Fakultät für Informatik
Technische Universität München

Bachelor's Thesis in Informatics

Automatic Code Generation from Spreadsheets

Jacob Zhang

Fakultät für Informatik
Technische Universität München

Bachelor's Thesis in Informatics

Automatic Code Generation from Spreadsheets

Bachelorarbeit in Informatik

Automatische Codegenerierung aus Kalkulationstabellen

Author: Jacob Zhang
Supervisor: Prof. Dr. Uwe Baumgarten
Advisor: Dr. Arnaud Fietzke
Submission Date: September 27, 2019

Statutory Declaration

I confirm that this Bachelor's thesis is my own work and I have documented all sources and material used.

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, September 27, 2019

JACOB ZHANG

Abstract

Spreadsheets are easy to use, require little training for the users and are very flexible. It is possible to model complex calculation logic within them. In this thesis, we examine ways of automatically generating code from spreadsheets, as with growing scale, they can become increasingly difficult to maintain compared to specialized software. We define a formal spreadsheet model that can be readily transformed into code, and examine a reference implementation that is able to generate code from spreadsheets.

Zusammenfassung

Tabellenkalkulationsprogramme sind leicht zu bedienen und im Einsatz sehr flexibel. Man kann in ihnen komplexe Berechnungslogik modellieren. Da jedoch die Wartbarkeit mit steigender Komplexität stark abnimmt, ist oft spezialisierte Software vorzuziehen. In dieser Arbeit wird untersucht, wie automatisch Quellcode aus Kalkulationstabellen gewonnen werden kann. Wir definieren ein formales Modell für Kalkulationstabellen, das als Basis für die Transformation in Code dient. Die darauf aufbauende Implementation ist in der Lage, automatisiert Quellcode aus Kalkulationstabellen zu generieren.

Acknowledgements

I would like to express my sincere gratitude to my advisor Dr. Arnaud Fietzke for his patient guidance, enthusiastic encouragement and useful critiques.

I would also like to thank Professor Baumgarten for supervising this thesis.

Finally, I wish to express my thanks for my friends and family for their support and encouragement. I am particularly grateful to Jonas Habel and Marcel Kollovieh for proofreading this work.

Contents

Statutory Declaration iii

Abstract iv

Acknowledgements v

1 Introduction 1
- 1.1 Related work . . . 2
- 1.2 Implementation . . . 2
- 1.3 Outline . . . 2

2 A formal spreadsheet model 3
- 2.1 Definitions and notations . . . 3
- 2.2 Formulas . . . 4
- 2.3 Spreadsheet graph . . . 5
  - 2.3.1 Graph generation . . . 6
  - 2.3.2 Cell classification . . . 7
- 2.4 Labels . . . 8
  - 2.4.1 Label classification . . . 8
  - 2.4.2 Label regions . . . 9
  - 2.4.3 Name generation . . . 13

3 Code generation 15
- 3.1 Object-oriented approach . . . 15
  - 3.1.1 Class creation . . . 17
  - 3.1.2 Topological ordering . . . 17
- 3.2 Function-based approach . . . 19
  - 3.2.1 Function generation . . . 21
  - 3.2.2 Cloning detection . . . 22

4 Implementation **24**
4.1 Code generation 24
4.2 Implementation details 25
4.2.1 Type safety 25
4.2.2 Empty cells 25
4.2.3 Ranges 26
4.2.4 Functions 27
4.2.5 Names 30
4.2.6 External references 30
4.3 User interface 31
4.3.1 Graph generation 32
4.3.2 Class generation 32
4.4 Automated testing 33

5 Conclusion and future work **34**

List of Algorithms

1 GraphGeneration 6
2 ReferencedCells 6
3 CreateLabelRegions 10
4 MergeLabelRegions 12

5 CreateClasses 17
6 TopologicalSort (Kahn's algorithm) 18
7 GenerateFunctions 21
8 StructuralEquivalence 22

List of Figures

1 Production rules for Excel formulas 4
2 Cell classification example 7
3 A grade calculator 8
4 The grade calculator with colored label regions 10
5 Region merging in action 11
6 A more complex region merging example 11
7 Merging overlapping regions in one step 13

8 An insurance plan calculator 15
9 The insurance plan calculator with formulas and cell classifications 15
10 The insurance plan calculator with Inputs 19

11 Excel's "name" feature 30
12 User interface with graph generation tab selected 31
13 User interface with class generation tab selected 32
14 Automated testing 33

1 Introduction

The use of spreadsheet software has become ubiquitous in many domains. With spreadsheets being the most commonly given example of *end-user programming* (EUP), which refers to tools that allow those who are not software developers to program computers, it can be argued that spreadsheets *are* programs and thus the most prevalent and successful programming language in the world [1].

As spreadsheets are highly flexible and there are few to none of the formal restrictions commonly found in conventional programming languages, they do not scale well and can become increasingly difficult to maintain over time. Similarly, due to the lack of abstraction mechanisms, they allow re-usability to only a very limited extent.

This is exacerbated by the fact that users commonly have little training as programmers, but still face the challenges of professional developers [1], leading to bad implementation practices that result in duplication, hard-to-follow dependency structures or even gross errors. In addition, the expressions and formulas used in a spreadsheet are not directly visible, but hidden in cells, making it difficult to get a high-level view of its logical structure, as one would by looking at a well-structured piece of code.

In this light, it is unsurprising that with growth, companies increasingly move away from spreadsheets towards specialized, modular software. However, spreadsheets can be a very useful tool in modeling business logic. As such, they can even be considered a natural intermediate step in the process of transforming informal or paper-based business procedures into integrated digital solutions. For this reason, spreadsheets are routinely included in requirement specifications of software engineering projects.

However, transforming spreadsheets to code is a manual and often complex and error-prone task, as there is no automated way to extract the calculation logic, which might have been implemented across thousands of cells.

In this thesis, we explore ways to aid the developer in that task, by formalizing and implementing a model that can be used to automatically generate structured and readable code from spreadsheets.

1.1 Related work

There are number of works that involve extracting data from spreadsheets. Most of these works, such as [2] and [3], use class-based approaches that seek to identify the relational tables within a spreadsheet.

However, in the context of this thesis we are not interested in the actual data contained in a spreadsheet, but rather the calculation logic that establishes relationships between the cells. To that end, most of the time we can ignore the layout of spreadsheets entirely and focus solely on formulas and values inside cells.

The only part in this thesis where we consider the actual layout within a spreadsheet is *name generation* (Section 2.4), where we try to infer variable names. There, we will be referring to the Wang model [4] and the work of Koci et al. in [5] and [6].

1.2 Implementation

Although the concepts developed in this thesis are abstract and can be applied to any spreadsheet, the reference implementation will deal with Microsoft Excel, the most widely used spreadsheet software.

Furthermore, the implementation is written in C# and will also generate code in C#.

1.3 Outline

In Chapter 2, we first define a formal model for spreadsheets. In Chapter 3, we examine methods to translate that model into programming concepts. Finally, we consider a reference implementation in Chapter 4.

2 A formal spreadsheet model

In this chapter, we examine a formal model for spreadsheets, which will serve as a basis for code generation.

2.1 Definitions and notations

First, we shall consider formal definitions and notational conventions of concepts used throughout this thesis.

A **cell** $c \in \mathbb{C}$ is an entity that holds data, whereas $\mathbb{C}$ denotes the set of all cells. With $\mathbb{T} = \{\text{Bool}, \text{Number}, \text{Date}, \text{Text}, \text{Unknown}\}$ and $\mathbb{V}$ as the set of all possible values a cell may contain (including ϵ, nothing), we define

- $\mathit{Value}\colon \mathbb{C} \to \mathbb{V}$ as a function that maps cells to the **value** contained inside;
- $\mathit{Type}\colon \mathbb{C} \to \mathbb{T}$ as a function that maps cells to the **type** of the value;
- $\mathit{IsEmpty}\colon \mathbb{C} \to \{\text{true}, \text{false}\}$ which returns true if the cell is empty ($\mathit{Value} = \epsilon$).

