Delft University of Technology
Software Engineering Research Group
Technical Report Series

Automatically Extracting Class Diagrams from Spreadsheets

Felienne Hermans, Martin Pinzger and Arie van Deursen

Report TUD-SERG-2010-013

TUD-SERG-2010-013

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
`http://www.se.ewi.tudelft.nl/techreports/`

For more information about the Software Engineering Research Group:
`http://www.se.ewi.tudelft.nl/`

Note: Accepted for publication in the Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP 2010), *Lecture Notes in Computer Science*, Springer-Verlag, 2010.

Automatically Extracting Class Diagrams from Spreadsheets

Felienne Hermans, Martin Pinzger, and Arie van Deursen

Delft University of Technology
{f.f.j.hermans,m.pinzger,arie.vandeursen}@tudelft.nl

Abstract. The use of spreadsheets to capture information is widespread in industry. Spreadsheets can thus be a wealthy source of domain information. We propose to automatically extract this information and transform it into class diagrams. The resulting class diagram can be used by software engineers to understand, refine, or re-implement the spreadsheet's functionality. To enable the transformation into class diagrams we create a library of common spreadsheet usage patterns. These patterns are localized in the spreadsheet using a two- dimensional parsing algorithm. The resulting parse tree is transformed and enriched with information from the library. We evaluate our approach on the spreadsheets from the Euses Spreadsheet Corpus by comparing a subset of the generated class diagrams with reference class diagrams created manually.

1 Introduction

To design and implement a software system a high degree of familiarity with the domain of the software is needed. We conjecture that a significant portion of this domain knowledge is already available in digital form. In particular spreadsheets, which are widely used for many professional tasks, are likely to contain a wealth of implicit knowledge of the underlying application domain. It is the purpose of this paper to make this knowledge explicit.

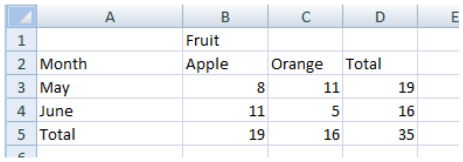
Spreadsheets were introduced in the early 1980's with the first spreadsheet tool called VisiCalc. This tool was then followed by SuperCalc and Lotus 123 and later on by Excel which currently is one of the most prominent spreadsheet tools. Since their introduction, spreadsheets are heavily used in industry. A study from the year 2005 shows about 23 million American workers use spreadsheets, which amounts to about 30% of the workforce [24].

Spreadsheets can be a rich source of information concerning the structure of the underlying domain. They contain groups of data, computations over these groups, and data dependencies between them. In our approach, we will attempt to make this structure explicit, by representing it as a class diagram. Groups of data are turned into classes, formula's into methods, and data dependencies into associations. The resulting class diagram can be used by software engineers to understand, refine, or re-implement the spreadsheet's functionality.

The research community noticed the importance of spreadsheets and devoted considerable research effort to them. This research mainly aims at two directions: 1) the localizations of errors within existing spreadsheets [1–5, 18] and 2) the development of guidelines on how to create well-structured and maintainable spreadsheets [10, 14, 15, 21, 22]. Both directions share the goal of improving spreadsheet quality, which is necessary because the current state of spreadsheet use leads to numerous problems as described in several papers, most notably in the work of Panko [20].

While some guidelines for designing spreadsheets and algorithms for detecting errors in spreadsheets exist, the elicitation of the domain information stored in spreadsheets for developing software systems has, to the best of our knowledge, not been addressed, yet (See Section 10 for a discussion of the most directly related literature).

To illustrate our approach, we will use the example presented in Figure 1, taken from Abraham and Erwig [1].



	A	B	C	D	E
1		Fruit			
2	Month	Apple	Orange	Total	
3	May	8	11	19	
4	June	11	5	16	
5	Total	19	16	35	

Fig. 1: Fruit example taken from [1].

This spreadsheet is used to list the number of different fruits (i.e., apples and oranges) that have been harvested in May and June. It also provides functions to calculate the total numbers per fruit, per month, and a function for the calculation of the overall number of harvested fruits. The structure of this spreadsheet is a common pattern that occurs in many spreadsheets. Taking a closer look at this spreadsheet, the information it contains could be represented by the class diagram shown in Figure 2.

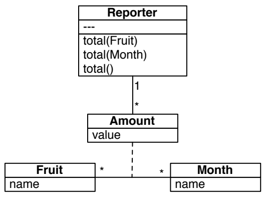


Fig. 2: Class diagram extracted from the fruit example.

For the extraction of this class diagram, we first identified the two classes *Fruit* and *Month*, with instances *Apple* and *Orange* and *May* and *June* respectively. The two classes are linked with each other by the cells B3 to C4 that specify the amount of fruits (instances of class *Fruit*) for each instance of the class *Month*. This link is represented by the association class *Amount* with an attribute *value*. Furthermore the spreadsheet

2

contains operations to calculate the *Total* per fruit, per month, and the overall total number of fruits. These operations can be provided by a *Reporter* class that we added to the class diagram. The resulting class diagram contains the core design elements to represent this spreadsheet and might be used by a developer, for example, to design a simple fruit-statistic application, or to reason about (errors in) the structure of the spreadsheet.

In this paper we focus on the automation of the extraction of such class diagrams from spreadsheets. We propose a systematic approach, called Gyro, which is supported by a tool capable of analyzing Microsoft Excel sheets. Gyro transforms spreadsheets into class diagrams automatically by exploiting commonality in spreadsheets, like the pattern in Figure 1. To that end we create a library containing common spreadsheet patterns, inspired by both related work in the field of spreadsheet design and analysis of a range of existing spreadsheets. These patterns are located within the spreadsheet using a combination of parsing and pattern matching algorithms. Each pattern in the library is associated with a mapping to a class diagram.

In order to evaluate our approach we made use of the Euses Spreadsheet Corpus [11]. This corpus contains over 4000 real world spreadsheets from domains such as finance, biology, and education. In our evaluation we demonstrate that our patterns can be found in around 40% of the spreadsheets. Furthermore we provide a systematic comparison of the generated class diagrams for a random selection of 50 spreadsheets for which we manually derived class diagrams.

The remainder of this paper is structured as follows: Section 2 introduces the necessary background information on modeling spreadsheets and two-dimensional language theory. The Gyro approach is presented in Section 3 with details of the parsing and transformation described in the Sections 4, 5 and 6. Section 7 gives a brief description of the current implementation of the Gyro prototype. The evaluation of the approach is presented in Section 8. The results are discussed in Section 9 followed by Section 10 that presents an overview of the work in the field of spreadsheets. The conclusions can be found in Section 11.

2 Background

Before presenting our Gyro approach for recognizing spreadsheet patterns, we provide a brief survey of the preliminaries we build upon. These originate from the realm of spreadsheet testing and analysis [1, 18], as well as from the domain of two-dimensional languages [12].

2.1 Cell types

Most papers on spreadsheet analysis distinguish between different cell types [1, 18]. Abraham and Erwig [1] for instance identifies header, footer, data and filler cells. Mittermeir and Clermont [18] on the other hand defines empty cells, formulas and constant values. We mostly follow the approach of the former, but we replace filler cells by empty cells. We do so because we use patterns to identify structure in a spreadsheet. Therefore we are not interested in filler cells, which usually occur between patterns. With this, the following basic cell types are recognized by our approach:

Label A cell that only contains text, giving information about other cells (called *header* in [1])
Data A cell filled with data
Formula A cell containing a calculation over other cells (called *footer* in [1])
Empty An empty cell

We prefer the terms *label* and *formula* over *header* and *footer*, because the latter have some kind of intrinsic meaning concerning their position. We want to be able to freely define any pattern, including some with 'footers' on top. To determine the type of a cell, we use a simple strategy, that basically follows the approach of [2]. This algorithm is described in Section 4.3.

