation or class identification).



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

Fig. 21. Embedded ClassSheet for the flights' table.

We can use the flights' table to compare the differences between the original ClassSheet and its embedded representation:

– In the original ClassSheet (Fig. 9a), there are two expansions: one denoted by the column between columns `E` and `F` for the horizontal expansion, and another denoted by the row between rows `4` and `5` for the vertical one. Applying

change 1 to the original model will add an extra column (F) and an extra row (5) to identify the expansions in the embedding (Fig. 21).
– To define the expansion limits in the original ClassSheet, there are no lines between the column headers of columns B, C, D and E which makes the horizontal expansion to use three columns and the vertical expansion only uses one row. This translates to a line between columns A and B and another line between rows 3 and 4 in the embedded ClassSheet as per change 2.
– To identify the classes, background colors are used (change 3), so that the class **Flights** is identified by the green[11] background, the class **PlanesKey** by the cyan background, the class **PilotsKey** by the yellow background, and the class that relates the **PlanesKey** with the **PilotsKey** by the dark green background. Moreover, the relation class (range B3:E5), called **PilotsKey_PlanesKey**, is colored in dark green.

Given the embedding of the spreadsheet model in one worksheet, it is now possible to have one of its instances in a second worksheet, as we will shortly discuss. As we will also see, this setting has some advantages: for once, users may evolve the model having the data automatically coevolved. Also, having the model near the data helps to document the latter, since users can identify clearly the structure of the logic behind the spreadsheet. Figure 22a illustrates the complete embedding for the ClassSheet model of the running example, whilst Fig. 22b shows one of its possible instances.

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

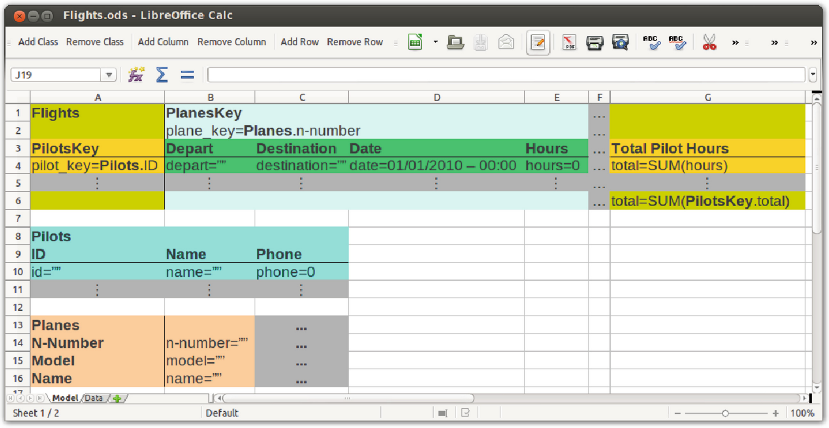
Model Creation. To create a model, several operations are available such as addition and deletion of columns and rows, cell editing, and addition or deletion of classes.

To create, for example, the flights' part of the spreadsheet used so far, one can:

1. add a class for the flights, selecting the range A1:G6 and choosing the green color for its background;
2. add a class for the planes, selecting the range B1:F6, choosing the cyan color for its background, and setting the class to expand horizontally;
3. add a class for the pilots, selecting the range A3:G5, choosing the yellow color for its background, and setting the class to expand vertically; and,
4. set the labels and formulas for the cells.

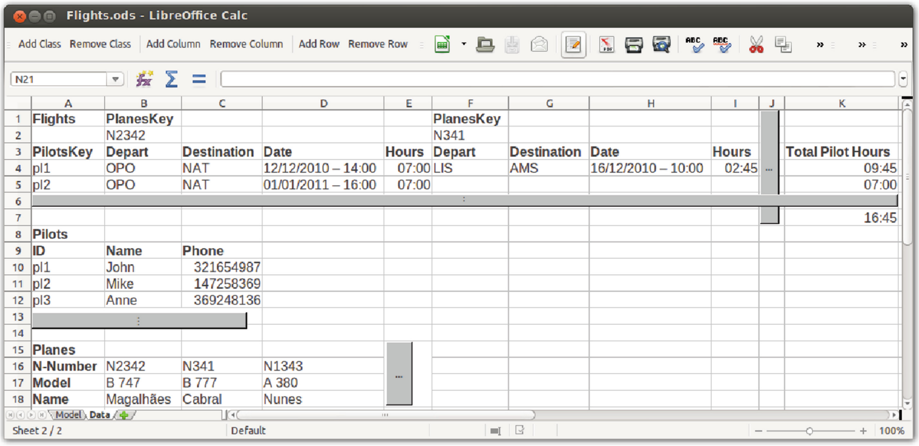
The addition of the relation class (range B3:E4) is not needed since it is added automatically when the environment detects superposing classes at the same level (**PlanesKey** and **PilotsKey** are within **Flights**, which leads to the automatic insertion of the relation class).

[11] We assume colors are visible in the digital version of this paper.


Flights.ods - LibreOffice Calc
Add Class Remove Class Add Column Remove Column Add Row Remove Row
J19
Model Data
Sheet 1 / 2
Default
100%

	A	B	C	D	E	F	G
1	Flights	PlanesKey				...	
2		plane_key=**Planes**.n-number				...	
3	PilotsKey	Depart	Destination	Date	Hours	...	Total Pilot Hours
4	pilot_key=**Pilots**.ID	depart=""	destination=""	date=01/01/2010 – 00:00	hours=0	...	total=SUM(hours)
5	⋮	⋮	⋮	⋮	⋮	...	⋮
6						...	total=SUM(**PilotsKey**.total)
7							
8	Pilots						
9	ID	Name	Phone				
10	id=""	name=""	phone=0				
11	⋮	⋮	⋮				
12							
13	Planes		...				
14	N-Number	n-number=""	...				
15	Model	model=""	...				
16	Name	name=""	...				

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


Flights.ods - LibreOffice Calc
Add Class Remove Class Add Column Remove Column Add Row Remove Row
N21
Model Data
Sheet 2 / 2
Default
100%

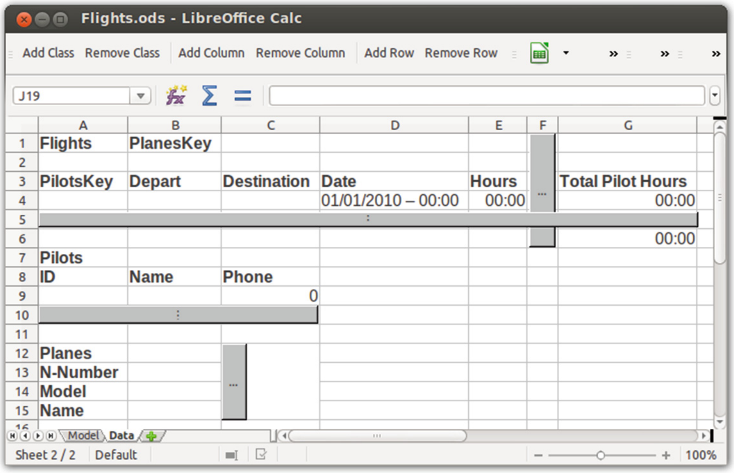
	A	B	C	D	E	F	G	H	I	J	K
1	Flights	PlanesKey				PlanesKey					
2		N2342				N341					
3	PilotsKey	Depart	Destination	Date	Hours	Depart	Destination	Date	Hours		Total Pilot Hours
4	pl1	OPO	NAT	12/12/2010 – 14:00	07:00	LIS	AMS	16/12/2010 – 10:00	02:45	...	09:45
5	pl2	OPO	NAT	01/01/2011 – 16:00	07:00						07:00
6					⋮						
7											16:45
8	Pilots										
9	ID	Name	Phone								
10	pl1	John	321654987								
11	pl2	Mike	147258369								
12	pl3	Anne	369248136								
13		⋮									
14											
15	Planes										
16	N-Number	N2342	N341	N1343							
17	Model	B 747	B 777	A 380	...						
18	Name	Magalhães	Cabral	Nunes							

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

Fig. 22. Flights' spreadsheet, with an embedded model and a conforming instance.

Instance Generation. From the flights' model described above, an instance without any data can be generated. This is performed by copying the structure of the model to another worksheet. In this process labels copied as they are, and attributes are replaced in one of two ways: *(i)*, if the attribute is simple (i.e., it is like $a = \varphi$), it is replaced by its default value; *(ii)*, otherwise, it is replaced by an instance of the formula. An instance of a formula is similar to the original one defined in the model, but the attribute references are replaced by references to cells where those attributes are instantiated. Moreover, columns and rows with ellipses have no content, having instead buttons to perform operations of adding new instances of their respective classes.

An empty instance generated by the flights' model is pictured in Fig. 23. All the labels (text in bold) are the same as the ones in the model, and in the same position, attributes have the default values, and four buttons are available to add new instances of the expandable classes.



	A	B	C	D	E	F	G
1	**Flights**	**PlanesKey**					
2							
3	**PilotsKey**	**Depart**	**Destination**	**Date**	**Hours**		**Total Pilot Hours**
4				01/01/2010 – 00:00	00:00	...	00:00
5				⋮			
6							00:00
7	**Pilots**						
8	**ID**	**Name**	**Phone**				
9			0				
10		⋮					
11							
12	**Planes**						
13	**N-Number**		...				
14	**Model**						
15	**Name**						

Fig. 23. Spreadsheet generated from the flights' model.