A **worksheet** $\mathcal{W} \in \mathbb{C}^{m \times n}$ is a matrix

$$\mathcal{W} = \begin{pmatrix} \mathcal{W}_{1,1} & \dots & \mathcal{W}_{m,1} \\ \vdots & \ddots & \vdots \\ \mathcal{W}_{1,n} & \dots & \mathcal{W}_{m,n} \end{pmatrix}$$

consisting of $m \cdot n$ cells in m columns and n rows. In accordance with spreadsheet conventions, the index notation of rows and columns is inverted. For readability, we assign letters to column numbers, so that $1 = A$, $2 = B$, ..., $26 = Z$, $27 = AA$, $28 = AB$, etc. As such, the notation $\mathcal{W}_{\text{C2}}$ shall refer to the same cell as $\mathcal{W}_{3,2}$. When it is clear which worksheet $\mathcal{W}$ we are referring to, we can also just write "C2".

A **region** $\mathcal{R}_{(i,j):(i',j')} \subseteq \mathcal{W}$ is a submatrix

$$\mathcal{R}_{(i,j):(i',j')} = \begin{pmatrix} \mathcal{W}_{i,j} & \dots & \mathcal{W}_{i',j} \\ \vdots & \ddots & \vdots \\ \mathcal{W}_{i,j'} & \dots & \mathcal{W}_{i',j'} \end{pmatrix} \qquad \begin{matrix} i, i' \in [1, m],\ j, j' \in [1, n] \\ i \leq i',\ j \leq j' \end{matrix}$$

that contains all cells within a defined rectangular area of a worksheet.

2.2 Formulas

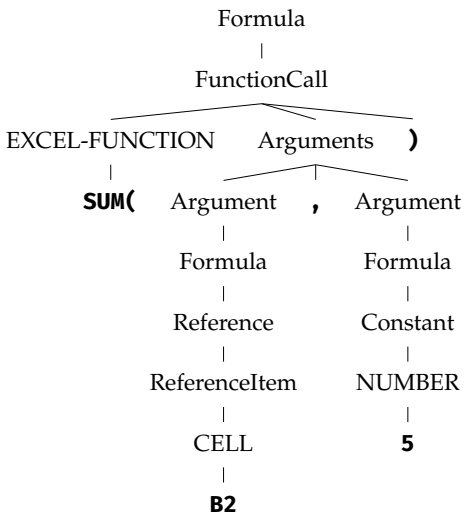
Formulas are deterministic calculation procedures contained within each cell. For example, the formula `123` or `=123` will simply yield the value 123, while `=A1+A2` will calculate $Value(\mathcal{W}_{A1}) + Value(\mathcal{W}_{A2})$.

With G as the grammar of spreadsheet formulas, and $L(G)$ as the language of G, we define $Formula\colon \mathbb{C} \to L(G) \cup \{\epsilon\}$ as the function that maps cells to their formula. A spreadsheet cell's value is always obtained by evaluation of its formula: $Value(c) := \mathsf{Evaluate}(Formula(c))$.

Defining G is not a trivial task, as spreadsheet formulas have become immensely powerful over the years. For instance, the official grammar used in Excel provided by Microsoft in Augmented Backus–Naur Form (ABNF) spans over 25 pages and contains hundreds of production rules. [7]

Aivaloglou et al. [8] took on the task to simplify Excel's formula grammar (see the production rules in Figure 1 and the list of lexical tokens in Table 1). They also provide a reference parser implementation for the grammar.

Using the grammar, the formula `SUM(B2,5)` would for example be parsed into



⟨*Start*⟩ ::= ⟨*Constant*⟩
| '=' ⟨*Formula*⟩
| '{=' ⟨*Formula*⟩ '}'

⟨*Formula*⟩ ::= ⟨*Constant*⟩
| ⟨*Reference*⟩
| ⟨*FunctionCall*⟩
| '(' ⟨*Formula*⟩ ')'
| ⟨*ConstantArray*⟩
| RESERVED-NAME

⟨*Constant*⟩ ::= NUMBER | STRING | BOOL | ERROR

⟨*FunctionCall*⟩ ::= EXCEL-FUNCTION ⟨*Arguments*⟩ ')'
| ⟨*UnOpPrefix*⟩ ⟨*Formula*⟩
| ⟨*Formula*⟩ '%'
| ⟨*Formula*⟩ ⟨*BinOp*⟩ ⟨*Formula*⟩

⟨*UnOpPrefix*⟩ ::= '+' | '−'

⟨*BinOp*⟩ ::= '+' | '−' | '*' | '/' | '^'
| '<' | '>' | '=' | '<=' | '>=' | '<>'

⟨*Arguments*⟩ ::= ⟨*Argument*⟩ { ',' ⟨*Argument*⟩ } | ϵ

⟨*Argument*⟩ ::= ⟨*Formula*⟩ | ϵ

⟨*Reference*⟩ ::= ⟨*ReferenceItem*⟩
| ⟨*RefFunctionCall*⟩
| '(' ⟨*Reference*⟩ ')'
| ⟨*Prefix*⟩ ⟨*ReferenceItem*⟩
| FILE '!' DDECALL

⟨*RefFunctionCall*⟩ ::= ⟨*Union*⟩
| ⟨*RefFunctionName*⟩ ⟨*Arguments*⟩ ')'
| ⟨*Reference*⟩ ':' ⟨*Reference*⟩
| ⟨*Reference*⟩ '␣' ⟨*Reference*⟩

⟨*ReferenceItem*⟩ ::= CELL
| ⟨*NamedRange*⟩
| VERTICAL-RANGE
| HORIZONTAL-RANGE
| UDF ⟨*Arguments*⟩ ')'
| ERROR-REF

⟨*Prefix*⟩ ::= SHEET
| FILE SHEET
| FILE '!'
| QUOTED-FILE-SHEET
| MULTIPLE-SHEETS
| FILE MULTIPLE-SHEETS

⟨*RefFunctionName*⟩ ::= REF-FUNCTION
| REF-FUNCTION-COND

⟨*NamedRange*⟩ ::= NR | NR-PREFIXED

⟨*Union*⟩ ::= '(' ⟨*Reference*⟩ { ',' ⟨*Reference*⟩ } ')'

⟨*ConstantArray*⟩ ::= '{' ⟨*ArrayColumns*⟩ '}'

⟨*ArrayColumns*⟩ ::= ⟨*ArrayRows*⟩ { ';' ⟨*ArrayRows*⟩ }

⟨*ArrayRows*⟩ ::= ⟨*ArrayConst*⟩ { ',' ⟨*ArrayConst*⟩ }

⟨*ArrayConst*⟩ ::= ⟨*Constant*⟩
| ⟨*UnOpPrefix*⟩ NUMBER
| ERROR-REF

Figure 1: Production rules for Excel formulas by Aivaloglou et al. [8]

Token Name	Description	Contents
BOOL	Boolean literal	TRUE | FALSE
CELL	Cell reference	$? [A-Z]+ $? [0-9]+
DDECALL	Dynamic Data Exchange link	' ([^ '] | '')+ '
ERROR	Error literal	#NULL! | #DIV/0! | #VALUE! | #NAME? | #NUM! | #N/A
ERROR-REF	Reference error literal	#REF!
EXCEL-FUNCTION	Excel built-in function	(Any entry from the function list) \(
FILE	External file reference	\[[0-9]+ \]
HORIZONTAL-RANGE	Range of rows	$? [0-9]+ : $? [0-9]+
MULTIPLE-SHEETS	Multiple sheet references	((□2+ : □2+)| ' (□3 | '')+ : (□3 | '')+ ')) !
NR	Named range	[A-Z_\\][A-Z0-9_.□1]*
NR-PREFIXED	Named range which starts with a string that could be another token	(TRUE | FALSE | [A-Z]+[0-9]+) [A-Z0-9_.□1]+
NUMBER	An integer, floating point or scientific notation number literal	[0-9]+ ,? [0-9]* (e [0-9]+)?
QUOTED-FILE-SHEET	A file reference within single quotes	'\[[0-9]+ \] (□3 | '')+ '!
REF-FUNCTION	Excel built-in reference function	(INDEX | OFFSET | INDIRECT)\(
REF-FUNCTION-COND	Excel built-in conditional ref function	(IF | CHOOSE)\(
RESERVED-NAME	An Excel reserved name	_xlnm\. [A-Z_]+
SHEET	The name of a worksheet	(□2+ | ' (□3 | '')+ ') !
STRING	String literal	" ([^ "] | "")* "
UDF	User Defined Function	(_xll\.)? [A-Z_\][A-Z0-9_\\.□1]* (
VERTICAL-RANGE	Range of columns	$? [A-Z]+ : $? [A-Z]+
Placeholder character	Placeholder for	Specification
□1	Extended characters	Non-control Unicode characters x80 and up
□2	Sheet characters	Any character except ' * [ ] \ : / ? ( ) ; { } # " = < > & + - * / ^ % , ␣
□3	Enclosed sheet characters	Any character except ' * [ ] \ : / ?