2.2 Pattern languages for two-dimensional languages

To define patterns over spreadsheets we use of existing notations from the theory of *two-dimensional languages* [12] which is a generalization of the standard theory of regular languages and finite automata.

Let Σ be a finite alphabet. Then we define:

Definition 1 *A two-dimensional* pattern *over* Σ *is a two-dimensional array of elements of* Σ.

Definition 2 *The set of all two-dimensional patterns over* Σ *is denoted by* Σ^{**}. *A* two-dimensional language *over* Σ *is a subset of* Σ^{**}.

Given a pattern p over an alphabet Σ, let $l_1(p)$ denote the number of rows of p and $l_2(p)$ denote the number of columns of p. The pair $\langle l_1(p), l_2(p)\rangle$ is called the *size* of p. Furthermore, if $0 \leq i < l_1(p)$ and $0 \leq j < l_2(p)$ then $p(i,j)$ denotes the symbol $\in \Sigma$ on position (i,j). The pattern with size $\langle 0,0\rangle$ is called the empty pattern and is denoted with λ. Pictures of the size $\langle 0,n\rangle$ or $\langle n,0\rangle$ with $n > 0$ are not defined.

Next we define concatenation operations used to combine patterns. Let p be a pattern over Σ of size $\langle m,n\rangle$ and q be a pattern over Σ' of size $\langle m',n'\rangle$. We first define the rectangle we can obtain by putting q to the right of p, assuming p and q have the same number of rows, resulting in a rectangle of size $\langle m = m', n+n'\rangle$

Definition 3 *The* column concatenation *of p and q (denoted by* $p \varobar q$*) is a partial operation, only defined if* $m = m'$, *is a pattern over* $\Sigma \cup \Sigma'$ *given by*

$$(p \varobar q)(i,j) = \begin{cases} p(i,j) & \text{if } j \leq n \\ q(i,j-n) & \text{otherwise} \end{cases}$$

Similarly, we define how we can position q directly below p if p and q have the same number of columns, resulting in a rectangle of size $\langle m+m', n = n'\rangle$

Definition 4 *The* row concatenation *of p and q (denoted by* $p \ominus q$*) is a partial operation, only defined if* $n = n'$, *is a pattern over* $\Sigma \cup \Sigma'$ *given by*

$$(p \ominus q)(i,j) = \begin{cases} p(i,j) & \text{if } i \leq m \\ q(i-m,j) & \text{otherwise} \end{cases}$$

4

We will refer to these two operations as the *catenation operations*. Catenation operations of p and the empty picture λ are always defined and λ is the neutral element for both catenation operations. The catenation operators can be extended to define concatenations between two-dimensional languages.

Definition 5 *Let L_1, L_2 be two-dimensional languages over alphabets Σ_1 and Σ_2 respectively, the* column concatenation *of L_1 and L_2 is a language over $\Sigma_1 \cup \Sigma_2$ denoted by $L_1 \varobar L_2$ is defined by*

$$L_1 \varobar L_2 = \{p \varobar q | p \in L_1 \wedge q \in L_2\}$$

Similarly the row concatenation *of L_1 and L_2 is a language over $\Sigma_1 \cup \Sigma_2$ denoted by $L_1 \ominus L_2$ is defined by*

$$L_1 \ominus L_2 = \{p \ominus q | p \in L_1 \wedge q \in L_2\}$$

Definition 6 *Let L be a pattern language. The* column closure *of L (denoted by $L^{*\varobar}$) is defined as*

$$L^{*\varobar} = \bigcup_{i \geq 1} L^{i\varobar}$$

where $L^{1\varobar} = L$ and $L^{n\varobar} = L \varobar L^{(n-1)\varobar}$. Similarly, the row closure *of L (denoted by $L^{*\ominus}$) is defined as*

$$L^{*\ominus} = \bigcup_{i \geq 1} L^{i\ominus}$$

where $L^{1\ominus} = L$ and $L^{n\ominus} = L \ominus L^{(n-1)\ominus}$.

We will refer to these two operations as the *closure operations*. With respect to priorities we define that closure operations bind stronger than catenation operations.

2.3 Pattern grammars

To describe common spreadsheet patterns, we make use of *pattern grammars*. Pattern grammars are a two-dimensional generalization of ordinary grammars. This generalization is based on the observation that a production rule of the form $S \rightarrow ab$ actually means that S may be replaced by a followed by b. In regular rewriting, the 'followed by' can only occur in one direction, so this is not expressed in the grammar. To define production rules in two dimensions, we use two symbols from two-dimensional language theory that express direction, $\ominus$ and $\varobar$ and their closure operations $^{*\ominus}$ and $^{*\varobar}$

Definition 7 *The set of all two-dimensional* pattern languages *over Σ is a subset of Σ^{**} called $\mathcal{L}(\Sigma)$ is inductively defined by:*

$$\begin{gathered}
\lambda \in \mathcal{P}(\Sigma) \\
a \in \mathcal{P}(\Sigma) \text{ , if } a \in \Sigma \\
L^{*\ominus} \in \mathcal{P}(\Sigma) \text{ , if } L \in \mathcal{L}(\Sigma) \\
L^{*\varobar} \in \mathcal{P}(\Sigma) \text{ , if } L \in \mathcal{L}(\Sigma) \\
L_1 \ominus L_2 \in \mathcal{P}(\Sigma) \text{ , if } L_1, L_2 \in \mathcal{L}(\Sigma) \\
L_1 \varobar L_2 \in \mathcal{P}(\Sigma) \text{ , if } L_1, L_2 \in \mathcal{L}(\Sigma)
\end{gathered}$$

5

To avoid ambiguity we use the convention that closure operations bind stronger than catenation operations.

Definition 8 *Just as a normal grammar, a* pattern grammar *G is defined as a quadruple*

$$G = (V, T, S, P)$$

where V is a finite set of non-terminals, T is a finite set of terminal symbols, $S \in V$ is a special symbol called the start symbol, and P is a finite set of productions.

Productions are tuples (v, p) of a non-terminal v and a pattern p, denoted as $v \rightarrow p$. v is also indicated with *lefthand side*, whereas p is called *righthand side*. Since we only allow non-terminals on the lefthand side, this is a context free grammar. The pattern grammars in the paper will always consist of the basic cell types, thus the alphabet of terminals is always equal to $\{Label, Empty, Formula, Data\}$, therefore we will omit T in definitions of grammars. Unless indicated otherwise, *Pattern* is the start symbol S of any grammar in this paper.

3 The Gyro Approach to Spreadsheet Reverse Engineering

The goal of this paper is to distill class diagrams from spreadsheets. To that end we propose the Gyro approach, in which typical spreadsheet usage patterns can be specified, automatically recognized and transformed into class diagrams.

When investigating the way people use spreadsheets, we noticed that there are some common ways in which people represent information in spreadsheets. Typically data that concerns the same topic is found grouped together in rows or columns separated by empty cells. These *spreadsheet patterns* are found in all kinds of spreadsheets, independent of the business area the spreadsheet originates from. We exploit this commonality by introducing a library of common spreadsheet structures. The transformation into class diagrams is done in two steps, as shown in Figure 3.