Data Editing. The editing of the data is performed like with plain spreadsheets, i.e., the user just edits the cell content. The insertion of new data is different since editing assistance must be used through the buttons available.

For example, to insert a new flight for pilot `pl1` in the **Flights** table, without models one would need to:

1. insert four new columns;
2. copy all the labels;
3. update all the necessary formulas in the last column; and,
4. insert the values for the new flight.

With a large spreadsheet, the step to update the formulas can be very error prone, and users may forget to update all of them. Using models, this process consists on two steps only:

1. press the button with label "· · ·" (in column `J`, Fig. 22b); and,
2. insert the values for the new flight.

The model-driven environment automatically inserts four new columns, the labels for those columns, updates the formulas, and inserts default values in all the new input cells.

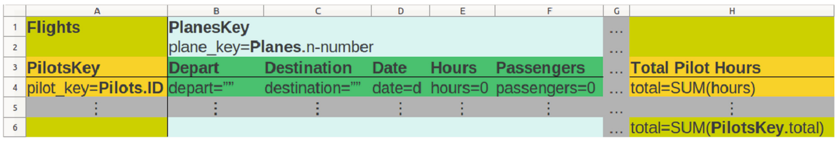
Note that, to keep the consistency between instance and model, all the cells in the instance that are not data entry cells are non-editable, that is, all the labels and formulas cannot be edited in the instance, only in the model. In Sect. 5 we will detail how to handle model evolutions.

Embedded Domain Specific Languages. In this section we have described the embedding of a visual, domain specific language in a general purpose visual spreadsheet system. The embedding of textual DSLs in host functional programming languages is a well-known technique to develop DSLs [44,70]. In our visual embedding, and very much like in textual languages, we get for free the powerful features of the host system: in our case, a simple, but powerful visual programming environment. As a consequence, we did not have to develop from scratch such a visual system (like the developers of ViTSL did). Moreover, we offer a visual interface familiar to users, namely, a spreadsheet system. Thus, they do not have to learn and use a different system to define their spreadsheet models.

The embedding of DSL is also known to have disadvantages when compared to building a specific compiler for that language. Our embedding is no exception: firstly, when building models in our setting, we are not able to provide domain-specific feedback (that is, error messages) to guide users. For example, a tool like ViTSL can produce better error messages and support for end users to construct (syntactic) correct models. Secondly, there are some syntactic limitations offered by the host language/system. In our embedding, we can see the syntactic differences in the vertical/horizontal ellipses defined in visual and embedded models (see Figs. 9 and 18).

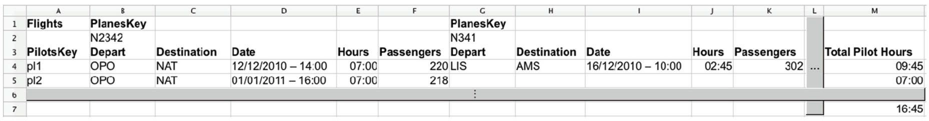
5 Evolution of Model-Driven Spreadsheets

The example we have been using manages pilots, planes and flights, but it misses a critical piece of information about flights: the number of passengers. In this case, additional columns need to be inserted in the block of each flight. Figure 24 shows an evolved spreadsheet with new columns (`F` and `K`) to store the number of passengers (Fig. 22b), as well as the new model that it instantiates (Fig. 22a).



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

(a) Evolved flights' model.



	A	B	C	D	E	F	G	H	I	J	K	L	M
1	**Flights**	**PlanesKey**					**PlanesKey**						
2		N2342					N341						
3	**PilotsKey**	**Depart**	**Destination**	**Date**	**Hours**	**Passengers**	**Depart**	**Destination**	**Date**	**Hours**	**Passengers**		**Total Pilot Hours**
4	pl1	OPO	NAT	12/12/2010 – 14:00	07:00	220	LIS	AMS	16/12/2010 – 10:00	02:45	302	...	09:45
5	pl2	OPO	NAT	01/01/2011 – 16:00	07:00	218							07:00
6							⋮						
7													16:45

(b) Evolved flights' instance.

Fig. 24. Evolved spreadsheet and the model that it instantiates.

Note that a modification of the year block in the model (in this case, inserting a new column) captures modifications to all repetitions of the block throughout the instance.

In this section, we will demonstrate that modifications to spreadsheet models can be supported by an appropriate combinator language, and that these model modifications can be propagated automatically to the spreadsheets that instantiate the models [28]. In the case of the flights example, the model modification is captured by the following expression:

```
addPassengers = once
                        (inside "PilotsKey_PlanesKey"
                          (after "Hours"
                            (insertCol "Passengers")))
```

The actual column insertion is done by the innermost *insertCol* step. The *after* and *inside* combinators specify the location constraints of applying this step. The *once* combinator traverses the spreadsheet model to search for a single location where these constraints are satisfied and the insertion can be performed.

The application of *addPassengers* to the initial model (Fig. 22a) will yield:

1. the modified model (Fig. 24a),
2. a spreadsheet migration function that can be applied to instances of the initial model (e.g. Fig. 22b) to produce instances of the modified model (e.g. Fig. 24b), and
3. an inverse spreadsheet migration function to backport instances of the modified model to instances of the initial model.

In the remaining of this section we will explain the machinery required for this type of coupled transformation of spreadsheet instances and models.

5.1 A Framework for Evolution of Spreadsheets in Haskell

Data refinement theory provides an algebraic framework for calculating with data types and corresponding values [52,54,55]. It consists of type-level coupled with value-level transformations. The type-level transformations deal with the evolution of the model and the value-level transformations deal with the instances of the model (e.g. values). Figure 25 depicts the general scenario of a transformation in this framework.

$$\begin{tikzcd} A \arrow[r, bend left, "to"] \arrow[r, phantom, "\leqslant"] & A' \arrow[l, bend left, "from"] \end{tikzcd}$$

A, A'	data type and transformed data type
to	witness function of type $A \to A'$ (injective)
from	witness function of type $A' \to A$ (surjective)

Fig. 25. Coupled transformation of data type A into data type A'.

Each transformation is coupled with witness functions *to* and *from*, which are responsible for converting values of type A into type A' and back.

2LT is a framework written in Haskell implementing this theory [12,18]. It provides the basic combinators to define and compose transformations for data types and witness functions. Since 2LT is statically typed, transformations are guaranteed to be type-safe ensuring consistency of data types and data instances.

To represent the witness functions *from* and *to* 2LT relies once again on the definition of a *Generalized Algebraic Data Type*[12] (GADT) [43,61]:

```
data PF a where
  id      :: PF (a → a)                                    -- identity function
  π₁      :: PF ((a, b) → a)                        -- left projection of a pair
  π₂      :: PF ((a, b) → b)                       -- right projection of a pair
  pnt     :: a → PF (One → a)                                       -- constant
  · △ ·   :: PF (a → b) → PF (a → c) → PF (a → (b, c))
                                                              -- split of functions
  · × ·   :: PF (a → b) → PF (c → d) → PF ((a, c) → (b, d))
                                                            -- product of functions
  · ∘ ·   :: Type b → PF (b → c) → PF (a → b) → PF (a → c)
                                                        -- composition of functions
  ·⋆      :: PF (a → b) → PF ([a] → [b])                      -- map of functions
  head    :: PF ([a] → a)                                         -- head of a list
  tail    :: PF ([a] → [a])                                       -- tail of a list
  fhead   :: PF (Formula1 → RefCell)        -- head of the args. of a formula
  ftail   :: PF (Formula1 → Formula1)       -- tail of the args. of a formula
```

This GADT represents the types of the functions used in the transformations. For example, π_1 represents the type of the function that projects the first part of a pair. The comments should clarify which function each constructor represents. Given these representations of types and functions, we can turn to the encoding of refinements. Each refinement is encoded as a two-level rewriting rule:

```
type Rule = ∀ a . Type a → Maybe (View (Type a))
data View a where View :: Rep a b → Type b → View (Type a)
data Rep a b = Rep {to = PF (a → b), from = PF (b → a)}
```

Although the refinement is from a type a to a type b, this can not be directly encoded since the type b is only known when the transformation completes, so the type b is represented as a *view* of the type a. A *view* expresses that a type a can be represented as a type b, denoted as *Rep* a b, if there are functions *to* $:: a \rightarrow b$ and *from* $:: b \rightarrow a$ that allow data conversion between one and the other. *Maybe* encapsulates an optional value: a value of type *Maybe* a either contains a value of type a (*Just* a), or it is empty (*Nothing*).

To better explain this system we will show a small example. The following code implements a rule to transform a list into a map (represented by $\cdot \rightharpoonup \cdot$):

```
listmap :: Rule
listmap ([a]) = Just (View (Rep {to = seq2index, from = tolist}) (Int ⇀ a))
listmap _ = mzero
```

[12] "It allows to assign more precise types to data constructors by restricting the variables of the datatype in the constructors' result types."

The witness functions have the following signature (for this example their code is not important):

$$\mathit{tolist} :: (\mathit{Int} \rightharpoonup a) \to [a]$$
$$\mathit{seq2index} :: [a] \to (\mathit{Int} \rightharpoonup a)$$

This rule receives the type of a list of a, $[a]$, and returns a view over the type map of integers to a, $\mathit{Int} \rightharpoonup a$. The witness functions are returned in the representation *Rep*. If other argument than a list is received, then the rule fails returning *mzero*. All the rules contemplate this last case and so we will not show it in the definition of other rules.