Table 1: Lexical tokens used in the grammar by Aivaloglou et al. [8]

2.3 Spreadsheet graph

In the following, we examine relationships between the cells of a worksheet, as established by references inside formulas.

For a worksheet $\mathcal{W}$, we can represent interdependencies with a **graph**

$$G_{\mathcal{W}} = (V, E), \quad V \subseteq \mathcal{W}, \quad E \subseteq \{(c_1, c_2) \mid (c_1, c_2) \in V^2\}$$

where V is a set of cells and E a binary relation that specifies directed edges. As spreadsheet formulas are not allowed to contain circular references, we require $(c, c) \notin E^+$ for

all $c \in V$ to ensure that there are no directed cycles, with E^+ denoting the transitive closure of E. This makes the graph a **directed acyclic graph (DAG)**.

2.3.1 Graph generation

To create a graph from a set of cells, we add edges to the referenced cells in each cell's formula (Algorithm 1).

Algorithm 1: GraphGeneration

Input: V, a set of cells inside $\mathcal{W}$.
Output: $G_{\mathcal{W}} = (V, E)$.

```
1 begin
2     E ← ∅
3     foreach c ∈ V do
4         E ← E ∪ {(c, c′) | c′ ∈ ReferencedCells(Parse(Formula(c)))}
5     return (V, E)
```

To obtain the referenced cells within a formula, we use the parsing library implemented by Aivaloglou et al. [8], which provides the function Parse to transform a formula string into a parse tree node. Each parse tree node is labeled with a production rule term, and may contain other parse tree nodes as children. We recursively traverse down the root node until we find the term "CELL", upon which we return the found cell (Algorithm 2).

Algorithm 2: ReferencedCells

Input: *node*, a parse tree node.
Output: C, a set of cells referenced in *node*.

```
1 begin
2     if node.Term = "CELL" then
3         return {W_node.Token}
4     return ⋃{ReferencedCells(node′) | node′ ∈ node.ChildNodes}
```

Here, *node.Term* refers the parse tree node's production rule term, while *node.Token* traverses down the tree and returns the first token found.

This rather simple algorithm does not consider advanced spreadsheets features such as ranges (e. g. `A1:C3`) or references to other worksheets (e. g. `Worksheet2!A1`). Those will be discussed in Section 4.2.

2.3.2 Cell classification

With a spreadsheet graph $G_{\mathcal{W}} = (V, E)$, we established relationships between cells, modelled as edges of the graph. Based on these relationships, we want to classify cells into *roles* they will take during code generation. For example, a cell that references other cells but is not referenced *by* other cells can be considered an *Output*. A cell that doesn't reference other cells but is referenced *by* other cells can be considered a *Constant*.

These classifications are formalized by a function

$$\mathit{Classification}\colon V \to \{\text{Constant, Intermediate, Output, None}\}$$

that maps cells in V to their respective roles. With

$$\begin{aligned} \mathit{Successors}\colon V \to \mathcal{P}(V), &\quad c \mapsto \{c' \mid (c, c') \in E\}, \\ \mathit{Predecessors}\colon V \to \mathcal{P}(V), &\quad c \mapsto \{c' \mid (c', c) \in E\}, \end{aligned}$$

classifications shall be obtained by:

$$\mathit{Classification}(c) := \begin{cases} \text{Constant} & \text{if } |\mathit{Predecessors}(c)| > 0 \wedge |\mathit{Successors}(c)| = 0 \\ \text{Intermediate} & \text{if } |\mathit{Predecessors}(c)| > 0 \wedge |\mathit{Successors}(c)| > 0 \\ \text{Output} & \text{if } |\mathit{Predecessors}(c)| = 0 \wedge |\mathit{Successors}(c)| > 0 \\ \text{None} & \text{if } |\mathit{Predecessors}(c)| = 0 \wedge |\mathit{Successors}(c)| = 0 \end{cases}$$

An example is given in Figure 2. With

$$\begin{aligned} V &= \{\text{A1}, \text{B1}, \text{A2}, \text{B2}, \text{A3}, \text{B3}\}, \\ E &= \{(\text{A2}, \text{A1}), (\text{B2}, \text{B1}), (\text{A3}, \text{A2}), (\text{A3}, \text{B2})\}, \end{aligned}$$

A1 and B1 are classified as Constants, A2 and B2 as Intermediates, and A3 as an Output.

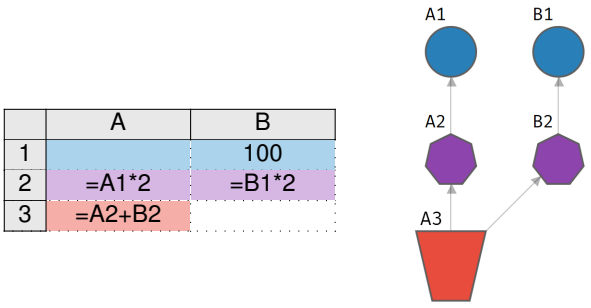


Figure 2: Cell classification example

2.4 Labels

When we generate code, we are interested in giving cells meaningful symbolic names; variable names such as "A1" or "B2" would not be very expresssive.

In spreadsheets, there are often labels assigned to rows and columns. For example, we can see in Figure 3 that column A is named "Module", column B "Assignments", column C "Examinations" and column D "Final Grade", while row 2 is named "IN0001", row 3 "IN0002" and row 4 "IN0003".

	A	B	C	D
1	**Module**	**Assignments**	**Examinations**	**Final Grade**
2	**IN0001**	80	60	=(B2+C2)/2
3	**IN0002**	100	80	=(B3+C3)/2
4	**IN0003**	95	65	=(B4+C4)/2

Figure 3: A grade calculator

Cells that serve an actual purpose in the spreadsheet's calculation logic shall be called **Data** cells. With respect to our spreadsheet model, this can be formalized by

$$c \text{ is } \textbf{Data} \iff \mathit{Classification}(c) \neq \text{None}.$$

The result of name generation shall be a function *Name* that maps Data cells to a name. In the example above, a meaningful mapping could be:

$$\begin{aligned} \mathit{Name}\colon\ & \text{B2} \mapsto \text{IN0001\_Assignments}, \\ & \text{B3} \mapsto \text{IN0002\_Assignments}, \\ & \text{B4} \mapsto \text{IN0003\_Assignments}, \\ & \text{C2} \mapsto \text{IN0001\_Examinations}, \\ & \text{C3} \mapsto \text{IN0002\_Examinations}, \\ & \text{C4} \mapsto \text{IN0003\_Examinations}, \\ & \text{D2} \mapsto \text{IN0001\_FinalGrade}, \\ & \text{D3} \mapsto \text{IN0002\_FinalGrade}, \\ & \text{D4} \mapsto \text{IN0003\_FinalGrade}. \end{aligned}$$

2.4.1 Label classification

In practice, spreadsheets are designed for humans rather than for machines. It is relatively easy for humans to interpret column and row labels, but as there are virtually no constraints in placing them, an algorithmic approach can be rather tricky.

An often cited approach in literature for modeling tables was designed by Wang in 1996 [4] and classifies labels into *Stubs*, *Stubheads* and *Boxheads*. A *Stub* contains row headings, e. g. A2, A3 and A4 in Figure 3. A *Boxhead* contains column headings, e. g. B1, C1 and D1. Finally, the *Stubhead* is the upper left cell and contains the categories in the stub, e. g. A1.

For Figure 3, classifying labels is rather straightforward, with only one stub column and boxhead row, and no gaps between labels. That may not always be the case: there may be multiple stub columns and boxhead rows, there might even be gaps inside rows and columns, or the spacing could be irregular. We therefore seek to design an algorithm that is as dynamic as possible.

Based on the Wang model, Koci et al. [5] [6] identify five "building blocks" for spreadsheet tables: *Headers*, *Attributes*, *Metadata*, *Data* and *Derived*. *Headers* correspond to *Boxheads* and *Stubheads* in the Wang model and represent column labels. *Attributes* correspond to *Stubs* in the Wang model and represent row headers. *Metadata* is additional information about the table, such as title and footnotes. *Derived* are aggregations (via formulas) of *Data*.