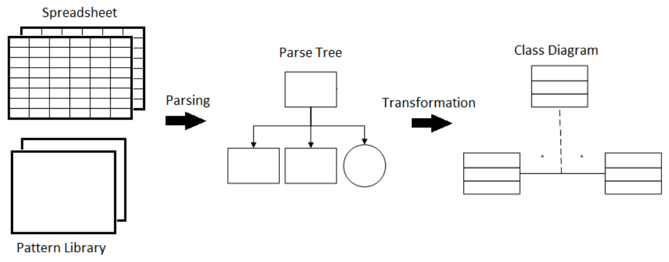


Fig. 3: Overview of the Gyro approach showing the two basic steps Parsing and Transformation to transform spreadsheets into class diagrams

We start by localizing patterns from the library in the spreadsheet, by using a two-dimensional parsing algorithm. If a pattern is found the result of this algorithm is a parse

6

tree. Each of the patterns in the library contains additional information that defines how the parse tree is transformed into a class diagram. This transformation represents the second step of our approach. The parsing is explained in more detail in Section 4 and Section 5 describes the transformation step.

The use of a library of patterns was greatly motivated by the need for flexible information extraction. We do not believe the current library contains all pattern grammars needed. So when we encounter a common spreadsheet that is not part of our library, we can add it to the library. Gyro is then able to recognize it immediately, without adaptation of the implementation. The patterns in the library are based both on existing work on spreadsheet design patterns [14, 21, 22] and on the analysis of patterns encountered in the Euses Corpus [11].

4 Pattern recognition

4.1 Overview

In order to identify regions in a spreadsheet that adhere to a given pattern, we follow the pattern recognition approach outlined in Figure 4. First, we identify rectangles in the spreadsheet that are filled with Data, Formula or Label cells, by using an algorithm to determine bounding boxes (Section 4.2). In Figure 1 for instance, this bounding box is given by the cells A1 $\times$ D5. Because we assume occurrences of pattern are separated by empty cells, we evaluate these rectangles as possible occurrences of patterns. Next each cell is assigned one of the basic cell types: `Data`, `Formula`, `Label` or `Empty` (Section 4.3).

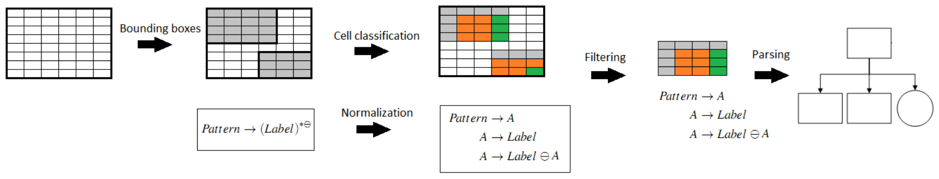


Fig. 4: Overview of the pattern recognition process

In parallel, the grammar is normalized to a form that only contains catenation operations, in order to simplify the parsing step (Section 4.4). Then a filtering step is applied, in which we check the type of the left-most upper-most corner of the bounding box. We determine which of the patterns in the library have this symbol as a possible first terminal (Section 4.5) We only start the parsing algorithm if this is the case. The parsing algorithm is an adaption of standard recursive descent parsing [6] adjusted to handle two-dimensional grammars (Section 4.6).

It is possible several patterns are applicable for processing the cells of a bounding box. In such cases the algorithm returns the set of all matched patterns.

4.2 Finding bounding boxes

A bounding box is defined as the smallest rectangle containing a connected group of cells of type `Data`, `Label` or `Formula`. Two cells are connected if they touch each other horizontally, vertically or diagonally. To find such a bounding box, we apply the following strategy: Find the left-most upper-most non-empty cell that is not yet contained within a bounding box. The initial bounding is set to contain only this cell. Next this bounding box is expanded until the size remains stable. Expanding is done by inspecting all cells that connect to the bounding box. If one of these cells is non empty, the bounding box is enlarged to include this cell. In Figure 4 the identified bounding boxes are marked grey.

4.3 Cell classification

To distinguish between cell types, we use the cell classification strategy described by Abraham and Erwig [1]. This algorithm starts with identifying all cells containing a formula and mark them as type `Formula` (green cells in Figure 4). Next we look at the content of the formula's. Cells that are referred to a formula are marked `Data`, unless they also contain a formula (orange cells in Figure 4). In that case they are marked as `Formula` as well. Remaining cells that are empty are identified as `Empty` (white cells in Figure 4); all others are recognized as a `Label` (grey cells in Figure 4).

4.4 Normalization

To simplify the parsing algorithm, we assume that the pattern under consideration only consists of $\ominus$ and $\oplus$ symbols. Therefore, we first transform the pattern to this form, that we call *catenation normal form*. Every possible pattern has an equivalent pattern in catenation normal form. To obtain this normal form row or column concatenation closures are replaced by right recursion. For instance

$$Pattern \rightarrow (Label \oplus Data)^{*\ominus}$$

becomes

$$\begin{aligned} Pattern &\rightarrow A \\ A &\rightarrow Label \oplus Data \\ (A &\rightarrow Label \oplus Data) \ominus A \end{aligned}$$

This strategy is applied repeatedly until the grammar does not contain any closure symbol.

8

4.5 Filtering

In two-dimensional pattern matching filtering is a widely used approach [7, 8, 26]. Filtering reduces the problem of finding a pattern P in an array T to finding a substring p in string t, where a detected match of p in t corresponds to a possible match of P in T. We use this idea by calculating all possible first terminals of a given pattern grammar. Next we determine whether there is a detected bounding box that has this symbol in its upper-left corner. We only start parsing the given pattern for these bounding boxes.

4.6 Parsing

To determine whether a given bounding box complies with a given pattern Q, we apply a recursive descent parsing strategy with some small modifications to handle two-dimensional structures.

Algorithm 1 provides an outline of our approach. This procedure takes a bounding box B and a pattern grammar G as its parameters. We begin with the start symbol of the grammar and expand it until the first symbol is a terminal. Expanding means replacing the left-hand side of a production rule by a corresponding right-hand side. (Algorithm 1, lines 7-12) If a non-terminal occurs as the left-hand side of multiple production rules, all of them are evaluated. If a terminal is found, we examine whether this is the expected terminal. If it is, the parsing continues, otherwise it fails. (Algorithm 1, lines 14-19)

This process requires some administration, for which we introduce a datatype Position, containing an x- and y-coordinate, representing a position in the spreadsheet, and a string T that has to be parsed at that position. The set S represents all Positions that still have to be evaluated. At the start of Algorithm 1, S is initialized to contain only one Position with coordinates (i, j) and string X. (i, j) is the upper left corner of the bounding box B and X is the start symbol of grammar G. The set T represents all successfully parsed patterns, so if parsing succeeds for a given pattern at a given position, this position is added to the set T.

The evaluation of a Position P is done in the body of the While-loop on lines 6-21 and works as follows. If P starts with a non-terminal, say Y, a new Position is added to S for every possible right-hand side r belonging to Y. That new Position has the same coordinates as P, and contains string T, where the first occurrence of Y is replaced by right-hand side r. Since S now contains all possible scenarios from the parsing of P, we can remove P from S. Evaluation continues with the remainder of set S.