ClassSheets and Spreadsheets in Haskell. The 2LT was originally designed to work with algebraic data types. However, this representation is not expressive enough to represent ClassSheet specifications or their spreadsheet instances. To overcome this issue, we extended the 2LT representation so it could support ClassSheet models, by introducing the following GADT:

```
data Type a where
  ...
  Value    :: Value → Type Value                                 -- plain value
  Ref      :: Type b → PF (a → RefCell) → PF (a → b) → Type a
              → Type a                                           -- references
  RefCell  :: Type RefCell                                       -- reference cell
  Formula  :: Formula1 → Type Formula1                           -- formulas
  LabelB   :: String → Type LabelB                               -- block label
  · = ·    :: Type a → Type b → Type (a, b)                      -- attributes
  · | ·    :: Type a → Type b → Type (a, b)     -- block horizontal comp.
  · ˆ ·    :: Type a → Type b → Type (a, b)       -- block vertical comp.
  EmptyB   :: Type EmptyB                                        -- empty block
  ·̲        :: String → Type HorH                -- horizontal class label
  | ·      :: String → Type VerV                   -- vertical class label
  | ·̲      :: String → Type Square                   -- square class label
  LabRel   :: String → Type LabS                             -- relation class
  · : ·    :: Type a → Type b → Type (a, b)                   -- labeled class
  · : (·)↓ :: Type a → Type b → Type (a, [b])    -- labeled expand. class
  · ˆ ·    :: Type a → Type b → Type (a, b)        -- class vertical comp.
  SheetC   :: Type a → Type (SheetC a)                          -- sheet class
  ·→       :: Type a → Type [a]                 -- sheet expandable class
  · | ·    :: Type a → Type b → Type (a, b)     -- sheet horizontal comp.
  EmptyS   :: Type EmptyS                                       -- empty sheet
```

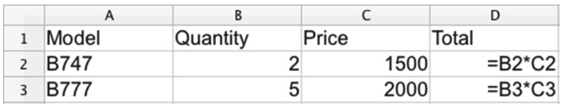
The comments should clarify what the constructors represent. The values of type *Type a* are representations of type a. For example, if t is of type *Type Value*, then t represents the type *Value*. The following types are needed to construct values of type *Type a*:

data *EmptyBlock* -- empty block
data *EmptySheet* -- empty sheet
type *LabelB* = *String* -- label
data *RefCell* = *RefCell1* -- referenced cell
type *LabS* = *String* -- square label
type *HorH* = *String* -- horizontal label
type *VerV* = *String* -- vertical label
data *SheetC a* = *SheetCC a* -- sheet class
data *SheetCE a* = *SheetCEC a* -- expandable sheet class
data *Value* = *VInt Int* | *VString String* | *VBool Bool* | *VDouble Double* -- values
data *Formula1* = *FValue Value* | *FRef* | *FFormula String* [*Formula1*] -- formula

Once more, the comments should clarify what each type represents. To explain this representation we will use as an example a small table representing the costs of maintenance of planes. We do not use the running example as it would be very complex to explain and understand. For this reduced model only four columns were defined: *plane model*, *quantity*, *cost per unit* and *total cost* (product of *quantity* by *cost per unit*). The Haskell representation of such model is shown next.

$$
\begin{array}{l}
costs = \\
\quad \mid Cost : Model \shortmid Quantity \shortmid Price \shortmid Total\,\hat{} \\
\quad \mid Cost : (model = \texttt{""} \shortmid quantity = 0 \shortmid price = 0 \shortmid total = \\
FFormula\ \texttt{"}\times\texttt{"}\ [FRef, FRef])^{\downarrow}
\end{array}
$$

This ClassSheet specifies a class called *Cost* composed by two parts vertically composed as indicated by the ˆ operator. The first part is specified in the first row and defines the labels for four columns: *Model*, *Quantity*, *Price* and *Total*. The second row models the rest of the class containing the definition of the four columns. The first column has default value the empty string (""), the two following columns have as default value 0, and the last one is defined by a formula (explained latter on). Note that this part is vertical expandable. Figure 26 represents a spreadsheet instance of this model.



	A	B	C	D
1	Model	Quantity	Price	Total
2	B747	2	1500	=B2*C2
3	B777	5	2000	=B3*C3

Fig. 26. Spreadsheet instance of the maintenance costs ClassSheet.

Note that in the definition of *Type a* the constructors combining parts of the spreadsheet (e.g. sheets) return a pair. Thus, a spreadsheet instance is written as nested pairs of values. The spreadsheet illustrated in Fig. 26 is encoded in Haskell as follows:

```
((Model  , (Quantity, (Price, Total))),
[("B747", (2          , (1500 , FFormula "×" [FRef, FRef]))),
 ("B777", (5          , (2000 , FFormula "×" [FRef, FRef])))])
```

The Haskell type checker statically ensures that the pairs are well formed and are constructed in the correct order.

Specifying References. Having defined a GADT to represent ClassSheet models, we need now a mechanism to define spreadsheet references. The safer way to accomplish this is making references strongly typed. Figure 27 depicts the scenario of a transformation with references. A reference from a cell s to the a cell t is defined using a pair of projections, *source* and *target*. These projections are statically-typed functions traversing the data type A to identify the cell defining the reference (s), and the cell to which the reference is pointing to (t). In this approach, not only the references are statically typed, but also always guaranteed to exist, that is, it is not possible to create a reference from/to a cell that does not exist.

$$\begin{tikzcd}
& {|s|} & \\
A \arrow[ur, bend left, "source"] \arrow[dr, bend right, "target"'] \arrow[rr, bend left=30, "to"] \arrow[rr, Rightarrow, "T" description] & & A' \arrow[ul, bend right, "source'"'] \arrow[dl, bend left, "target'"] \arrow[ll, bend left=30, "from"] \\
& {|t|} &
\end{tikzcd}$$

source Projection over type A identifying the reference
target Projection over type A identifying the referenced cell

$source' = source \circ from$
$target' = target \circ from$

Fig. 27. Coupled transformation of data type A into data type A' with references.

The projections defining the reference and the referenced type, in the transformed type A', are obtained by post-composing the projections with the witness function *from*. When *source*$'$ and *target*$'$ are normalized they work on A' directly rather than via A. The formula specification, as previously shown, is specified directly in the GADT. However, the references are defined separately by defining projections over the data type. This is required to allow any reference to access any part of the GADT.

Using the spreadsheet illustrated in Fig. 26, an instance of a reference from the formula *total* to *price* is defined as follows (remember that the second argument of *Ref* is the source (reference cell) and that the third is the target (referenced cell)):

$$\begin{array}{l} costWithReferences = \\ Ref\ Int\ (fhead \circ head \circ (\pi_2 \circ \pi_2 \circ \pi_2)^{\star} \circ \pi_2)\ (head \circ (\pi_1 \circ \pi_2 \circ \pi_2)^{\star} \circ \pi_2)\ cost \end{array}$$

The *source* function refers to the first *FRef* in the Haskell encoding shown after Fig. 26. The *target* projection defines the cell it is pointing to, that is, it defines a reference to the the value 1500 in column *Price*.

To help understand this example, we explain how *source* is constructed. Since the use of GADTs requires the definition of models combining elements in a pairwise fashion, π_2 is used to get the second element of the model (a pair), that is, the list of planes and their cost maintenance. Then, we apply $(\pi_2 \circ \pi_2 \circ \pi_2)^{\star}$ which will return a list with all the formulas. Finally *head* will return the first formula (the one in cell D2) from which *fhead* gets the first reference in a list of references, that is, the reference B2 that appears in cell D2.

Note that our reference type has enough information about the cells and thus we do not need value-level functions, that is, we do not need to specify the projection functions themselves, just their types. In the cases we reference a list of values, for example, constructed by the class expandable operator, we need to be specific about the element within the list we are referencing. For these cases, we use the type-level constructors *head* (first element of a list) and *tail* (all but first) to get the intended value in the list.

5.2 Evolution of Spreadsheets

In this section we define rules to perform spreadsheet evolution. These rules can be divided in three main categories: *Combinators*, used as helper rules, *Semantic* rules, intended to change the model itself (e.g. add a new column), and *Layout* rules, designed to change the visual arrangement of the spreadsheet (e.g. swap two columns).

Combinators. The semantic and the layout rules are defined to work on a specific part of the model. The combinators defined next are then used to apply those rules in the desired places.

Pull up all references. To avoid having references in different levels of the models, all the rules pull all references to the topmost level of the model. This allows to create simpler rules since the positions of all references are know and do not need to be changed when the model is altered. To pull a reference in a particular place we use the following rule (we show just its first case):

$$\begin{array}{l} pullUpRef :: Rule \\ pullUpRef\ ((Ref\ tb\ fRef\ tRef\ ta) \mid b2) = \mathbf{do} \\ \quad return\ (View\ idrep\ (Ref\ tb\ (fRef \circ \pi_1)\ (tRef \circ \pi_1)\ (ta \mid b2))) \end{array}$$

The representation *idrep* has the *id* function in both directions. If part of the model (in this case the left part of a horizontal composition) of a given type has a reference, it is pulled to the top level. This is achieved by composing the existing projections with the necessary functions, in this case π_1. This rule has two cases (left and right hand side) for each binary constructor (e.g. horizontal/vertical composition).

To pull up all the references in all levels of a model we use the rule

$$pullUpAllRefs = many\ (once\ pullUpRef)$$

The *once* operator applies the *pullUpRef* rule somewhere in the type and the *many* ensures that this is applied everywhere in the whole model.