To classify each cell into those five building blocks, Koci et al. use machine learning methods for empirical classification [5], taking into account styling factors such as cell style and font size. This approach is able to infer entire *layouts* in spreadsheets. Our goal however, is just to generate variable names from labels. To that end, we can simplify the model.

For our purposes, *Metadata* can be ignored. Similarly, we do not need to distinguish between *Data* and *Derived*. Thus, we define three classification types: **Headers**, **Attributes** and **Data**.

2.4.2 Label regions

We now discuss an algorithm that is able to identify Headers, Attributes and Data to a reasonable degree of accuracy, taking into account our graph model and the spatial positions of the cells in relation to each other.

To achieve that, we employ the concept of a **label region**, a term introduced by Koci et al. [6], albeit with a different definition. A label region is a region $\mathcal{R}_{(i,j):(i',j')} \subseteq \mathcal{W}$ of a specific *type*, defined as

$$\textit{RegionType}\colon \mathcal{W} \to \{\text{Header, Attribute, Data}\},$$

$$\mathcal{R}_{(i,j):(i',j')} \mapsto \begin{cases} \text{Data} & \text{if } \exists c \in \mathcal{R} : \textit{Classification}(c) \neq \text{None} \\ \text{Attribute} & \text{if } j' - j > i' - i \\ \text{Header} & \text{if } j' - j \leq i' - i \end{cases}$$

Intuitively, this means that a label region is a Data region, if it contains Data cells; an Attribute region, if it has a "vertical" shape; and a Header region, if it has a "horizontal" shape.

We seek to identify label regions that are as big as possible. The grade calculator shown in Figure 3 can be decomposed into three label regions: a Header region $\mathcal{R}_{\text{A1:D1}}$, an Attribute region $\mathcal{R}_{\text{A2:A4}}$, and a Data region $\mathcal{R}_{\text{B2:D4}}$ (see Figure 4).

	A	B	C	D
1	Header	Header	Header	Header
2	Attribute	Data	Data	Data
3	Attribute	Data	Data	Data
4	Attribute	Data	Data	Data

Figure 4: The grade calculator from Figure 3 with colored label regions

The algorithm initially creates a new label region for each non-empty cell. These are then successively merged, until no regions can be merged anymore.

Algorithm 3: CreateLabelRegions

Input: $\mathcal{W}$, a worksheet.
Output: L, a list of label regions.

```
1  begin
       // Initialize with a label region for every non-empty cell
2      L ← ∅
3      for row ← 1 to W.rowCount do
4          for col ← 1 to W.columnCount do
5              if ¬IsEmpty(W_(col,row)) then
6                  L.Add(R_(col,row):(col,row))
       // Merge regions
7      for i ← 1 to L.Length do
8          for j ← 1 to L.Length do
9              if i = j then continue
               // Returns a new region R_merged and set of merged regions M to remove
10             (R_merged, M) ←MergeLabelRegions(L[i], L[j], L)
11             if |M| > 0 then
12                 L[i] ← R_merged
13                 L ← L \ M
14                 j ← j − 1 // As L[j] was merged, decrease j
15     return L
```

An example of Algorithm 3 in action is given in Figure 5, which produces an output of three regions:

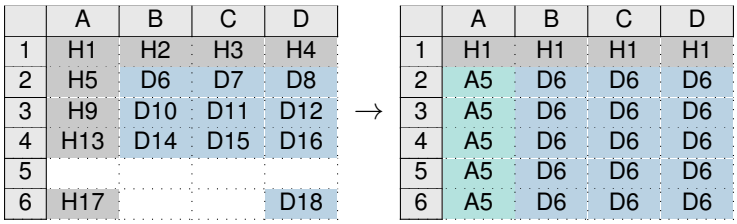


Figure 5: Region merging in action

Figure 6 shows a more complex example, a spreadsheet which contains multiple gaps:

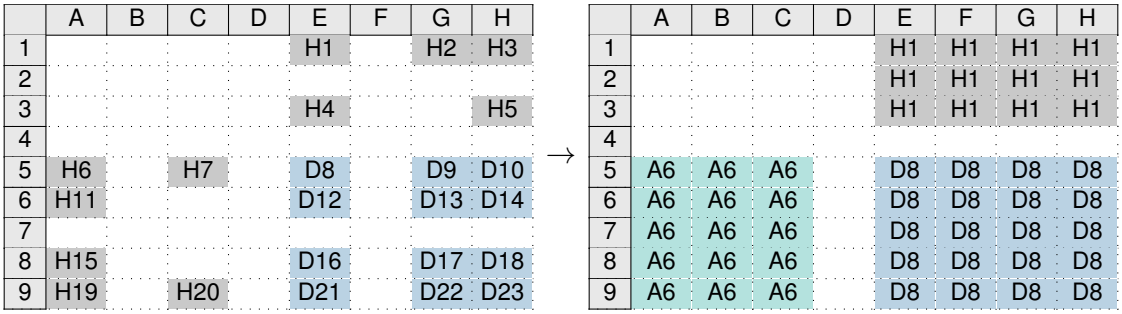


Figure 6: A more complex region merging example

With n denoting the number of non-empty cells, Algorithm 3 has a best-case run-time complexity of $\mathcal{O}(n)$ and a worst-case complexity of $\mathcal{O}(n^2)$, because L will get progressively smaller as more and more regions are getting merged.

We now examine MergeLabelRegions more closely, the function referenced in line 10 of Algorithm 3 that performs the merging of two (and possibly more) label regions.

The horizontal and vertical *distance* between two regions is defined as

$$\mathit{HorizontalDistance}(\mathcal{R}_{(i,j):(i',j')}, \mathcal{R}_{(u,v):(u',v')}) := \begin{cases} u - i' & \text{if } i' \leq u \\ u' - i & \text{if } i \leq u' \\ -1 & \text{otherwise.} \end{cases}$$

$$\mathit{VerticalDistance}(\mathcal{R}_{(i,j):(i',j')}, \mathcal{R}_{(u,v):(u',v')}) := \begin{cases} v - j' & \text{if } j' \leq v \\ j - v' & \text{if } v' \leq j \\ -1 & \text{otherwise.} \end{cases}$$

with the distance being -1 if the two regions overlap (which can not happen as a result of the algorithm).

In addition, let *HorizontalMergingRange* be a constant that specifies the maximum horizontal distance between two regions allowed to merge; conversely, *VerticalMergingRange* specifies the maximum vertical distance.

Algorithm 4: MergeLabelRegions

Input: $\mathcal{R}_1 = \mathcal{R}_{(i,j):(i',j')}$ and $\mathcal{R}_2 = \mathcal{R}_{(u,v):(u',v')}$, two label regions.
S, a set of all current label regions.
Output: $\mathcal{R}_{merged}$, a new label region.
M, a set of label regions that were merged.

```
1  begin
       // Do not allow merging of Header/Attribute regions with Data regions
2      if RegionType(R_1) = Data ⊕ RegionType(R_2) = Data then
3          return (∅, ∅)
4      if VerticalDistance(R_1, R_2) > VerticalMergingRange ∨
         HorizontalDistance(R_1, R_2) > HorizontalMergingRange then
5          return (∅, ∅)
       // Indices of new region
6      x ← min{i, u}, y ← min{j, v}
7      x' ← max{i', u'}, y' ← max{j', v'}
8      R_merged ← R_(x,y):(x',y')
9      M ← {R_1, R_2} // Set of label regions that were merged
       // Merge other regions that now lie in R_merged
10     foreach R_overlap ∈ {R' | R' ∈ S \ {R_1, R_2} ∧ R' ∩ R_merged ≠ ∅} do
           // Abort merge if a Header/Attribute region would be merged with a Data region
11         if RegionType(R_1) = Data ⊕ RegionType(R_overlap) = Data then
12             return (∅, ∅)
           // Abort merge if the overlapping region protrudes beyond the new region
13         if ∃c ∈ R_overlap : c ∉ R_merged then
14             return (∅, ∅)
           // Mark the overlapping region as merged
15         M ← M ∪ {R_overlap}
16     return (R_merged, M)
```

We disallow merging if we would merge a Header or Attribute region with a Data region (line 2) or when the vertical or horizontal distance exceeds the allowed merging range (line 4). We then create a new rectangular region that is just big enough to fit both regions inside (lines 6–8).