If P starts with a terminal, say t, the actual parsing happens. (Lines 14-19) We determine if the symbol at position (x, y) is equal to t. If that is not the case, parsing for this particular position fails, and P is removed from S. If (x, y) is equal to t, t is removed from T. Since terminals are always followed by a catenation symbol, the first symbol of T is a catenation symbol, say c. The cursor is then moved according to c. If this is a column catenation symbol, the cursor moves to the right, if it is a row concatenation, the cursor moves downward. This moving is aware of the size of the bounding box and will fail if the cursor is about to go out of bounds. After that the evaluation continues with the modified P.

Algorithm 1 Two-dimensional Parsing(BoundingBox B, Grammar G)

```
1: $Position\ \ P \leftarrow (B.i, B.j, G.X)$
2: $Set\ \ T \leftarrow \emptyset$
3: $Set\ \ S \leftarrow \{P\}$
4: while $S \neq \emptyset$ do
5:     $P \leftarrow$ a position from $S$
6:     while $P.T \neq$ ”” do
7:         if FirstSymbol($P.T$) is non-terminal Y then
8:             for all Productions Y $\rightarrow r$ do
9:                 Create new Position $P_l$,
10:                    with $x = P.x$, $y = P.y$ and $T = r \cdot Rest(P.T)$
11:                Add $P_l$ to $S$
12:            end for
13:        else
14:            if FirstSymbol($P.T$) == B(P.x,P.y) then
15:                Remove First Symbol of P.T
16:                Move Cursor according to FirstSymbol(P.T)
17:            else
18:                Remove $P$ from $S$ {Parsing fails for this position}
19:            end if
20:        end if
21:    end while
22:    Add $P$ to $T$
23:    Remove $P$ from $S$
24: end while
```

5 From patterns to class diagrams

5.1 Using Annotations

Given a pattern and a spreadsheet, the two-dimensional parsing algorithm just described can identify rectangles in the spreadsheet that match the pattern. The result is a *parse tree*, representing the full content of the rectangle, as well as the hierarchy of productions applied to match the rectangle to the pattern.

In order to turn this parse tree into a class diagram, our primary interest is in the *structure* of the spreadsheet, not in the actual cell contents. Therefore, our next step consists of identifying those nodes in the parse tree that can help to reveal this structure. We do so by offering the possibility to add *annotations* to pattern definitions, which will subsequently guide a transformation of the parse tree into class diagram.

Using annotations to turn a parse tree into a class diagram is done in two steps, as depicted in Figure 5. First, the annotations are used to *prune* the parse tree into a *representation tree* only containing relevant nodes. Next, this representation tree is *enriched* so that a meaningful class diagram can emerge.

To see annotations in action, consider the simple spreadsheet shown in Figure 6. In this spreadsheet, one can recognize a class “Customer”, with fields “Surname” and

10

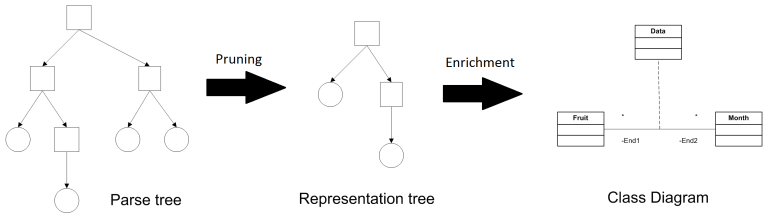
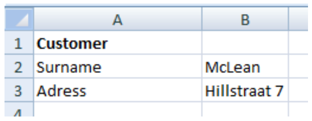


Fig. 5: Overview of the Transformation step to transform a parse tree into a class diagram



	A	B
1	**Customer**	
2	Surname	McLean
3	Adress	Hillstraat 7
4		

Fig. 6: Simple Customer spreadsheet

"Address". Thus, we define a pattern that can recognize spreadsheets like this one:

$$\begin{aligned} &G: \\ Pattern &\rightarrow Headerrow \ominus (Datarow^{*\ominus}) \\ Headerrow &\rightarrow \texttt{Label} \obar \texttt{Empty} \\ Datarow &\rightarrow \texttt{Label} \obar \texttt{Data} \end{aligned}$$

In this pattern definition, the classname can be obtained from the headerrow, and the field names from the data rows. Thus, we add *annotations* to capture exactly this:

$$\begin{aligned} &G: \\ Pattern : class &\rightarrow Headerrow \ominus (Datarow^{*\ominus}) \\ Headerrow &\rightarrow \texttt{Label} : name \obar \texttt{Empty} \\ Datarow &\rightarrow \texttt{Label} : field \obar \texttt{Data} \end{aligned}$$

Here we see that the `Label` of a header row represents the name of the *class*, and that the `Label` of a data row represents the *field* name.

Annotations are permitted on both terminals and non-terminals. For terminals they are an indication that the content of the cell contains relevant information (such as the name of a field). For non-terminals they are an indication that the non-terminal in question should be kept in the representation tree. Note that an annotation for the root is not required: Hence the result of the transformation can be either a tree or a forest.

Algorithm 2 Tree transformation

```
1: Remove all non-annotated leaves
2: Remove all non-annotated nodes without annotated descendants
3: Remove all non-annotated nodes, their (annotated) children become children of their lowest
   annotated ancestor
```

The annotations can be used to prune the parse tree as described in Algorithm 2. The original parse tree, with annotations marked in grey, is depicted in Figure 7; the corresponding pruned representation tree is shown in Figure 8.

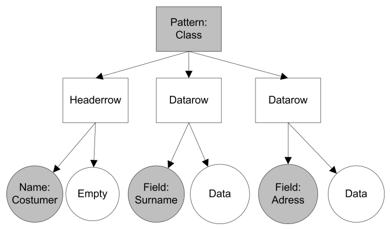


Fig. 7: Parse tree generated for the Customer spreadsheet

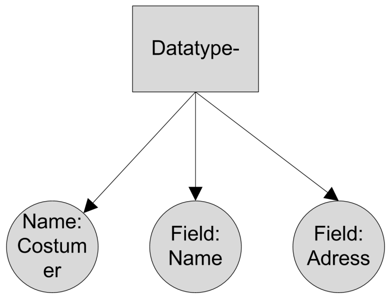


Fig. 8: Representation tree after the transformation of the Customer parse tree

5.2 Class Diagrams

In this paper, the output of the transformation step are class diagrams. Therefore, the annotations that can be used to define the transformation represent the basic building blocks of class diagrams: *class*, *name*, *field*, and *method*. Since the latter three are properties of a Class, they can be used as annotations of terminals, and Class itself occurs as annotation of non-terminals. Referring to the Customer example the class diagram contains a class `Customer` with two fields `Surname` and `Adress`.

5.3 Enrichment

The transformation step results in one or more classes. In most cases there is a relation between the different classes. In the second part of the annotation we can describe the relation between the resulting trees. The following relations can be defined in the pattern:

Association(C_1, C_2, m_1, m_2, C_3) This defines an association between two classes, C_1 and C_2. Optionally, we can define multiplicities m_1 and m_2 for this association. The last argument is again a class and represents an association class of this association.

Merge(C_1,C_2) The operation merges two classes into one class containing all fields and methods of both classes. If fields or methods with equal names are encountered both will appear in the new class. To distinguish between them their original class name will be appended to the method of field name. The Merge-operation is useful if class information is scattered around the spreadsheet and can not be easily captured within one production rule.