Apply after and friends. The combinator *after* finds the correct place to apply the argument rule (second argument) by comparing the given string (first argument) with the existing labels in the model. When it finds the intended place, it applies the rule to it. This works because our rules always do their task on the right-hand side of a type.

```
after :: String → Rule → Rule
after label r (label′ ı a) | label ≡ label′ = do
  View s l′ ← r label′
  return (View (Rep {to = to s × id, from = from s × id}) (l′ ı a))
```

Note that this code represents only part of the complete definition of the function. The remaining cases, e.g. $\cdot\,\hat{}\,\cdot$, are not shown since they are quite similar to the one presented.

Other combinators were also developed, namely, *before*, *bellow*, *above*, *inside* and *at*. Their implementations are not shown since they are similar to the *after* combinator.

Semantic Rules. Given the support to apply rules in any place of the model given by the previous definitions, we now present rules that change the semantics of the model, that is, that change the meaning and the model itself, e.g., adding columns.

Insert a block. The first rule we present is one of the most fundamentals: the insertion of a new block into a spreadsheet. It is formally defined as follows:

$$\begin{tikzcd} Block \arrow[r, bend left, "id \triangle (pnt\ a)"] \arrow[r, phantom, "\leqslant"] & Block \shortmid Block \arrow[l, bend left, "\pi_1"] \end{tikzcd}$$

This diagram means that a horizontal composition of two blocks refines a block when witnessed by two functions, *to* and *from*. The *to* function, $id \triangle (pnt\ a)$, is a split: it injects the existing block in the first part of the result without modifications (id) and injects the given block instance a into the second part of the result. The *from* function is π_1 since it is the one that allows the recovery of the existent block. The Haskell version of the rule is presented next.

```
insertBlock :: Type a → a → Rule
insertBlock ta a tx | isBlock ta ∧ isBlock tx = do
  let rep = Rep {to = (id△(pnt a)), from = π₁}
  View s t ← pullUpAllRefs (tx ı ta)
  return (View (comprep rep s) t)
```

The function *comprep* composes two representations. This rule receives the type of the new block *ta*, its default instance *a*, and returns a *Rule*. The returned rule is itself a function that receives the block to modify *tx*, and returns a view of the new type. The first step is to verify if the given types are blocks using the function *isBlock*. The second step is to create the representation *rep* with the witness functions given in the above diagram. Then the references are pulled up in result type $tx \shortmid ta$. This returns a new representation s and a new type t (in fact, the type is the same $t = tx \shortmid ta$). The result view has as representation the composition of the two previous representations, *rep* and s, and the corresponding type t.

Rules to insert classes and sheets were also defined, but since these rules are similar to the rule to insert blocks, we omit them.

Insert a column. To insert a column in a spreadsheet, that is, a cell with a label *lbl* and the cell bellow with a default value *df* and vertically expandable, we first need to create a new class representing it: $clas =\mid lbl : lbl\hat{}(lbl = df^{\downarrow})$. The label is used to create the default value $(lbl, [\,])$. Note that since we want to create an expandable class, the second part of the pair must be a list. The final step is to apply *insertSheet*:

```
insertCol :: String → VFormula → Rule
insertCol l f@(FFormula name fs) tx | isSheet tx = do
  let clas =| lbl : lbl^(lbl = df↓)
  ((insertSheet clas (lbl, [])) ▷ pullUpAllRefs) tx
```

Note the use of the rule *pullUpAllRefs* as explained before. The case shown in the above definition is for a formula as default value and it is similar to the value case. The case with a reference is more interesting and is shown next:

```
insertCol l FRef tx | isSheet tx = do
  let clas =| lbl : Ref ⊥ ⊥ ⊥ (lbl^((lbl = RefCell)↓))
  ((insertSheet clas (lbl, [])) ▷ pullUpAllRefs) tx
```

Recall that our references are always local, that is, they can only exist with the type they are associated with. So, it is not possible to insert a column that references a part of the existing spreadsheet. To overcome this, we first create the reference with undefined functions and auxiliary type $(\perp)$. We then set these values to the intended ones.

```
setFormula :: Type b → PF (a → RefCell) → PF (a → b) → Rule
setFormula tb fRef tRef (Ref _ _ _ t) =
  return (View idrep (Ref tb fRef tRef t))
```

This rule receives the auxiliary type ($\mathit{Type}\ b$), the two functions representing the reference projections and adds them to the type. A complete rule to insert a column with a reference is defined as follows:

```
$insertFormula =$
  $(once\ (insertCol\ label\ FRef)) \rhd (setFormula\ auxType\ fromRef\ toRef)$
```

Following the original idea described previously in this section, we want to introduce a new column with the number of passengers in a flight. In this case, we want to insert a column in an existing block and thus our previous rule will not work. For these cases we write a new rule:

```
$insertColIn :: String \to VFormula \to Rule$
$insertColIn\ l\ (FValue\ v)\ tx \mid isBlock\ tx = \mathbf{do}$
  $\mathbf{let}\ block = lbl\,\hat{}\,(lbl = v)$
  $((insertBlock\ block\ (lbl, v)) \rhd pullUpAllRefs)\ tx$
```

This rule is similar to the previous one but it creates a block (not a class) and inserts it also after a block. The reasoning is analogous to the one in *insertCol*.

To add the column "Passengers" we can use the rule *insertColIn*, but applying it directly to our running example will fail since it expects a block and we have a spreadsheet. We can use the combinator *once* to achieve the desired result. This combinator tries to apply a given rule somewhere in a type, stopping after it succeeds once. Although this combinator already existed in the 2LT framework, we extended it to work for spreadsheet models/types.

Make it expandable. It is possible to make a block in a class expandable. For this, we created the rule *expandBlock*:

```
$expandBlock :: String \to Rule$
$expandBlock\ str\ (label : clas) \mid compLabel\ label\ str = \mathbf{do}$
  $\mathbf{let}\ rep = Rep\ \{to = id \times tolist, from = id \times head\}$
  $return\ (View\ rep\ (label : (clas)^{\downarrow}))$
```

It receives the label of the class to make expandable and updates the class to allow repetition. The result type constructor is $\cdot : (\cdot)^{\downarrow}$; the *to* function wraps the existing block into a list, *tolist*; and the *from* function takes the head of it, *head*. We developed a similar rule to make a class expandable. This corresponds to promote a class c to $c^{\rightarrow}$. We do not show its implementation here since it is quite similar to the one just shown.

Split. It is quite common to move a column in a spreadsheet from on place to another. The rule *split* copies a column to another place and substitutes the original column values by references to the new column (similar to create a pointer). The rule to move part of the spreadsheet is presented in Sect. 5.2. The first step of *split* is to get the column that we want to copy:

$getColumn :: String \rightarrow Rule$
$getColumn\ h\ t\ (l' \,\hat{}\, b1) \mid h \equiv l' = return\ (View\ idrep\ t)$

If the corresponding label is found, the vertical composition is returned. Note that as in other rules, this one is intended to be applied using the combinator *once*. As we said, we aim to write local rules that can be used at any level using the developed combinators.

In a second step the rule creates a new a class containing the retrieved block:

```
do View s c′ ← getBlock str c
   let nsh =| str : (c′)↓
```

The last step is to transform the original column that was copied into references to the new column. The rule $makeReferences :: String \rightarrow Rule$ receives the label of the column that was copied (the same as the new column) and creates the references. We do not shown the rest of the implementation because it is quite complex and will not help in the understanding of the paper.

Layout Rules. We will now describe rules focused on the layout of spreadsheets, that is, rules that do not add/remove information to/from the model, but only rearrange it.

Change orientation. The rule *toVertical* changes the orientation of a block from horizontal to vertical.

$toVertical :: Rule$
$toVertical\ (a \mid b) = return\ (View\ idrep\ (a \,\hat{}\, b))$

Note that since our value-level representation of these compositions are pairs, the *to* and the *from* functions are simply the identity function. The needed information is kept in the type-level with the different constructors. A rule to do the inverse was also designed but since it is quite similar to this one, we do not show it here.

Normalize blocks. When applying some transformations, the resulting types may not have the correct shape. A common example is to have as result the following type:

$A \mid B \,\hat{}\, C \mid D \,\hat{}$
$E \mid F$

However, given the rules in [29] to ensure the correctness of ClassSheets, the correct result is the following:

$A \mid B \mid D \,\hat{}$
$E \mid C \mid F$

The rule *normalize* tries to match these cases and correct them. The types are the ones presented above and the witness functions are combinations of π_1 and π_2.

```
$normalize :: Rule$
$normalize\ (a \mid b\ \hat{}\ c \mid d\ \hat{}\ e \mid f) = \mathbf{do}$
  $\mathbf{let}\ to = id \times \pi_1 \times id \circ \pi_1 \triangle \pi_1 \circ \pi_2 \triangle \pi_2 \circ \pi_1 \circ \pi_2 \times \pi_2$
       $from = \pi_1 \circ \pi_1 \triangle \pi_1 \circ \pi_2 \times \pi_1 \circ \pi_2 \triangle \pi_2 \circ \pi_2 \circ \pi_1 \triangle id \times \pi_2 \circ \pi_2$
  $return\ (View\ (Rep\ \{to = to, from = from\})\ (a \mid b \mid d\ \hat{}\ e \mid c \mid f))$
```

Although the migration functions seem complex, they just rearrange the order of the pairs so they have the correct arrangement.