We now try to detect other regions that might be overlapping with the new region (line 10 onwards) and mark them as merged, if the overlapping region lies wholly inside the new region. An example of this can be seen in Figure 7, in which a third region $\mathcal{R}_{overlap}$ is merged during the merging of $\mathcal{R}_1$ and $\mathcal{R}_2$.

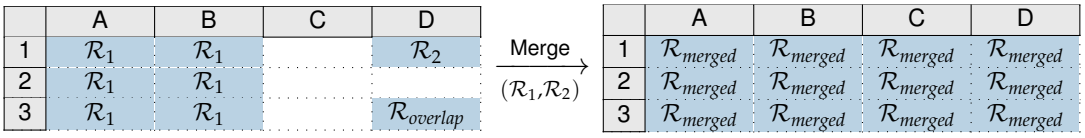


Figure 7: Merging overlapping regions in one step

2.4.3 Name generation

Having now established our Header, Attribute and Data regions, we want to infer relationships between them. Specifically, we assign Header and Attribute regions to Data regions.

We define *HeaderAssociationRange* as the maximum vertical distance a Header region can be *above* a Data region so that the Header region is associated with the Data region. Similarly, we define *AttributeAssociationRange* as the maximum horizontal distance an Attribute region can be *left* to a Data region so that the Attribute region is associated with the Data region.

Given D, a set of Data regions, and HA, a set of Header and Attribute regions, the assignment for an $\mathcal{R}_{data} \in D$ can be obtained with

$$Headers(\mathcal{R}_{data}) := \begin{array}{l}\{\mathcal{R} \mid \mathcal{R} \in HA \wedge Type(\mathcal{R}) = \text{Header} \wedge \mathcal{R} \text{ is above } \mathcal{R}_{data} \wedge \\ \quad VerticalDistance(\mathcal{R}, \mathcal{R}_{data}) \leq HeaderAssociationRange\}\end{array}$$

$$Attributes(\mathcal{R}_{data}) := \begin{array}{l}\{\mathcal{R} \mid \mathcal{R} \in HA \wedge Type(\mathcal{R}) = \text{Attribute} \wedge \mathcal{R} \text{ is left to } \mathcal{R}_{data} \wedge \\ \quad HorizontalDistance(\mathcal{R}, \mathcal{R}_{data}) \leq AttributeAssociationRange\}\end{array}$$

with predicates "is left to" and "is above" defined as

$$\mathcal{R}_{(i,j):(i',j')} \text{ is left to } \mathcal{R}_{(u,v):(u',v')} := \begin{cases}\text{true} & \text{if } i' < u \\ \text{false} & \text{otherwise.}\end{cases}$$

$$\mathcal{R}_{(i,j):(i',j')} \text{ is above } \mathcal{R}_{(u,v):(u',v')} := \begin{cases}\text{true} & \text{if } j' < v \\ \text{false} & \text{otherwise.}\end{cases}$$

We can now finally define the *Name* function mentioned in the beginning of this section to assign variable names to Data cells.

Given a cell $\mathcal{W}_{(i,j)} \in \mathcal{R}_{data}$ with $Headers = Headers(\mathcal{R}_{data})$ ordered in ascending row numbers and $Attributes = Attributes(\mathcal{R}_{data})$ ordered in ascending column numbers, a meaningful variable name can be generated with

$$\begin{aligned} Name(\mathcal{W}_{(i,j)}) = \bigcirc \{Value(\mathcal{W}_{(u,v)})\_ \mid \mathcal{W}_{(u,v)} \in \mathcal{R}, \mathcal{R} \in Attributes \wedge j = v\} \circ \\ \bigcirc \{Value(\mathcal{W}_{(u,v)})\_ \mid \mathcal{W}_{(u,v)} \in \mathcal{R}, \mathcal{R} \in Headers \wedge i = u\} \end{aligned}$$

whereas $\circ$ refers to string concatenation and $\bigcirc$ to the string concatenation of an entire set. We can ignore the last "_" at the end of the string generated by the function.

For example, in Figure 6 we obtained three label regions $\mathcal{R}_{\text{E1:H3}}$ (Header), $\mathcal{R}_{\text{A5:C9}}$ (Attribute) and $\mathcal{R}_{\text{E5:H9}}$ (Data) after setting $HorizontalMergingRange, VerticalMergingRange \geq 2$.

With $HeaderAssociationRange, AttributeAssociationRange \geq 2$, we get

$$\begin{aligned} Headers(\mathcal{R}_{\text{E5:H9}}) &= \{\mathcal{R}_{\text{E1:H3}}\}, \\ Attributes(\mathcal{R}_{\text{E5:H9}}) &= \{\mathcal{R}_{\text{A5:C9}}\}. \end{aligned}$$

Thus, we obtain:

$$\begin{aligned} Name\colon\ & \text{E5} \mapsto Value(\text{A5})\_Value(\text{B5})\_Value(\text{C5})\_Value(\text{E1})\_Value(\text{E2})\_Value(\text{E3}), \\ & \text{F5} \mapsto Value(\text{A5})\_Value(\text{B5})\_Value(\text{C5})\_Value(\text{F1})\_Value(\text{F2})\_Value(\text{F3}), \\ & \dots \\ & \text{E6} \mapsto Value(\text{A6})\_Value(\text{B6})\_Value(\text{C6})\_Value(\text{E1})\_Value(\text{E2})\_Value(\text{E3}), \\ & \dots \\ & \text{H9} \mapsto Value(\text{A9})\_Value(\text{B9})\_Value(\text{C9})\_Value(\text{H1})\_Value(\text{H2})\_Value(\text{H3}). \end{aligned}$$

In the case that both *Headers* and *Attributes* are empty, *Name* shall simply return the address of the cell, e. g. "A1".

3 Code generation

In this chapter, we examine methods to transform our graph-based spreadsheet model into actual code. Naturally, there is no *definitive* way of expressing a spreadsheet as a computer program. As such, we focus on two approaches that were developed in this thesis: an **object-oriented** approach for object-oriented programming languages, and a **function-based** approach for any imperative programming language.

3.1 Object-oriented approach

In this approach, we create a **class** for each Output.

Let us consider Figure 8, a simple spreadsheet an insurance company might use to calculate costs for two insurance plans A and B.

	A	B	C	D	E	F
1		**Plan A**	**Plan B**			**Tax rate**
2	**Base rate**	100	120		Tax 1:	15%
3	**Surcharge**	50	60		Tax 2:	5%
4					Total tax:	20%
5	**Subtotal**	150	180			
6						
7	**Total cost**	180	216			

Figure 8: An insurance plan calculator

	A	B	C	D	E	F
1		**Plan A**	**Plan B**			**Tax rate**
2	**Base rate**	100	120		Tax 1:	15%
3	**Surcharge**	50	60		Tax 2 :	5%
4					Total tax:	=F2+F3
5	**Subtotal**	=B2+B3	=C2+C3			
6						
7	**Total cost**	=B5*(1+F4)	=C5*(1+F4)			

Figure 9: The insurance plan calculator with formulas and cell classifications

Once we have established our spreadsheet graph $G_{\mathcal{W}} = (V, E)$ and classified the cells (Figure 9), it becomes clear that the spreadsheet uses the inputs *Base rate*, *Surcharge* and *Tax rate* to calculate the *total cost*.

When we generate code, we wish to structure our code in a logical way to improve readability. For the example above, we see that the calculation logic for "Plan A" and "Plan B" are largely independent from each other, save a "shared" factor *Tax rate* (F4).

A natural way to structure code in object-oriented programming are classes. We create a class for each Output. Each Constant and Intermediate that can be assigned to one and only one Output is assigned to that Output's class. Constants and Intermediates that are used in the calculation logic of multiple classes will be assigned to a new, separate static "Global" class.

With this approach, we might get the following output, a class `PlanA` for Output B7 and a class `PlanB` for Output C7:

Listing 1: The insurance plan calculator in code, object-oriented approach

```
static class Global {
   static double tax1 = 0.15;
   static double tax2 = 0.05;
   static double totalTax = tax1 + tax2;
}

class PlanA {
   double planA_BaseRate = 100;
   double planA_Surcharge = 50;

   double Calculate() {
      double planA_Subtotal = planA_BaseRate + planA_Surcharge;
      double planA_TotalCost = planA_Subtotal * (1 + Global.totalTax);
      return planA_TotalCost;
   }
}

class PlanB {
   double planB_BaseRate = 120;
   double planB_Surcharge = 60;

   double Calculate() {
      double planB_Subtotal = planB_BaseRate + planB_Surcharge;
      double planB_TotalCost = planB_Subtotal * (1 + Global.totalTax);
      return planB_TotalCost;
   }
}
```

Within each class, Constants are declared as fields. Each class contains a method `Calculate` executing the list of Intermediates before returning with the Output value.