Reference(C_1,C_2) A reference is used in the creation of the class diagram, but will not appear in the class diagram itself. A reference between two classes is added when information about the names of fields and methods of C_1 can be found in C_2. This is used when information from the spreadsheet has to be used in multiple classes.

We will see examples of the use of these relation declarations in the next section.

6 Spreadsheet patterns

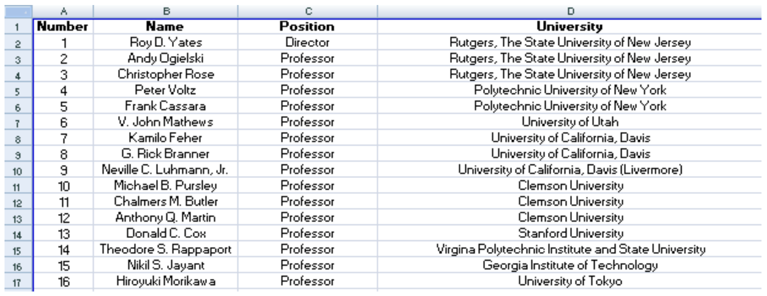
By inspecting the Euses spreadsheet corpus [11], and looking at related work in spreadsheet design [14, 15, 21, 22] we have identified a number of reoccurring patterns in spreadsheets into a *pattern library*. In this section, we describe the syntax of a selection of the most interesting spreadsheet design patterns in this library.

6.1 Simple class

The simplest pattern in the library is a list of instances as shown in Figure 9. The column headers provide the names of the fields, as captured in the following pattern.

$$\begin{aligned} Pattern : class &\rightarrow X^{*⦶} \\ X &\rightarrow \texttt{Label} : Field \ominus \texttt{Data}^{*\ominus} \end{aligned}$$

Note that the spreadsheet content provides no clue about the name of the class. The class diagram that corresponds to this pattern is shown in Figure 10.



	A	B	C	D
1	**Number**	**Name**	**Position**	**University**
2	1	Roy D. Yates	Director	Rutgers, The State University of New Jersey
3	2	Andy Ogielski	Professor	Rutgers, The State University of New Jersey
4	3	Christopher Rose	Professor	Rutgers, The State University of New Jersey
5	4	Peter Voltz	Professor	Polytechnic University of New York
6	5	Frank Cassara	Professor	Polytechnic University of New York
7	6	V. John Mathews	Professor	University of Utah
8	7	Kamilo Feher	Professor	University of California, Davis
9	8	G. Rick Branner	Professor	University of California, Davis
10	9	Neville C. Luhmann, Jr.	Professor	University of California, Davis (Livermore)
11	10	Michael B. Pursley	Professor	Clemson University
12	11	Chalmers M. Butler	Professor	Clemson University
13	12	Anthony Q. Martin	Professor	Clemson University
14	13	Donald C. Cox	Professor	Stanford University
15	14	Theodore S. Rappaport	Professor	Virgina Polytechnic Institute and State University
16	15	Nikil S. Jayant	Professor	Georgia Institute of Technology
17	16	Hiroyuki Morikawa	Professor	University of Tokyo

Fig. 9: Simple class spreadsheet pattern

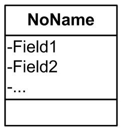


Fig. 10: Class diagram extracted from a Simple class spreadsheet pattern

6.2 Simple class including name

If there is a row above a simple class pattern with only one label, we assume this is the name of this pattern, as described by the second pattern in the library.

$$\begin{gathered} Pattern : class \rightarrow \texttt{Label} : name \obar \texttt{Empty}^{*\obar} \ominus X^{*\obar} \\ X \rightarrow \texttt{Label} : Field \ominus Data^{*\ominus} \end{gathered}$$

6.3 Simple class including methods

If one of the columns contains formula's, this column is likely to represent a method of the class. To include this, we add the following production rule to the simple class pattern (with or without class name).

$$X \rightarrow \texttt{Label} : method \ominus \texttt{Formula}^{*\ominus}$$

6.4 Aggregation

If there is a formula below a simple class, this represents a calculation over all instances, which we catch in a Reporter Class that references all the instances. For each Formula

14

we encounter, we introduce a nameless method in the reporter class.

$$\begin{aligned} Pattern &\rightarrow Simple \ominus Reporter \\ Simple : class &\rightarrow \texttt{Label} \obar \texttt{Empty}^{*\obar} \ominus X^{*\obar} \\ X &\rightarrow \texttt{Label} : field \ominus \texttt{Data}^{*\ominus} \\ X &\rightarrow \texttt{Label} : method \ominus \texttt{Formula}^{*\ominus} \\ Reporter : class &\rightarrow \texttt{Label} \obar \texttt{Formula}^{*\ominus} : method \end{aligned}$$

The methods of the Reporter class are empty in this case, but they correspond one on one to the names of the fields of the simple class. Therefore a Reference clause is added, in the enrichment step the names will be copied to the Reporter class. The relation between the simple class and the reporter class is defined by the Association clause.

$$\begin{gathered} Reference(Reporter, Simple) \\ Association(Simple, Reporter, *, 1) \end{gathered}$$

All of the above patterns can also occur vertically, rotated 90 degrees. Figure 6 shows an example of a spreadsheet in which the rotated version of the "Simple class" pattern is applied. Our library also contains the vertical variants of all above patterns.

6.5 Associated data

The final pattern in our library concerns two different classes, with data associated to both of them. This pattern represents two classes with an association between the two classes that contains the data.

$$\begin{aligned} Pattern &\rightarrow C_1 \obar (C_2 \ominus C_3) \\ C_1 : class &\rightarrow \texttt{Empty} \ominus \texttt{Label}^{*\ominus} \\ C_2 : class &\rightarrow \texttt{Label} \obar \texttt{Empty}^{*\obar} \\ C_3 : class &\rightarrow (\texttt{Label} \ominus \texttt{Data}^{*\ominus})^{*\obar} \end{aligned}$$

The relation between the classes is defined by the following clause.

$$Association(C_1, C_2, *, *, C_3)$$

Furthermore, there could also exist aggregations over this data, like in the fruit example in Figure 1. In that case we add an association between a Reporter class containing the aggregation methods and the association class, as shown in Figure 2. We model this in the representation tree as an Association with methods. To recognize this the following pattern is included in the library. This pattern also occurs in one direction only, in that case either D_1 or D_2 is omitted.

$$\begin{aligned}
Pattern &\rightarrow (C_1 \oplus (C_2 \ominus C_3) \oplus D_1) \ominus D_2 \\
C_1 : class &\rightarrow \texttt{Empty} \ominus \texttt{Label}^{*\ominus} \\
C_2 : class &\rightarrow \texttt{Label} \oplus \texttt{Empty}^{*\oplus} \\
C_3 : class &\rightarrow (\texttt{Label} \ominus \texttt{Data}^{*\ominus})^{*\oplus} \\
D_1 : class &\rightarrow \texttt{Label} : method \ominus \texttt{Formula}^{*\ominus} \\
D_2 : class &\rightarrow \texttt{Label} : method \oplus \texttt{Formula}^{*\oplus}
\end{aligned}$$

In this case, the classes D_1 and D_2 both represent the Reporter Class, but it is not possible to catch them within one production rule because of the structure of the spreadsheet. Therefore, we need a Merge-clause in this case. Furthermore one more association needs to be defined, between the reporter class and the association class.

$$\begin{aligned}
Merge(D_1, D_2) \\
Association(C_1, C_2, *, *, C_3) \\
Association(C_3, D_1, *, 1)
\end{aligned}$$

7 Implementation

The approach for extracting class diagrams from spreadsheets as described in the previous sections has been implemented in the Gyro tool suite,[1] targeting Microsoft Excel spreadsheets. Gyro users can specify the directory they want to analyze. Gyro loads all .xls and .xlsx files from that directory and ignores other files. Furthermore the user can specify in which directory the patterns can be found. Patterns are just plain text files containing a pattern grammar. Options of the current implementation include the coloring of a spreadsheet representing the basic cell classification, the search for patterns within spreadsheets, the visualization of the parse tree and the pruned parse tree, and the full transformation into class diagrams.