Shift. It is quite common to move parts of the spreadsheet across it. We designed a rule to shift parts of the spreadsheet in the four possible directions. We show here part of the *shiftRight* rule, which, as suggested by its name, shifts a piece of the spreadsheet to the right. In this case, a block is moved and an empty block is left in its place.

```
$shiftRight :: Type\ a \rightarrow Rule$
$shiftRight\ ta\ b1 \mid isBlock\ b1 = \mathbf{do}$
  $Eq \leftarrow teq\ ta\ b1$
  $\mathbf{let}\ rep = Rep\ \{to = pnt\ (\bot :: EmptyBlock) \triangle id, from = \pi_2\}$
  $return\ (View\ rep\ (EmptyBlock \mid b1))$
```

The function *teq* verifies if two types are equal. This rule receives a type and a block, but we can easily write a wrapper function to receive a label in the same style of *insertCol*.

Another interesting case of this rules occurs when the user tries to move a block (or a sheet) that has a reference.

```
$shiftRight\ ta\ (Ref\ tb\ frRef\ toRef\ b1) \mid isBlock\ b1 = \mathbf{do}$
  $Eq \leftarrow teq\ ta\ b1$
  $\mathbf{let}\ rep = Rep\ \{to = pnt\ (\bot :: EmptyBlock) \triangle id, from = \pi_2\}$
  $return\ (View\ rep\ (Ref\ tb\ (frRef \circ \pi_2)\ (toRef \circ \pi_2)\ (EmptyBlock \mid b1))$
```

As we can see in the above code, the existing reference projections must be composed with the selector π_2 to allow to retrieve the existing block *b1*. Only after this it is possible to apply the defined selection reference functions.

Move blocks. A more complex task is to move a part of the spreadsheet to another place. We present next a rule to move a block.

```
$moveBlock :: String \rightarrow Rule$
$moveBlock\ str\ c = \mathbf{do}$
  $View\ s\ c' \leftarrow getBlock\ str\ c$
  $\mathbf{let}\ nsh =\mid str : c'$
  $View\ r\ sh \leftarrow once\ (removeRedundant\ str)\ (c \mid nsh)$
  $return\ (View\ (comprep\ s\ r)\ sh)$
```

After getting the intended block and creating a new class with it, we need to remove the old block using *removeRedundant*.

```
removeRedundant :: String → Rule
removeRedundant s (s′) | s ≡ s′ = do
  let rep = Rep {to = pnt (⊥ :: EmptyBlock), from = pnt s′}
  return (View rep EmptyBlock)
```

This rule will remove the block with the given label leaving an empty block in its place.

6 Model-Driven Spreadsheet Development in MDSheet

The embedding and evolution techniques presented previously have been implemented as an add-on to a widely used spreadsheet system, the OpenOffice/LibreOffice system. The add-on provides a model-driven spreadsheet development environment, named MDSheet, where a (model-driven) spreadsheet consists of two type of worksheets: `Sheet 0`, containing the embedded ClassSheet model, and `Sheet 1`, containing the spreadsheet data that conforms to the model. Users can interact both with the ClassSheet model and the spreadsheet data. Our techniques guarantee the synchronization of the two representations.

In such an model-driven environment, users can evolve the model by using standard editing/updating techniques as provided by spreadsheets systems. Our add-on/environment also provides predefined buttons that implement the usual ClassSheets evolution steps. Each button implements an evolution rule, as described in Sect. 5. For each button, we defined a BASIC script that interprets the desired functionality, and sends the contents of the spreadsheet (both the model and the data) to our Haskell-based co-evolution framework. This Haskell framework implements the co-evolution of the spreadsheet models and data presented in Sect. 5.

MDSheet also allows the development of ClassSheet models from scratch by using the provided buttons or by traditional editing. In this case, a first instance/spreadsheet is generated from the model which includes some business logic rules that assist users in the safe and correct introduction/editing of data. For example, in the spreadsheet presented in Fig. 22, if the user presses the button in column `J`, four new columns will automatically be inserted so the user can add more flights. This automation will also automatically update all formulas in the spreadsheet.

The global architecture of the model-driven spreadsheet development we constructed is presented in Fig. 28.

Tool and demonstration video availability. The MDSheet tool [24] and a video with a demonstration of its capabilities are available at the SSaaPP – SpreadSheets as a Programming Paradigm project's website[13].

In the next section we present in detail the empirical study we have organized and conducted to assess model-driven spreadsheets running through MDSheet.

[13] http://ssaapp.di.uminho.pt.

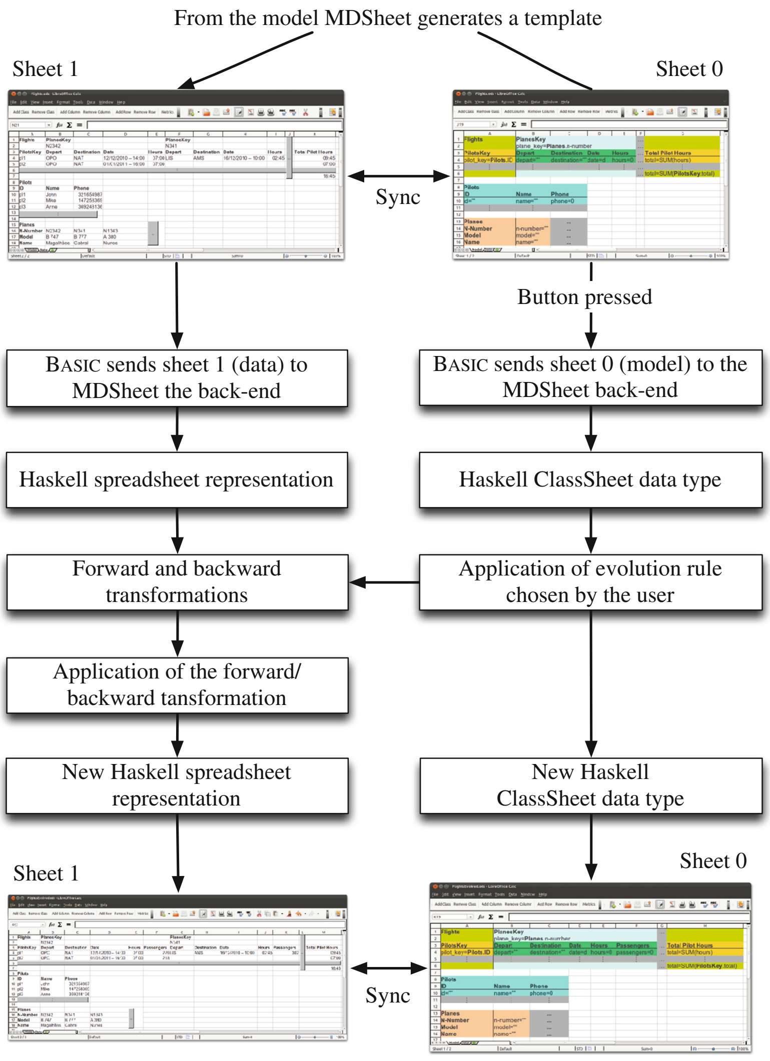

From the model MDSheet generates a template
Sheet 1
Sheet 0
Sync
Button pressed
BASIC sends sheet 1 (data) to MDSheet the back-end
BASIC sends sheet 0 (model) to the MDSheet back-end
Haskell spreadsheet representation
Haskell ClassSheet data type
Forward and backward transformations
Application of evolution rule chosen by the user
Application of the forward/ backward tansformation
New Haskell spreadsheet representation
New Haskell ClassSheet data type
Sheet 1
Sheet 0
Sync

Fig. 28. Model-driven spreadsheet development environment.

Exercise 7. *Consider the spreadsheet and corresponding model defined in Exercise 5. First, write the ClassSheet model in the MDSheet environment. Second, update the spreadsheet instance with the data.*

7 Studies with School Participants

In this section we present the feedback we obtained from school participants regarding spreadsheets and their engineering. This feedback was solicited in two different moments and aimed at realizing the participants perspective on two different spreadsheet aspects.

For once, we asked participants to name, from the characteristics that have been natively incorporated in spreadsheet systems, the ones they realized as the most important. Also, we asked them to name the single feature they missed the most. The details on this inquiry are presented in Sect. 7.1, and its results were already presented to the participants during the summer school.

Secondly, we relied on the participants' feedback to identify possible improvements for our framework. This occurred after our tutorial sessions, and during a lab session, where participants volunteered to actually perform concrete spreadsheet engineering tasks under the framework that we have built and that we have described in this tutorial. The details on this experiment are described in Sect. 7.2.

In both cases, we believe that the generalization of the results we observe here would require a larger sample of participants, namely for statistical reasons. Nevertheless, we also believe that the volunteer nature and the interest demonstrated by the participants when providing concrete feedback is surely worth its analyzis and publishing.

7.1 Participants' Perspective on Spreadsheets

In the beginning of our tutorial, we asked for the participants cooperation in filling in an inquiry on the spreadsheet characteristics they understood as the most important and on the feature they would like to see incorporated in traditional spreadsheet systems. We chose this moment to do so, since we wanted to understand the participants' perspective unbiased from the materials we later exposed.

The inquiry that we conducted consisted in handing a paper form to each participant, asking:

1. *Please provide the three most important characteristics of spreadsheets, in (descending) order of preference.*
2. *Please provide the feature you miss the most in spreadsheet systems.*

Answers were completely open, in that no pre-defined set of possible answers was given.

Figure 29 shows the feedback we received with respect to the first (most important) characteristic identified by the school participants.