`Global` is a static class that contains Constants and Intermediates used by at least two other classes. Calling `new PlanA().Calculate()` will return the value of Output B7 (180), while `new PlanB().Calculate()` will return the value of Output C7 (216).

3.1.1 Class creation

We now examine the process of creating classes in detail.

To assign cells to classes, we consider the transitive closure E^+ of the graph. For each cell c in V, we obtain the set of Outputs that *require* this cell with

$$OutputsDependingOn(c) := \{c' \mid Classification(c') = \text{Output} \wedge (c', c) \in E^+\}$$

We now define an algorithm to create a set of classes, given a spreadsheet graph.

Algorithm 5: CreateClasses

Input: $G_{\mathcal{W}} = (V, E)$, a spreadsheet graph.
Output: *Classes*, a set of classes.

```
1  begin
2      Classes ← ∅
3      foreach c_Output ∈ {c | c ∈ V ∧ Classification(c) = Output} do
4          classVariables ← {c | c ∈ V ∧ OutputsDependingOn(c) = {c_Output}}
5          class ← new Class(classVariables)
6          Classes ← Classes ∪ {class}
7      sharedVariables ← {c | c ∈ V ∧ |OutputsDependingOn(c)| > 1}
8      sharedClass ← new Class(sharedVariables)
9      Classes ← Classes ∪ {sharedClass}
10     return Classes
```

Algorithm 5 creates a new class for each Output (lines 3–6), with variables that are required by that Output alone. Variables that are required by multiple Outputs are assigned to a separate class (lines 7–9).

3.1.2 Topological ordering

The order of variables within a class is important. For example, in Figure 9 we are forced to evaluate B2, B3, B5, F2, F3 and F4 first before we can calculate the value in B7. An ordering like that is commonly referred to as a **topological ordering**, in which for every edge (c_1, c_2), c_1 must come after c_2 in the ordering.

We perform topological sorting on each class. In that context, V refers to cells assigned to that class, while E only includes edges between the cells in V.

An algorithm commonly used for topological sorting was described by Kahn in 1962 [9] and has a run-time linear to the number of cells plus the number of edges $\mathcal{O}(|V| + |E|)$.

Kahn's algorithm (Algorithm 6) starts with a set S of cells with no incoming edge. By the definition of a DAG, at least one such cell must exist. It then iteratively removes outgoing edges from these cells, adding cells that now have no incoming edges to S (lines 7–10). In the end, the list of edges must be empty, otherwise there is at least one cycle in the graph.

Algorithm 6: TopologicalSort (Kahn's algorithm)

Input: V, a set of cells.
$E \subseteq \{(c_1, c_2) \mid (c_1, c_2) \in V^2\}$, a set of directed edges with no cycles.
Output: L, an ordered list of cells that contains the same elements as S, with
$L.\text{IndexOf}(c_1) > L.\text{IndexOf}(c_2)\ \forall (c_1, c_2) \in E$.

```
1  begin
2      L ← ∅
       // Set of cells with no incoming edge
3      S ← {c | ∀c_parent ∈ V : (c_parent, c) ∉ E}
4      while S ≠ ∅ do
5          c ← S.RemoveOne()
6          L.Prepend(c)
7          foreach c_child ∈ {c_child | (c, c_child) ∈ E} do
8              E ← E \ {(c, c_child)}
9              if ∄c' : (c', c_child) ∈ E then
10                 S ← S ∪ {c_child}
11     if E = ∅ then
12         return L
13     else
           // Error: there is at least one cycle
```

Most of the time, there are multiple valid topological sorts for a given input. For example, when we sort the graph in Figure 9 topologically, it does not matter whether we evaluate B2 or B3 first. The sort largely depends on the implementation of RemoveOne (line 5), the manner of choosing which cell to remove from the set.

3.2 Function-based approach

In the function-based approach, we are not interested in actual values, but rather the calculation logic encapsulated within formulas. Instead of *classes*, we only use *functions*, making this approach viable for programming languages that are not objected-oriented.

The core principle of this approach is generating functions for each formula. Those functions can take parameters. To determine those parameters, we expand our cell classification model defined in Section 2.3.2.

Previously, we classified cells into Constants, Intermediates, Outputs and None. For this approach, we introduce a new classification type **Input**, which is defined as an *empty* Constant:

$$Classification(c) := \begin{cases} \text{Input} & \text{if } |Predecessors(c)| > 0 \land |Successors(c)| = 0 \land IsEmpty(c) \\ \text{Constant} & \text{if } |Predecessors(c)| > 0 \land |Successors(c)| = 0 \land \neg IsEmpty(c) \\ \text{Intermediate} & \text{if } |Predecessors(c)| > 0 \land |Successors(c)| > 0 \\ \text{Output} & \text{if } |Predecessors(c)| = 0 \land |Successors(c)| > 0 \\ \text{None} & \text{if } |Predecessors(c)| = 0 \land |Successors(c)| = 0 \end{cases}$$

With that in mind, let us consider the insurance plan calculator from the previous example again. We clear the cells which we want to have as Inputs. The modified calculator can be seen in Figure 10:

	A	B	C	D	E	F
1		**Plan A**	**Plan B**			**Tax rate**
2	**Base rate**				Tax 1:	15%
3	**Surcharge**				Tax 2 :	5%
4					Total tax:	=F2+F3
5	**Subtotal**	=B2+B3	=C2+C3			
6						
7	**Total cost**	=B5*(1+F4)	=C5*(1+F4)			

Figure 10: The insurance plan calculator with Inputs

The generated code can be divided in two disjunct "parts" V_1, V_2.

The first part includes Constants and Intermediates which do not depend on Inputs:

$$V_1 := \{c \mid c \in V \land Classification(c) \in \{\text{Constant}, \text{Intermediate}\} \land \neg DependsOnInput(c)\}$$

where

$$DependsOnInput(c) := \begin{cases} \text{true} & \text{if } \exists c' \in V : Classification(c') = \text{Input} \land (c, c') \in E^+ \\ \text{false} & \text{otherwise.} \end{cases}$$

In Figure 10, this includes Constants F2 and F3 as well as Intermediate F4.

The second part includes Intermediates and Outputs which depend on at least one Input:

$$V_2 := \{c \mid c \in V \wedge \mathit{Classification}(c) \in \{\text{Intermediate}, \text{Output}\} \wedge \mathit{DependsOnInput}(c)\}$$

In Figure 10, this includes Intermediates B5, C5 and Outputs B7, C7.

For each Intermediate and Output in V_2, we generate a function which *encapsulates* that cell's calculation logic. The result of such a generation can be seen in Listing 2:

Listing 2: The insurance plan calculator in code, function-based approach

```
// Part 1: Constants, and Intermediates not depending on Inputs
double tax1 = 0.15;
double tax2 = 0.05;
double totalTax = tax1 + tax2;

// Part 2: Outputs and Intermediates
double OutputTotalCost(double baseRate, double surcharge) {
   double subtotal = CalcSubtotal(baseRate, surcharge);
   double totalCost = CalcTotalCost(subtotal);
   return totalCost;
}

double CalcSubtotal(double baseRate, double surcharge) {
   return baseRate + surcharge;
}

double CalcTotalCost(double subtotal) {
   return subtotal * (1 + totalTax);
}
```

To calculate the total cost, we can now call `OutputTotalCost` with parameters *Base rate* and *Surcharge*.

Notice how functions `OutputTotalCost` and `CalcSubtotal` exist only once, even though we have more than one Output. This is because we can detect that the formulas in B5 and C5 as well as B7 and C7 are equivalent. This will be discussed in more detail in Section 3.2.2.

The reason why this approach is called a *function-based* approach rather than a *functional* one lies in the generation of Output methods. With each Output function, we create statements for each Intermediate involved in the calculation of that Output and sort them topologically. To make the approach fully functional and thus viable for functional programming languages, we could successively reduce the list of statements into one:

Listing 3: The insurance plan calculator in code, *functional* approach

```
...
// Part 2: Outputs and Intermediates
double OutputTotalCost(double baseRate, double surcharge) {
   return CalcTotalCost(CalcSubtotal(baseRate, surcharge));
}
...
```

However, as the function-based approach is superior in terms of legibility, especially when lots of variables are used, we abandon the functional approach.