Gyro is subject to continuous development; at the moment we are in the process of enriching the Gyro user interface, and providing a web interface in which a spreadsheet can be simply uploaded, relevant spreadsheet patterns can be selected, and spreadsheets can be analyzed "as a service."

Gyro is implemented in C#.net using Visual Studio 2010, beta 1. We use of the Microsoft SQL Server Modeling platform (formerly "Oslo") and its MGrammar language to specify the grammar for our pattern definitions.

8 Evaluation

We evaluated the strength of our approach by testing the Gyro approach on the Euses Spreadsheet Corpus [11]. The evaluation was twofold, first we tested the quality of the

[1] Gyro can be downloaded from `http://www.st.ewi.tudelft.nl/~hermans/Gyro`.

16

Table 1: Number and percentage of spreadsheets of the Euses Spreadsheet Corpus that can be processed with the current Gyro pattern library

Type	Number of sheets	Pattern found	Success percentage
Cs101	10	4	40.0%
Database	726	334	46.0%
Filby	65	31	47.7%
Financial	904	334	36.9%
Forms3	34	14	41.2%
Grades	764	307	40.2 %
Homework	804	375	46.7%
Inventory	895	125	14.0%
Jackson	21	7	33.3%
Modeling	692	334	48.3%
Personal	7	6	85.7%

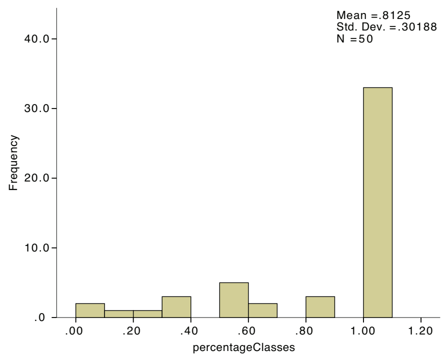
patterns, by determining how often the chosen patterns occur in the Spreadsheet Corpus. This way we can check whether the patterns we chose are really frequently used. Secondly, for the patterns that were found, we checked whether the generated class-diagram is a faithful representation of the underlying domain. This second evaluation was done by comparing generated class diagrams to class diagrams that were manually created.

8.1 The data set

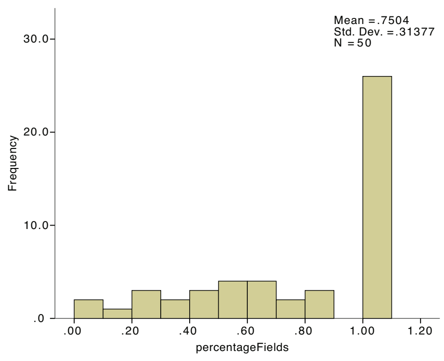
The Euses Spreadsheet Corpus is a set of spreadsheets created to help researchers to evaluate methodologies and tools for creating and maintaining spreadsheets. It consists of 4498 unique spreadsheet files, divided into 11 categories varying from financial spreadsheets to educational ones. Many papers on spreadsheet analysis use the Corpus to test their methodologies and algorithms, among which are [2] and [9].

8.2 Quality of chosen patterns

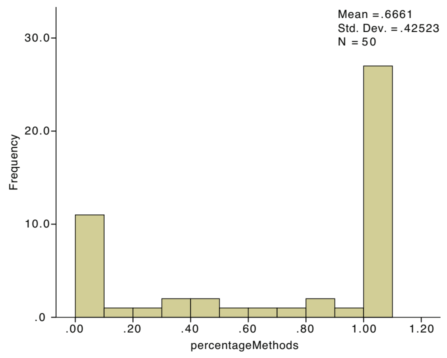
The first part of the evaluation focusses on measuring the number of spreadsheets that can be processed with our pattern library. For this we applied Gyro to the Corpus and counted the number of worksheets which in which at least one pattern could be applied. A worksheet is an individual sheet within a spreadsheet. We ignored protected, empty and invisible worksheets, as well as worksheets containing a VBA-macro. The results of this experiment can be found in Table 1. The results indicate that the patterns we chose are indeed quite common. There are 4 categories with a score in the 45-50% range, 3 in the 40-45% range, 3 in the <40% range and there is one (small) category scoring 85%. We noticed that spreadsheets originating from the inventory and financial categories score lower than spreadsheets from the other categories. The inventory category mainly consists of *empty* forms, spreadsheets in which the data still has to be filled in. Since our approach focusses on spreadsheets filled with data, the algorithm logically performs less on these inventory-sheets. The lower score on financial spreadsheets is



(a) Matched classes



(b) Matched fields



(c) Matched methods

Fig. 11: Histogram of correctly matched classes (a), fields (b), and methods (c)

probably caused by the complexity of financial spreadsheets in general. The lack of financial knowledge of the authors of this paper could also play a role. Since we are no financial experts, we do not have knowledge of common financial structures that could occur within spreadsheets, making it more difficult to design suitable patterns for this category.

8.3 Quality of mapping

The second evaluation measures the quality of extracted class diagrams. For this we randomly selected 50 spreadsheets from the Euses Spreadsheet Corpus in which one of the Gyro patterns occurs. We divided these 50 sheets among the three authors. For these spreadsheets, each author created a class diagram by hand, by looking at the structure and content of the spreadsheet. We refer to these class diagrams as *reference class diagrams*. They were created without looking at the generated class diagrams. In this evaluation we compared the generated class diagrams to the reference class diagrams and counted the number of matched classes, fields and methods.

Figures 11a, 11b and 11c depict the results of this experiment.[2] In the majority of the spreadsheets (32), Gyro found all classes and in about half of the cases all fields and methods were also extracted correctly. In the most cases in which the right pattern was selected, all methods and fields were correct. Considering the methods, we see that there is a significant number of spreadsheets in which no methods were found. This is mainly due to the fact that values in the spreadsheet can have a numerical relation that did not result from a formula. This typically occurs when data is imported into a spreadsheet from another software system where the value was calculated. A human quickly notices this relationship and decides that a column represents a method. Since Gyro only takes into account the cell types, it does not recognize this implicit relationship. We keep this as a point for future work.

Furthermore, we divided the results into four categories to get an overall view on the quality of extracted class diagrams. These categories are:

Perfect All classes, fields and methods correspond
Structure OK All classes are present, but their names, fields or methods are missing or incorrect
Classes missing Some classes are present, but others are missing
Useless The generated diagram does correspond to the the reference diagram at all

This results of this evaluation are listed in Table 2. In 20 cases Gyro produced exactly the same class diagram as the one created by the authors. In 13 cases the structure was correct. There were 6 cases in which the result generated by Gyro was classified useless. We believe these results are promising since Gyro performs just as well as a human on a large portion of the spreadsheets. However there are also some generated class diagrams that do not represent the underlying domain. These spreadsheets contained difficult calculation structure the Gyro toolkit is not able to process yet, like financial calculations or use a very specific layout that does not occur in the pattern library.