Out of a total number of 29 answers, 17 (almost 60 %) identify the simplicity in their usage as the most important characteristic of spreadsheets. Also, the tabular layout of spreadsheets, with 5 answers (exactly 17 %), and their underlying incremental engine, with 3 (10 %), were significantly acknowledged. Finally, their flexibility, multi-purpose, availability on almost any computer as well as their presentation-oriented nature were also mentioned, with 1 answer each (nearly 3 % of all answers).

Next, we follow this same analysis regarding the characteristics pointed out as the second most important of spreadsheets, and that are presented in Fig. 30.

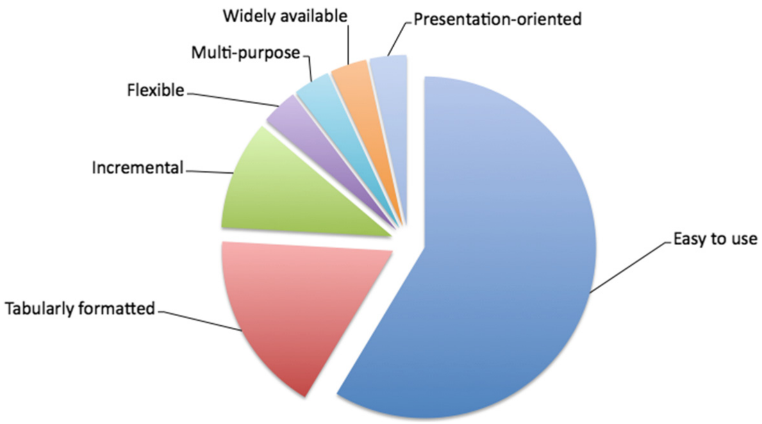


Fig. 29. Most important spreadsheet characteristic.

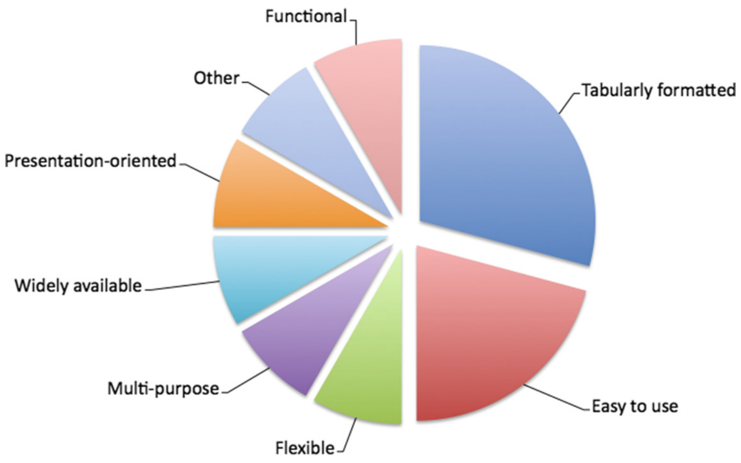


Fig. 30. Second most important spreadsheet characteristic.

In this case, we have received a total number of 24 answers. Out of these, the tabular nature of spreadsheets, with 7 answers (circa 30 %), and their simple usage, with 5 (around 10 %) are again the most pointed characteristics. Characteristics such as availability, multi-purpose, flexibility, functionality or presentation-orientation were all pointed out by 2 participants (i.e., by 8 % of all answers).

Regarding the answers that were given as the third most important spreadsheet characteristic, we have received a total of 20 valid answers, which are sketched in Fig. 31.

We observe a predominance of the availability and functionality of spreadsheets, with 4 answers each (i.e., 20 % of all answers each). The layout of

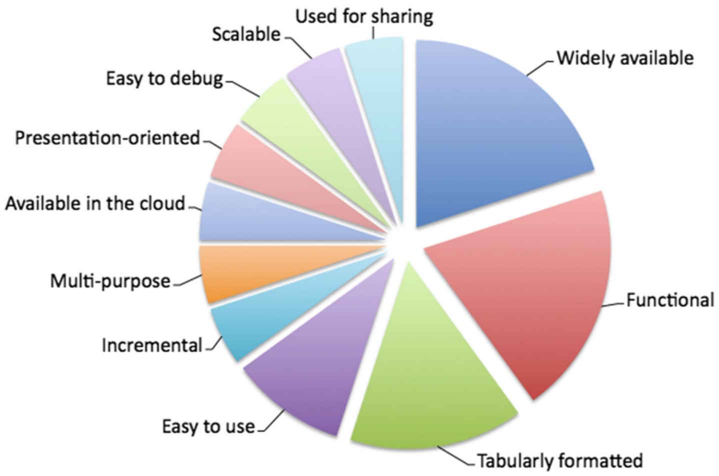


Fig. 31. Third most important spreadsheet characteristic.

spreadsheets was pointed by 3 participants (15 %) and all other characteristics were pointed out by a single participant (corresponding to 5 % of all answers).

Considering all the characteristics that were identified, irrespective to their order of preference, we obtain the results sketched in Fig. 32.

The top three identified characteristics were then the easiness of usage of spreadsheets, with 24 answers (33 %), their tabular format, 15 answers (21 %), and their availability, 7 answers (10 %).

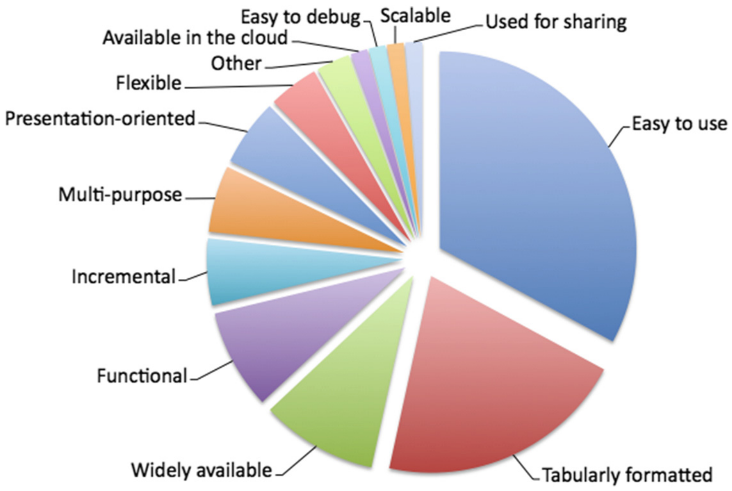


Fig. 32. Overall most important spreadsheet characteristic.

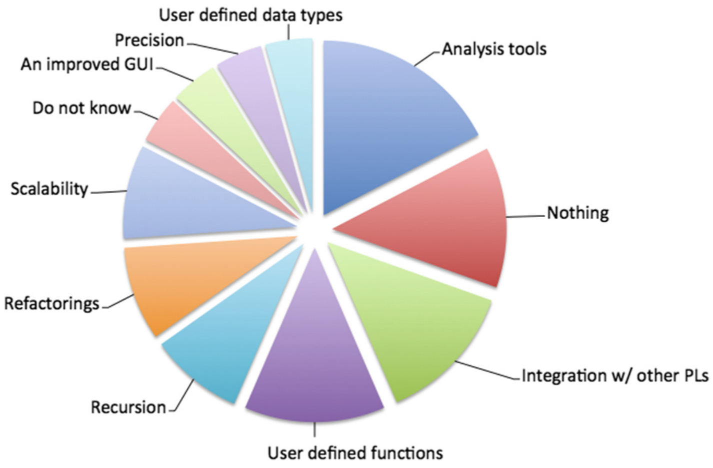


Fig. 33. The single most missed feature on spreadsheets.

Finally, regarding our pre-tutorial inquiry, participants were asked to identify the feature they missed the most on a spreadsheet system. The answers we received, in number of 23, are depicted in Fig. 33.

We see that analysis tools, integration with other programming languages, and user defined functions are the most missed features (with 4, 3 and 3 answers, respectively). Interestingly enough, 3 participants say that spreadsheet systems are fine in their current state, i.e., that they do not miss a single feature there. With 2 participants referring to them, recursion, refactorings, and a better scalability model are also identified as missing features. Finally, an improved graphical user interface, precision and user defined data types were also identified each by a single participant.

7.2 Participants' Perspective on MDSheet

In this section, we describe the simple experiment that we have devised in order to obtain feedback on the MDSheet framework from the participants. Five participants volunteered to join the experiment: 4 males and 1 female; all of them had never had contact with MDSheet prior to the summer school.

Our experiment consisted in executing three specific tasks. Prior to participants actually performing each one, we have ourselves demonstrated with equivalent actions. Also, the tasks that were solicited consisted of editing steps on an already built model to deal with a simple budget (registering incomes and expenses). These tasks followed the order:

1. Add an attribute to a class, being given its default value.
2. Add and attribute to a class, being its value defined by a given formula.
3. Remove an attribute from a class.

After the execution of each task, we made available a (web) form, where participants had the opportunity to answer the following questions:

(i) Did you find this functionality useful?
(ii) For this functionality, describe an advantage in using our environment.
(iii) For this functionality, describe a disadvantage in using our environment.
(iv) For this functionality, please give us a suggestion to improve our environment.
(v) Assuming that you are familiar with our environment, would you prefer to use standard *Excel* to complete this task? Please elaborate.

In the remaining of this section, we present our analysis on the feedback that was provided by participants.

Analyzing task 1. All participants found the functionality of adding attributes to a class useful. Also, they in general see as beneficial the fact that attributes may later be called by name, instead of by (alphabetical) column identifiers. In fact, this is in line with the results presented in [46] where authors showed that spreadsheet users create a mental model of the spreadsheet that helps them understand and work with the spreadsheet. These mental models are created using names from the real world as it is the case with our ClassSheet models.