3.2.1 Function generation

We now examine the process of generating Output and Intermediate functions. In this context, we shall represent a **function** formally by a 4-tuple

$$(Name, Parameters, ReturnType, Body)$$

where:

- $Name \in \mathbb{V}$ is the name.
- $Parameters \subsetneq \mathbb{V} \times \mathbb{T}$ is a set of tuples $(Name, Type)$.
- $ReturnType \in \mathbb{T}$ is the return type.
- $Body$ is the function's body, representing a list of statements.

To generate functions, we utilize Algorithm 7:

Algorithm 7: GenerateFunctions

Input: V_1, cells in Part 1.
V_2, cells in Part 2
Output: F, a set of functions.

begin
  $F \leftarrow \emptyset$
  foreach $c \in V_2$ **do**
    $Name \leftarrow (Classification(c) = \text{Output}$ **?** "Output" **:** "Calc" $) \circ Name(c)$
    $Parameters \leftarrow \{(Name(c'), Type(c')) \mid c' \in Successors(c) \cap V_2\}$
    $ReturnType \leftarrow Type(c)$
    $Body \leftarrow$ FormulaToCode(c)
    $F \leftarrow F \cup \{(Name, Parameters, ReturnType, Body)\}$
  return F

To determine the parameters of a function (line 5), we find the referenced cells inside the formula (which are defined as the *Successors*, see Section 2.3.2), excluding "global" constants in Part 1.

The implementation of FormulaToCode (line 7) will be described in Chapter 4.

3.2.2 Cloning detection

When we consider Figure 10, we observe that cells B5 and C5 as well as B7 and C7 contain the same calculation logic, albeit with references to different cells. In software engineering terms, this is what would be called *code cloning*.

To detect and eliminate cloning, we need to determine if two functions are **equivalent**. Formally speaking, given two functions $F_1 := (Name_1, Parameters_1, ReturnType_1, Body_1)$ and $F_2 := (Name_2, Parameters_2, ReturnType_2, Body_2)$, F_1 is said to be *equivalent* to F_2 if and only if $F_1(args) = F_2(args)$ for any set of arguments $args$.

Determining the equivalence of two arbitrary functions is known to be impossible by Rice's theorem [10] and reduction to the halting problem. There are various heuristics for approximating the equivalence test, but those are outside of the scope of this thesis.

Instead, for now we shall only test for **structural equivalence**, which checks, intuitively speaking, whether two functions "look the same". To structurally compare F_1 and F_2, we replace references to $Parameters_2$ inside $Body_2$ with references to $Parameters_1$ and check if $Body_1$ and $Body_2$ are then equal, see Algorithm 8:

Algorithm 8: StructuralEquivalence

Input: $F_1 = (Name_1, Parameters_1, ReturnType_1, Body_1)$, a function.
$F_2 = (Name_2, Parameters_2, ReturnType_2, Body_2)$, a function.
Output: true or false.

```
1  begin
2      if |Parameters_1| ≠ |Parameters_2| then
3          return false
4      for i ← 1 to |Parameters_1| do
5          (Name_1, Type_1) ← Parameters_1[i]
6          (Name_2, Type_2) ← Parameters_2[i]
7          if Type_1 ≠ Type_2 then
8              return false
9          Body_2.Replace(Name_2, Name_1)
10     return Body_1 = Body_2
```

For example, given

$$F_1 := (\mathit{Add},\ \{(x, \text{Number}), (y, \text{Number})\},\ \text{Number},\ \texttt{x} + \texttt{y})$$
$$F_2 := (\mathit{Add},\ \{(a, \text{Number}), (b, \text{Number})\},\ \text{Number},\ \texttt{a} + \texttt{b})$$

we replace references to $\mathit{Parameters}_2$ in Body_2 (a and b) with references to $\mathit{Parameters}_1$ (x and y). Thus $\mathit{Body}_2 = \texttt{x} + \texttt{y}$ and $\mathit{Body}_1 = \mathit{Body}_2$.

Structural equivalence implies semantical equivalence, but not the other way around. For example, given

$$F_1 := (\mathit{Halve},\ \{(x, \text{Number})\},\ \text{Number},\ \texttt{0.5} * \texttt{x})$$
$$F_2 := (\mathit{Halve},\ \{(x, \text{Number})\},\ \text{Number},\ \texttt{x/2.0})$$

we can observe that F_1 and F_2 are *semantically*, but not *structurally* equivalent.

4 Implementation

In this chapter, we examine the actual process of generating code from spreadsheet formulas.

As part of this thesis, I provide a reference implementation written in C# 7.3. The implementation produces code in C#.

4.1 Code generation

We use the .NET Compiler SDK ("Roslyn") [11] to build C# syntax trees, which the Roslyn API then translates into code.

For example, the method

```
double CalcTotalCost(double subtotal) {
   return subtotal * (1 + totalTax);
}
```

can be generated with

```
MethodDeclaration(
   PredefinedType(Token(SyntaxKind.DoubleKeyword)),
   Identifier("CalcTotalCost"))
   .AddParameterListParameters(
      Parameter(Identifier("subtotal"))
         .WithType(PredefinedType(Token(SyntaxKind.DoubleKeyword))))
   .WithBody(Block(
      ReturnStatement(
         BinaryExpression(
            SyntaxKind.MultiplyExpression,
            IdentifierName("subtotal"),
            ParenthesizedExpression(
               BinaryExpression(
                  SyntaxKind.AddExpression,
                  LiteralExpression(SyntaxKind.NumericLiteralExpression, Literal(1)),
                  IdentifierName("totalTax")))))));
```

As we can see, syntax trees can become quite convoluted when written out. By using the **divide and conquer** paradigm, code generation can be broken down into simpler parts.

To do that, we once again examine the parse tree generated from a cell's formula (see Section 2.2). We can directly map production rules into code generation procedures.

For example, the production rule "Constant" can be mapped into a `LiteralExpression`. A "Cell" can be mapped into an `IdentifierName` expression, referencing the variable name of the cell. A "FunctionCall" can be mapped in various ways, as described in Section 4.2.4.

We define a series of mapping procedures, one for each production rule, until we are able to map entire formulas.

4.2 Implementation details

In this section, we examine some of the challenges that were encountered during the implementation of the code generator.

4.2.1 Type safety

Spreadsheet formulas are not, by any means, type-safe. For example, the logical formula `IF(A1=0, "foo", 42)` is perfectly legal and will return the *text* "foo" if $Value(\text{A1}) = 0$ or otherwise the *number* 42.

This presents a challenge when we try to translate formulas that are not type safe into a (relatively) type-safe language such as C#.

Our solution is to set $Type(c) =$ Unknown when we encounter a formula which is not type-safe at cell c. Then, when we generate code for Unknown cells, we use C#'s **dynamic** types feature[1], which tells the compiler that a variable's type is not known until runtime, and can even change during runtime:

```
1 dynamic cell = A1 == 0 ? (dynamic)"foo" : (dynamic)42;
```

4.2.2 Empty cells

Another concept that does not translate so well into conventional programming languages are **empty cells** in spreadsheets. For example, if A1 is an empty cell, `=1+A1` will return 1, `=A1&"foo"` will return "foo" and `=A1=0`, `=A1=""` and `=A1=FALSE` will all return TRUE.

We model an empty cell's behavior in a class called `EmptyCell`, see Listing 4:

[1] https://docs.microsoft.com/dotnet/csharp/programming-guide/types/using-type-dynamic

Listing 4: EmptyCell

```
1  public class EmptyCell {
2     public static implicit operator double(EmptyCell a) => 0;
3     public static implicit operator string(EmptyCell a) => "";
4     public static implicit operator bool(EmptyCell a) => false;
5
6     public static double operator +(EmptyCell a, EmptyCell b) { return 0; }
7     public static double operator -(EmptyCell a, EmptyCell b) { return 0; }
8     public static double operator *(EmptyCell a, EmptyCell b) { return 0; }
9     public static double operator /(EmptyCell a, EmptyCell b) { /* Error */ }
10    public static double operator %(EmptyCell a, EmptyCell b) { /* Error */ }
11    public static bool operator ==(EmptyCell a, EmptyCell empty) { return true; }
12    public static bool operator !=(EmptyCell a, EmptyCell b) { return false; }
13    public static bool operator <(EmptyCell a, EmptyCell b) { return false; }
14    public static bool operator >(EmptyCell a, EmptyCell b) { return false; }
15    public static bool operator <=(EmptyCell a, EmptyCell b) { return true; }
16    public static bool operator >=(EmptyCell a, EmptyCell b) { return true; }
17 }
18 public static EmptyCell Empty = new EmptyCell();
```

Here we take advantage of operator overloading. Whenever an EmptyCell is used in a function, C# performs an automatic cast to the type of the other argument, which is defined by implicit operator overloading (lines 2–4). If both arguments are empty cells, a pre-defined value is returned (lines 6–16).