Table 2: Overall view on the quality of extracted class diagrams

Number of sheets	Perfect	Structure OK	Classes missing	Useless
50	20	13	11	6

9 Discussion

The current implementation of Gyro, while still a prototype, enables software engineers to derive domain knowledge, in the form of a class diagram, from a collection of spreadsheets. In this section, we discuss a variety of issues that affect the applicability and suitability of the proposed approach.

[2] Selected spreadsheets and reference class diagrams can be found at `http://www.st.ewi.tudelft.nl/~hermans/Gyro`

9.1 Spreadsheet limitations

There are some spreadsheet concepts the current implementation can not handle properly. Most importantly Gyro uses only one spreadsheet as input for the detection algorithms. There often exist multiple spreadsheet files with a similar structure within a company. Imagine a spreadsheet form to record the data of an order. There will probably be one file or worksheet for every order, all having the same structure. The performance of our methods could be improved by applying the pattern matching techniques described in this paper to multiple instances. This approach has several benefits. A higher degree of certainty could be achieved about the type of a cell. If a cell contains the same value in multiple spreadsheet files, it is very likely to be a label. There is also the benefit of gathering more information making it possible to do statistical analysis on the results. Furthermore, the current cell classification of only four cell types is quite coarse-grained. Improvements could for instance be made by refining the `Data` type into `Text` data and `Number` data.

9.2 Class diagram limitations

The most important class diagram feature that is currently not supported by Gyro is the inheritance relationship between classes. Inheritance is a difficult design concept per se and is typically not modeled in spreadsheets explicitly. This is also the main reason why Gyro can not extract it directly from spreadsheets using its pattern library. However, there exist approaches that can derive inheritance relationships by normalizing class diagrams. Such an approach is, for example, the FUN-algorithm [19], which we plan to integrate into Gyro.

9.3 Beyond class diagrams

In the current paper we only focussed on creating class diagrams, but the information contained in spreadsheets could also be represented in a different format. Possibilities include generating a database scheme, generating General Purpose Language code or generating more specific code, like WebDSL [13] code. The extracted domain information could also be used to create a domain-specific language tailored towards a particular business domain, since important domain concepts are present in the output of our algorithm.

We also envision the use of our algorithms to support the migration of a spreadsheet-based style of working to one where a commercial ERP (Enterprise Resource Planning) or CRM (Customer Relationship Management) solution is used. This requires understanding to what extent the data and business processes reflected in the spreadsheets can be mapped onto the models imposed by the ERP or CRM tools, for which we expect that our approach can deliver very useful information.

9.4 Dealing with spreadsheets errors

As we mentioned in the introduction, spreadsheets are hardly ever free of errors. This could raise the question how our approach deals with errors in spreadsheets. We believe errors mostly occur within formula's; the wrong cells might be referenced or a

20

copy-paste-error is made. Since the cells still are Formula typed in this case, this does not concern the patterns and our methods will still be able to find the right pattern. It would be interesting to compare the results of error-finding algorithms on the Euses spreadsheet with our results to gain more confidence in this assumption. Furthermore our approach allows the option to introduce a tolerance for minor errors within the pattern, so if some cells do not fit within the pattern, it can still be recognized. More research is needed however to determine a tolerance level that is neither to strict, nor too tolerant.

9.5 Meaningful identifiers

Spreadsheet do not always contain all information needed to create a perfect class diagram. Consider the spreadsheet from the introduction again, in Figure 1. In the spreadsheet the value of the numbers is not expressed. We might assume it is the amount of fruit sold in a particular month, but it could also be the price of the fruit or the number of customers that the farmer sold to.

Because not all information in the spreadsheet is named, the naming of methods and fields is not always perfect. We do not believe this is a big problem, because the class diagram still reveals the structure of the data if some names are missing or incorrect. in this case Gyro sketches the basic diagram and users can fill in the blanks.

9.6 Threats to validity

A threat to the external validity of our evaluation concerns the representativeness of the Euses Corpus spreadsheet set. This set, however, is large (over 4000 spreadsheets), and is collected from practice. Furthermore, for the manual comparison of the class diagrams we randomly picked 50 spreadsheets from this set.

With respect to internal validity, one of the threats is the fact that the reference class diagrams were only created by three people, who were also involved in the paper. The outcome might have been different if other people had created the reference diagrams, because experience, education and personal taste influence how a person creates class diagrams. This effect can be decreased by using a larger test group in future experiments. We however believe the current test group serves as a good reference group, as the persons involved all have years of experience in software engineering and modeling. Furthermore, the collection of 50 spreadsheets and accompanying class diagrams is available from our web site, allowing other researchers to challenge our reference set.

With respect to threats to reliability (repeatability), the Gyro tool, the pattern library used, the selected spreadsheets from the spreadsheet corpus and the reference set of class diagrams are available from our web site, enabling other researchers to redo our experiments.

10 Related work

Spreadsheets analysis is a subject of ongoing research. Most papers in this field focus on testing spreadsheets and certifying their correctness. Abraham and Erwig have written a series of articles on *unit* inference [1–4]. Their units form a type system based on values in the spreadsheet that is used to determine whether all cells in a column or row have the same type. Their work was of inspiration to us, but their objectives differ from ours. They focus on error finding, where we aim at extracting information. However, their *hex* and *vex* groups - similarly shaped rows and columns - inspired us in defining patterns in the library. Ahmad *et al.* [5] also created a system to annotate spreadsheets, however their approach requires users to indicate the types of fields themselves. Mittermeir and Clermont [18] investigate the possibility of structure finding, but their aim again was localizing errors by finding discrepancies within the structures found.

Another group of papers presents best practices in spreadsheet design [14, 15, 21, 22], which gave us more insight into the patterns that had to be included in our library. Fisher *et al.* [10] suggest a systematic approach to building a spreadsheet, but their methods are used to create spreadsheet from scratch, and not to analyze existing ones.

Besides the goal also the approach of existing work differs from ours. Where existing papers recognize structure bottom up, by building up their knowledge of the spreadsheet cell by cell, we apply a more top-down approach, by checking whether a spreadsheet complies with a given structure.

For the recognition of patterns our initial approach was to use techniques from two-dimensional pattern matching. However they match patterns of fixed size within an array. Although this does not suit our purpose, these algorithms did provide us with some valuable ideas. For instance, there is a class of *filter-based algorithms*, like Baker [7], Bird [8] and Takaoka-Zhu [26]. This class of algorithms is based on the reduction of two-dimensional matching to one dimension. Another useful principle we took from existing algorithms is the notion of *approximate matching*, where small differences between patterns and arrays are allowed. In this variant of matching an integer *k* is introduced that indicates how many mismatches are allowed with respect to a given distance, like for instance the Levenshtein distance [17].

Because the two-dimensional pattern matching approach was not applicable in our case, we started to investigate two-dimensional parsing. There have been some attempts to generalize parsing to two dimensions. First there is *array parsing* [23]. In this paradigm, rewriting may be applied to subarrays of the equal size. Because of that property it is never possible to generate an array from a single start symbol. Hence the use of these grammars does not solve the problem of variable size matching. Secondly, there are *matrix grammars* [25], which generate arrays in two phases, a horizontal and a vertical phase. Although they are better applicable than array grammars, matrix grammars are not able to recognize patterns that are a combination of rows and columns.