In terms of disadvantages, they point out the fact that attribute names are not visible on the instances, and potential efficiency problems when using larger models. Some participants suggest that we should improve further our graphical user interface. Still, all participants state that they prefer using our environment over using a standard spreadsheet system for this type of action.

Analyzing task 2. All participants found the functionality of adding attributes whose values are given by a formula to a class useful. Indeed, they state that being able of defining formulas using attribute names is very intuitive and helpful. Also, they see as important the fact that a formula is defined only once, in the model, being automatically copied (and updated) wherever (and whenever) necessary at the instance level.

In terms of disadvantages, the one thing that is identified is that it would be better, when defining a formula, to be able to use the mouse to point to an attribute instead of having to type its name, as our tool in its current state demands. Actually, overcoming this disadvantage is the main suggestion for improvement that we receive here. For faster feedback, two participants state that they would prefer using *Excel* in the particular scenario that we have set and for this particular task. However, they also state that if they were dealing with a larger model, they would prefer MDSheet.

Analyzing task 3. Again, all participants found the functionality of removing an attribute from a class useful. All participants but one were particularly enthusiastic about the fact that all the necessary editions (e.g., in the scopes of all formulas affected by the deletion of the attribute) are automatically implemented.

Also, most participants found no disadvantages in using our framework for this type of tasks. Nevertheless, the issues that were raised here concern to the fact that using our model-based approach may sometimes restrict the flexibility of standard spreadsheets. The comments with respect to this task mainly suggest that we should make available a message confirming the will to delete data. Finally, no participant declared to prefer a standard spreadsheet system to accomplish a task such as this one.

Apart from feedback on accomplishing specific tasks, we also requested general feedback regarding MDSheet. Indeed, we asked each participant to choose the descriptions they believed were applicable to our framework from the following list. Any number of options was selectable.

Helpful	Not helpful	Useful for professional programmers
Usable	Requires specific knowledge	Useful for non-professional programmers
Intuitive	Counter intuitive	Can improve my productivity
Useless	Not useful in practice	Can not improve my productivity

The options that were selected, and the number of times they were selected is given next.

Helpful	4
Can improve my productivity	4
Useful for non-professional programmers	3
Usable	3
Intuitive	1
Requires specific knowledge	1

Finally, we asked for comments on MDSheet and for suggestions that could improve it. The most referred suggestion was to add an undo button to the framework. In another direction, one participant commented that our framework may be unfit for agile business practices.

As we explained before, this empirical study was performed in the laboratory sessions of our tutorial course. The number of participants in the study is small, and no statistical conclusions can be obtained from the study. However, we have conducted a larger study with end users where we evaluated their efficiency (measured as the time needed to complete a task) and effectiveness (measure as the number of errors produced in solving the task) using regular and model-driven spreadsheets [20]. Those results show that using our MDSD environment, end users are both more efficient and effective. In fact, those results just confirm the feedback we received from summer school participants.

8 Conclusion

This document presents a set of techniques and tools to analyze and evolve spreadsheets. First, it presents data mining and database techniques to infer a ClassSheet that represents the business logic of spreadsheet data. Next, it shows the embedding of the visual, domain specific language of ClassSheet in a general purpose spreadsheet system. Finally, it presents model-driven engineering techniques, based on data refinements, to evolve the model and have the instance automatically co-evolved. These techniques are implemented in the MDSheet framework: an add-on for a widely used, open source, spreadsheet system.

In order to validate both our embedding of a visual DSL and the evolution of our model-driven spreadsheets, we have conducted an empirical study with the summer school participants. The results show that regular spreadsheet users are able to perform the proposed tasks, and they recognize the advantages of using our setting when compared to standard spreadsheet systems.

The techniques and tools described in this paper were developed in the context of the SSaaPP - Spreadsheets as a Programming Paradigm research project. In the project's webpage, the reader may find the tools presented in this paper and other contributions in the area of spreadsheet engineering: namely the definition of a catalog of spreadsheet bad smells, the definition of a query language for model-driven spreadsheets, and a quality model for spreadsheets. They are available, as a set of research papers and software tools, at the following webpage:

http://ssaapp.di.uminho.pt

Acknowledgments. The theories, techniques and tools presented in this tutorial paper were developed under the project SSaaPP - SpreadSheets as a Programming Paradigm: a research project funded by the Portuguese Science Foundation (contract number `FCOMP-01-0124-FEDER-010048`). We would like to thank the members and consultants of this project who made important contributions for the results presented in this document, namely: Rui Maranhão Abreu, Tiago Alves, Laura Beckwith, Orlando Belo, Martin Erwig, Pedro Martins, Hugo Pacheco, Christophe Peixoto, Rui Pereira, Alexandre Perez, Hugo Ribeiro, André Riboira, André Silva, and Joost Visser.

References

1. Abraham, R., Erwig, M.: Header and unit inference for spreadsheets through spatial analyses. In: 2004 IEEE Symposium on Visual Languages and Human Centric Computing, pp. 165–172, September 2004
2. Abraham, R., Erwig, M.: UCheck: a spreadsheet type checker for end users. J. Vis. Lang. Comput. **18**(1), 71–95 (2007)
3. Abraham, R., Erwig, M.: Goal-directed debugging of spreadsheets. In: VL/HCC, pp. 37–44. IEEE Computer Society (2005)
4. Abraham, R., Erwig, M.: Autotest: a tool for automatic test case generation in spreadsheets. In: Proceedings of the 2006 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2006), pp. 43–50. IEEE Computer Society (2006)
5. Abraham, R., Erwig, M.: Inferring templates from spreadsheets. In: Proceedings of the 28th International Conference on Software Engineering, pp. 182–191. ACM, New York (2006)
6. Abraham, R., Erwig, M.: Type inference for spreadsheets. In: Bossi, A., Maher, M.J. (eds.) Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, Venice, Italy, 10–12 July 2006, pp. 73–84. ACM (2006)

7. Abraham, R., Erwig, M.: Goaldebug: a spreadsheet debugger for end users. In: ICSE 2007: Proceedings of the 29th International Conference on Software Engineering, pp. 251–260. IEEE Computer Society, Washington, DC (2007)
8. Abraham, R., Erwig, M.: Mutation operators for spreadsheets. IEEE Trans. Softw. Eng. **35**(1), 94–108 (2009)
9. Abraham, R., Erwig, M., Kollmansberger, S., Seifert, E.: Visual specifications of correct spreadsheets. In: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2005, pp. 189–196. IEEE Computer Society (2005)
10. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison Wesley, Reading (1986)
11. Alhajj, R.: Extracting the extended entity-relationship model from a legacy relational database. Inf. Syst. **28**(6), 597–618 (2003)
12. Alves, T.L., Silva, P.F., Visser, J.: Constraint-aware schema transformation. Electron. Notes Theor. Comput. Sci. **290**, 3–18 (2012)
13. Bricklin, D.: VisiCalc: Information from its creators, Dan Bricklin and Bob Frankston. http://www.bricklin.com/visicalc.htm. Accessed 5 Dec 2013
14. Bruins, E.: On Plimpton 322. Pythagorean numbers in Babylonian mathematics. Koninklijke Nederlandse Akademie van Wetenschappen **52**, 629–632 (1949)
15. Burnett, M., Cook, C., Pendse, O., Rothermel, G., Summet, J., Wallace, C.: End-user software engineering with assertions in the spreadsheet paradigm. In: Proceedings of the 25th International Conference on Software Engineering, ICSE 2003, pp. 93–103. IEEE Computer Society (2003)
16. Campbell-Kelly, M., Croarken, M., Flood, R., Robson, E.: The History of Mathematical Tables: From Sumer to Spreadsheets. Oxford University Press, Oxford (2003)
17. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6), 377–387 (1970)
18. Cunha, A., Oliveira, J.N., Visser, J.: Type-safe two-level data transformation. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 284–299. Springer, Heidelberg (2006)
19. Cunha, J., Erwig, M., Saraiva, J.: Automatically inferring classsheet models from spreadsheets. In: IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2010, pp. 93–100. IEEE Computer Society (2010)
20. Cunha, J., Fernandes, J., Mendes, J., Saraiva, J.: Embedding, evolution, and validation of model-driven spreadsheets. IEEE Trans. Software Eng. **PP**(99), 1 (2014)
21. Cunha, J., Fernandes, J.P., Ribeiro, H., Saraiva, J.: Towards a catalog of spreadsheet smells. In: Murgante, B., Gervasi, O., Misra, S., Nedjah, N., Rocha, A.M.A.C., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2012, Part IV. LNCS, vol. 7336, pp. 202–216. Springer, Heidelberg (2012)
22. Cunha, J., Fernandes, J.P., Mendes, J., Martins, P., Saraiva, J.: Smellsheet detective: a tool for detecting bad smells in spreadsheets. In: Proceedings of the 2012 IEEE Symposium on Visual Languages and Human-Centric Computing, VLHCC 2012, pp. 243–244. IEEE Computer Society, Washington, DC (2012)
23. Cunha, J., Fernandes, J.P., Mendes, J., Saraiva, J.: Extension and implementation of ClassSheet models. In: Proceedings of the 2012 IEEE Symposium on Visual Languages and Human-Centric Computing, VLHCC 2012, pp. 19–22. IEEE Computer Society (2012)
24. Cunha, J., Fernandes, J.P., Mendes, J., Saraiva, J.: MDSheet: a framework for model-driven spreadsheet engineering. In: Proceedings of the 34th International Conference on Software Engineering, ICSE 2012, pp. 1412–1415. ACM (2012)