The idea to transform spreadsheets into databases has been described in [9]. Their goal is to transform spreadsheets into relational databases by using the FUN algorithm [19] to find functional dependencies within rows. Therefore their approach is limited to spreadsheet resembling non normalized databases.

Due to the use of spreadsheets as simple databases, our work also connects to the problem of Object-Relational Mapping. In particular, we attempt to map from two-dimensional relations back to object structures.

Last but not least, reverse engineering class diagrams from regular programs written in, e.g., Java has been studied extensively. An overview is provided by [16], who also

22

include a comparison with existing roundtrip engineering tools. The key problem in these approaches is to reverse engineer class associations from class implementations, which differs from our purpose of extracting class diagrams from spreadsheet logic.

11 Concluding Remarks

The goal of this paper is to underline the importance of spreadsheet analysis as a means to better understand the business domain of companies and users. To that end we have designed an approach to describe common spreadsheet design patterns, and we implemented the Gyro tool to extract the domain information automatically. The key contributions of this paper are as follows:

– A notation for expressing spreadsheet patterns, as well as two-dimensional parsing algorithm capable of recognizing these patterns (Section 4);
– A systematic approach to transform recognized patterns into class diagrams (Section 5);
– A library of frequently occurring patterns (Section 5);
– An implementation of the proposed methods and library in the Gyro system (Section 7);
– An evaluation of the proposed approach on a corpus of over 4000 spreadsheets (Section 8).

The results of our evaluation with the Euses Spreadsheet Corpus showed that Gyro can extract valuable domain information from 40% of the given spreadsheets. The evaluation further showed that extracted class diagrams are of reasonable quality—out of 50 spreadsheets 20 class diagrams were extracted perfectly, 13 contained minor flaws, in 11 cases classes were missing, and in only 6 cases the extracted class diagrams were rated as useless. This clearly underlines the potential of the Gyro approach.

We see several avenues for future research. First the description of patterns could be improved. Pattern grammars might be a convenient way of describing spreadsheet patterns for users with experience in programming and formal languages, but it is probably not that easy for users from the business domain. To make Gyro easier for this kind of users, we intend to create a visual editor for patterns. Furthermore spreadsheets do not have to be replaced by software in all cases. A possible other use for Gyro could be to aid users in creating structured spreadsheets, by offering pattern-based edit assistance, comparable to the discovery-based assistance in [9]. Finally we have several ideas to speed up the implementation of the algorithm. For instance, the filter-based part of the algorithm now only checks the left-most upper-most cell of the pattern. It might be better to look at the first row or the first column or a combination of both, to determine earlier that there is no match. The current parsing approach is recursive descent, which is known to be very inefficient in some cases. We would like to explore the possibilities of using an LR-like parsing strategy on the recognition of pattern grammars.

References

1. Robin Abraham and Martin Erwig. Header and unit inference for spreadsheets through spatial analyses. In *Proceedings of the IEEE International Symposium on Visual Languages and Human-Centric Computing(VL/HCC)*, pages 165–172, 2004.

2. Robin Abraham and Martin Erwig. Inferring templates from spreadsheets. In *Proceedings of the 28th International Conference on Software Engineering(ICSE)*, pages 182–191, New York, NY, USA, 2006. ACM.
3. Robin Abraham and Martin Erwig. Mutation operators for spreadsheets. *IEEE Transactions on Software Engineering*, 35(1):94–108, 2009.
4. Robin Abraham, Martin Erwig, and Scott Andrew. A type system based on end-user vocabulary. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 215–222, Washington, DC, USA, 2007. IEEE Computer Society.
5. Y. Ahmad, T. Antoniu, S. Goldwater, and S. Krishnamurthi. A type system for statically detecting spreadsheet errors. In *Proceedings of the IEEE International Conference on Automated Software Engineering*, pages 174–183, 2003.
6. Alfred Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
7. Theodore P. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM Journal on Computing*, 7(4):533–541, 1978.
8. R. S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6(5):168 – 170, 1977.
9. Jácome Cunha, João Saraiva, and Joost Visser. Discovery-based edit assistance for spreadsheets. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 233–237. IEEE, 2009.
10. M. Fisher, Mingming Cao, G. Rothermel, C. R. Cook, and M. M. Burnett. Automated test case generation for spreadsheets. In *Proceedings of the International Conference on Software Engineering(ICSE)*, pages 141–151, 2002.
11. M. Fisher and Gregg Rothermel. The EUSES spreadsheet corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms. In *In 1st Workshop on End-User Software Engineering*, pages 47–51, 2005.
12. Dora Giammarresi and Antonio Restivo. Two-dimensional finite state recognizability. *Fundamenta Informaticae*, 25(3):399–422, 1996.
13. Danny M. Groenewegen, Zef Hemel, Lennart C. L. Kats, and Eelco Visser. WebDSL: A domain-specific language for dynamic web applications. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 779–780, 2008.
14. Diane Janvrin and Joline Morrison. Using a structured design approach to reduce risks in end user spreadsheet development. *Information & Management*, 37(1):1–12, 2000.
15. Brian Knight, David Chadwick, and Kamalesen Rajalingham. A structured methodology for spreadsheet modelling. *Proceedings of the European Spreadsheet Risks Interest Group(EuSpRiG)*, 1:158, 2000.
16. Ralf Kollman, Petri Selonen, Eleni Stroulia, Tarja Systä, and Albert Zündorf. A study on the current state of the art in tool-supported uml-based static reverse engineering. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 22–, 2002.
17. Vladimir I. Levenshtein. On the minimal redundancy of binary error-correcting codes. *Information and Control*, 28(4):268–291, 1975.
18. R. Mittermeir and M. Clermont. Finding high-level structures in spreadsheet programs. In *Proceedings of the the Ninth Working Conference on Reverse Engineering (WCRE)*, page 221, Washington, DC, USA, 2002. IEEE Computer Society.
19. Noel Novelli and Rosine Cicchetti. Fun: An efficient algorithm for mining functional and embedded dependencies. In *Proceedings of the International Conference on Database Theory(ICDT)*, pages 189–203, 2001.
20. Raymond R. Panko. What we know about spreadsheet errors. *Journal of End User Computing*, 10(2):15–21, 1998.

21. Raymond R. Panko and Richard P. Halverson Jr. Individual and group spreadsheet design: Patterns of errors. In *Proceedings of the Hawaii International Conference on System Sciences (HICSS)*, pages 4–10, 1994.
22. Boaz Ronen, Boaz Ronen, Michael A. Palley, Michael A. Palley, Henry C. Lucas, and Henry C. Lucas. Spreadsheet analysis and design. *Communications of the ACM*, 32:84–93, 1989.
23. Azriel Rosenfeld. Array grammars. In *Graph-Grammars and Their Application to Computer Science*, volume 291 of *LNCS*, pages 67–70. Springer-Verlag, 1986.
24. Christopher Scaffidi, Mary Shaw, and Brad A. Myers. Estimating the numbers of end users and end user programmers. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 207–214, 2005.
25. G. Siromoney, R. Siromoney, and K. Krithivasan. Abstract families of matrices and picture languages. *Computer Graphics and Image Processing*, 1(3):284–307, 1972.
26. Rui Feng Zhu and Tadao Takaoka. A technique for two-dimensional pattern matching. *Commununications of the ACM*, 32(9):1110–1120, 1989.

TUD-SERG-2010-013
ISSN 1872-5392