25. Cunha, J., Fernandes, J.P., Saraiva, J.: From relational ClassSheets to UML+OCL. In: Proceedings of the Software Engineering Track at the 27th Annual ACM Symposium on Applied Computing, pp. 1151–1158. ACM (2012)
26. Cunha, J., Mendes, J., Fernandes, J.P., Saraiva, J.: Embedding and evolution of spreadsheet models in spreadsheet systems. In: Proceedings of the 2011 IEEE Symposium on Visual Languages and Human-Centric Computing, VLHCC 2011, pp. 186–201. IEEE (2011)
27. Cunha, J., Saraiva, J., Visser, J.: Model-based programming environments for spreadsheets. Sci. Comput. Program. (SCP) **96**, 254–275 (2014)
28. Cunha, J., Visser, J., Alves, T., Saraiva, J.: Type-safe evolution of spreadsheets. In: Giannakopoulou, D., Orejas, F. (eds.) FASE 2011. LNCS, vol. 6603, pp. 186–201. Springer, Heidelberg (2011)
29. Engels, G., Erwig, M.: ClassSheets: automatic generation of spreadsheet applications from object-oriented specifications. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 124–133. ACM (2005)
30. Erdweg, S., et al.: The state of the art in language workbenches. In: Erwig, M., Paige, R.F., Van Wyk, E. (eds.) SLE 2013. LNCS, vol. 8225, pp. 197–217. Springer, Heidelberg (2013)
31. Erwig, M.: Software engineering for spreadsheets. IEEE Softw. **29**(5), 25–30 (2009)
32. Erwig, M., Abraham, R., Cooperstein, I., Kollmansberger, S.: Automatic generation and maintenance of correct spreadsheets. In: Proceedings of the 27th International Conference on Software Engineering, pp. 136–145. ACM (2005)
33. Erwig, M., Abraham, R., Kollmansberger, S., Cooperstein, I.: Gencel: a program generator for correct spreadsheets. J. Funct. Program. **16**(3), 293–325 (2006)
34. Erwig, M., Burnett, M.: Adding apples and oranges. In: Adsul, B., Ramakrishnan, C.R. (eds.) PADL 2002. LNCS, vol. 2257, pp. 173–191. Springer, Heidelberg (2002)
35. Fisher II, M., Cao, M., Rothermel, G., Cook, C., Burnett, M.: Automated test case generation for spreadsheets. In: Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), pp. 141–154. ACM Press, New York, 19–25 May 2002
36. Fisher II, M., Rothermel, G., Brown, D., Cao, M., Cook, C., Burnett, M.: Integrating automated test generation into the WYSIWYT spreadsheet testing methdology. ACM Trans. Softw. Eng. Methodol. **15**(2), 150–194 (2006)
37. Fisher II, M., Rothermel, G., Creelan, T., Burnett, M.: Scaling a dataflow testing methodology to the multiparadigm world of commercial spreadsheets. In: Proceedings of the 17th IEEE International Symposium on Software Reliability Engineering, Raleigh, NC, USA, pp. 13–22, November 2006
38. Gibbons, J.: Functional programming for domain-specific languages. In: Zsok, V. (ed.) Central European Functional Programming - Summer School on Domain-Specific Languages, July 2013
39. Hermans, F., Pinzger, M., van Deursen, A.: Automatically extracting class diagrams from spreadsheets. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 52–75. Springer, Heidelberg (2010)
40. Hermans, F., Pinzger, M., van Deursen, A.: Supporting professional spreadsheet users by generating leveled dataflow diagrams. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, pp. 451–460. ACM (2011)
41. Hermans, F., Pinzger, M., van Deursen, A.: Detecting and visualizing interworksheet smells in spreadsheets. In: Proceedings of the 2012 International Conference on Software Engineering, ICSE 2012, pp. 441–451. IEEE Press (2012)

42. Hermans, F., Pinzger, M., van Deursen, A.: Detecting code smells in spreadsheet formulas. In: ICSM, pp. 409–418 (2012)
43. Hinze, R., Löh, A., Oliveira, B.C.S.: "Scrap your boilerplate" reloaded. In: Hagiya, M. (ed.) FLOPS 2006. LNCS, vol. 3945, pp. 13–29. Springer, Heidelberg (2006)
44. Hudak, P.: Building domain-specific embedded languages. ACM Comput. Surv. **28**(4es), 196 (1996)
45. Jones, S.P., Blackwell, A., Burnett, M.: A user-centred approach to functions in excel. In: Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, pp. 165–176. ACM (2003)
46. Kankuzi, B., Sajaniemi, J.: An empirical study of spreadsheet authors' mental models in explaining and debugging tasks. In: 2013 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2013, pp. 15–18 (2013)
47. Kuiper, M., Saraiva, J.: Lrc - a generator for incremental language-oriented tools. In: Koskimies, K. (ed.) CC 1998. LNCS, vol. 1383, pp. 298–301. Springer, Heidelberg (1998)
48. Lämmel, R., Visser, J.: A *Strafunski* application letter. In: Dahl, V. (ed.) PADL 2003. LNCS, vol. 2562, pp. 357–375. Springer, Heidelberg (2002)
49. Lämmel, R., Saraiva, J., Visser, J. (eds.): GTTSE 2005. LNCS, vol. 4143. Springer, Heidelberg (2006)
50. Luckey, M., Erwig, M., Engels, G.: Systematic evolution of model-based spreadsheet applications. J. Vis. Lang. Comput. **23**(5), 267–286 (2012)
51. Maier, D.: The Theory of Relational Databases. Computer Science Press, Rockville (1983)
52. Morgan, C., Gardiner, P.: Data refinement by calculation. Acta Inform. **27**, 481–503 (1990)
53. Nardi, B.A.: A Small Matter of Programming: Perspectives on End User Computing, 1st edn. MIT Press, Cambridge (1993)
54. Oliveira, J.: A reification calculus for model-oriented software specification. Form. Asp. Comput. **2**(1), 1–23 (1990)
55. Oliveira, J.N.: Transforming data by calculation. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) Generative and Transformational Techniques in Software Engineering II. LNCS, vol. 5235, pp. 134–195. Springer, Heidelberg (2008)
56. Panko, R.R.: What we know about spreadsheet errors. J. End User Comput. (Special issue on Scaling Up End User Development) **10**(2), 15–21 (1998)
57. Panko, R.R.: Spreadsheet errors: what we know. what we think we can do. In: Proceedings of the European Spreadsheet Risks Interest Group (EuSpRIG) (2000)
58. Panko, R.R.: Facing the problem of spreadsheet errors. Decis. Line **37**(5), 8–10 (2006)
59. Panko, R.R., Aurigemma, S.: Revising the panko-halverson taxonomy of spreadsheet errors. Decis. Support Syst. **49**(2), 235–244 (2010)
60. Panko, R.R., Ordway, N.: Sarbanes-Oxley: What About all the Spreadsheets? CoRR abs/0804.0797 (2008)
61. Peyton Jones, S., Washburn, G., Weirich, S.: Wobbly types: type inference for generalised algebraic data types. Technical report, MS-CIS-05-26, University of Pennsylvania, July 2004
62. Powell, S.G., Baker, K.R., Lawson, B.: A critical review of the literature on spreadsheet errors. Decis. Support Syst. **46**(1), 128–138 (2008)
63. Rajalingham, K., Chadwick, D.R., Knight, B.: Classification of spreadsheet errors. In: Proceedings of the 2001 European Spreadsheet Risks Interest Group, EuSpRIG 2001, Amsterdam (2001)

64. Reinhart, C.M., Rogoff, K.S.: Growth in a time of debt. Am. Econ. Rev. **100**(2), 573–578 (2010)
65. Robson, E.: Neither Sherlock Holmes nor Babylon: a reassessment of Plimpton 322. Historia Mathematica **28**(3), 167–206 (2001)
66. Rothermel, G., Burnett, M., Li, L., Sheretov, A.: A methodology for testing spreadsheets. ACM Trans. Softw. Eng. Methodol. **10**, 110–147 (2001)
67. Ruthruff, J., Creswick, E., Burnett, M., Cook, C., Prabhakararao, S., Fisher II, M., Main, M.: End-user software visualizations for fault localization. In: Proceedings of the ACM Symposium on Software Visualization, San Diego, CA, USA, pp. 123–132, June 2003
68. Scaffidi, C., Shaw, M., Myers, B.: Estimating the numbers of end users and end user programmers. In: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 207–214 (2005)
69. Stevens, P., Whittle, J., Booch, G. (eds.): UML 2003. LNCS, vol. 2863. Springer, Heidelberg (2003)
70. Swierstra, D., Azero, P., Saraiva, J.: Designing and implementing combinator languages. In: Swierstra, S.D., Oliveira, J.N. (eds.) AFP 1998. LNCS, vol. 1608, pp. 150–206. Springer, Heidelberg (1999)
71. Ullman, J.D., Widom, J.: A First Course in Database Systems. Prentice Hall, Upper Saddle River (1997)
72. Ullman, J.: Principles of Database and Knowledge-Base Systems, vol. I. Computer Science Press, Rockville (1988)
73. Visser, E.: A survey of strategies in rule-based program transformation systems. J. Symbolic Comput. **40**, 831–873 (2005)
74. Visser, J., Saraiva, J.: Tutorial on strategic programming across programming paradigms. In: 8th Brazilian Symposium on Programming Languages, Niteroi, Brazil, May 2004