Now, empty cells will be handled correctly:

```
1 double d = 1 + Empty; // returns 1
2 string s = Empty + "foo"; // returns "foo"
3 bool b = Empty == false; // returns true
```

4.2.3 Ranges

In spreadsheets, ranges are expressions like `A1:C3` that correspond to our definition of regions $\mathcal{R}$. In Excel terminology, they are also referred to as "Matrices" or "Arrays". Ranges are often used for shorthand notations; for example, one could write `SUM(A1:A3)` instead of `A1+A2+A3`. However, they are perhaps most useful when used inside reference functions (see Section 4.2.4).

We model ranges using two-dimensional arrays. To improve legibility, we define wrapper classes `Matrix`, `Row` and `Column`. For example, the range `A1:C3` can be expressed in code as either one of:

```
Matrix.Of(Row.Of(A1, B1, C1),
          Row.Of(A2, B2, C2),
          Row.Of(A3, B3, C3));
```

```
Matrix.Of(Column.Of(A1, A2, A3),
          Column.Of(B1, B2, B3),
          Column.Of(C1, C2, C3));
```

When we encounter a range, we create a new variable of type `Matrix`. Ranges can consist only of constants (e. g. `Row.Of(1, 2, 3)`) or reference variables defined elsewhere, e. g.

```
double applesCount = 10;
double bananasCount = 5;
double orangesCount = 20;
Matrix inventory = Matrix.Of(Row.Of("Apples", applesCount),
                             Row.Of("Bananas", bananasCount),
                             Row.Of("Oranges", orangesCount));
```

4.2.4 Functions

Spreadsheet software implement a wide variety of functions, including basic arithmetic functions (`SUM`, `MIN`, `ROUND` etc.), logic functions (`IF`, `AND` etc.), reference functions (`VLOOKUP`, `CHOOSE` etc.) and more.

When we generate code, we need to replicate the behavior that those functions bring with them. While some operations can be mapped directly onto C# language features, most functions require us to define our own methods.

As part of the implementation, I implemented some of the most commonly used Excel functions as functions in C#.

Arithmetic functions

SUM	Adds numbers in arguments together. If an argument is a `Matrix`, sums all values in the matrix.
MIN **MAX**	Finds the smallest/biggest value in the provided arguments. If an argument is a `Matrix`, considers the smallest/biggest value in the matrix. If no valid arguments are provided, returns 0.
COUNT	Counts the number of numbers provided in the arguments. If an argument is a `Matrix`, adds the number of numbers in the matrix.
AVERAGE	Defined as `SUM` / `COUNT`.

ROUND	As Excel allows for the rounding to a negative number of digits (e. g. $Round(250, -2) = 300$), we have to extend the rounding logic. With number x and number of digits d: $$Round(x,d) := \begin{cases} -\mathrm{sgn}(x)\frac{\lceil -\lvert x\cdot 10^d\rvert - 0.5\rceil}{10^d} & \text{if } d \geq 0 \\ \frac{Round(x\cdot 10^d, 0)}{10^d} & \text{if } d < 0 \end{cases}$$ Here, we round half away from zero.
ROUNDUP	Rounds away from zero: $$RoundUp(x,d) := \frac{\lceil x \cdot 10^d \rceil}{10^d}$$ For example, $RoundUp(201, -2) = 300$.
ROUNDDOWN	Rounds towards zero: $$RoundDown(x,d) := \frac{\lfloor x \cdot 10^d \rfloor}{10^d}$$ For example, $RoundDown(299, -2) = 200$.

Note that in Excel, string arguments that are numbers (e. g. `"20"`) or percentages (e. g. `"15%"`, equal to `15/100`) are parsed into numbers during the evaluation of the function.

Logical functions

IF	Called with arguments *condition*, *ifTrue*, *ifFalse*. Translates to a ternary conditional operation `condition ? ifTrue : ifFalse`. When *ifFalse* is not provided, *ifFalse* will equal `false`. This can cause a type safety problem (see Section 4.2.1), e. g. `IF(A1=0, 100)` will translate into `A1 == 0 ? (dynamic)100 : (dynamic)false`.
NOT	With a single logical argument *logical*, returns `!logical`.
AND **OR** **XOR**	Folds the arguments into a tree of binary expressions of type `&&`, `||` and `^` respectively. For example, `AND(A1=0, A2=0, A3=0)` translates into `A1 == 0 && A2 == 0 && A3 == 0`.

Reference functions

Reference functions operate on cell values inside a Matrix that is provided as an argument.

VLOOKUP

Called with arguments *lookupValue*, *matrix*, *columnIndex* and an optional argument *matchMode*.

`VLOOKUP` searches for the *lookupValue* inside the first column of the matrix. When found, it returns the value from the column with index *columnIndex*.

For example, given a matrix *inventory*,

```
Matrix inventory = Matrix.Of(Row.Of("Apples", 10),
                             Row.Of("Bananas", 5),
                             Row.Of("Oranges", 20));
```

calling `VLOOKUP("Bananas", inventory, 2)` will return 5.

The last parameter *matchMode* specifies whether to perform an **approximate** (*matchMode* $\in \{1, \text{TRUE}\}$ or not provided) or an **exact** match (*matchMode* $\in \{0, \text{FALSE}\}$).

If *lookupValue* does not exist within *matrix*, exact matching will throw an error, while approximate matching will return the lookup for the first value that is *smaller* than *lookupValue*.

For example, `VLOOKUP("B", inventory, 2, TRUE)` will return 10, because technically, `"Apples"` < `"B"` < `"Bananas"`.

HLOOKUP

Called with arguments *lookupValue*, *matrix*, *rowIndex* and an optional argument *matchMode*, `HLOOKUP` is very similar to `VLOOKUP` and searches for the *lookupValue* inside the first **row** of the matrix. When found, it returns the value from the row with index *rowIndex*.

For example, given a matrix *inventory*,

```
Matrix inventory = Matrix.Of(Row.Of("Apples","Bananas","Oranges"),
                             Row.Of(10, 5, 20));
```

calling `HLOOKUP("Bananas", inventory, 2)` will return 5.

CHOOSE

Called with arguments *index*, $value_1, \ldots, value_n$, returns $value_{index}$. For example, `CHOOSE(3, A1, A2, A3, A4, A5)` returns A3.

MATCH

Called with arguments *lookupValue*, *matrix* and optionally *matchMode*. *matrix* must be one-dimensional, i. e. consist of only one column or row.

For $matchMode = 1$ or omitted, returns the index of the first value less than or equal to *lookupValue* within *matrix*.

For $matchMode = 0$, returns the index of the first value equal to *lookupValue* within *matrix*. Throws an error if not found.

For $matchMode = -1$, returns the index of the first value greater than or equal to *lookupValue* within *matrix*.

For example, with

```
1 Matrix fruit = Matrix.Of(Row.Of("Apples","Bananas","Oranges"));
```

calling `MATCH("Bananas", fruit, 0)` will return 2.

INDEX

Called with arguments *matrix*, *rowIndex* and optionally *colIndex*, which defaults to 1.

Returns $matrix_{colIndex,rowIndex}$.

Operators

Operators such as `+`, `-`, `<=` etc. can usually be translated to code one-to-one. The only exception is the Equals-Operator (`=`) on strings, which is case-insensitive by default in Excel. Thus we translate the string comparison operator `=` with a custom function `CIEquals`, which performs a case-insensitive string comparison.

4.2.5 Names

A common feature in spreadsheet software is the ability to assign *names* to cells or entire ranges for better legibility in formulas. When we encounter such names, we adopt those names instead of generating a name on our own.

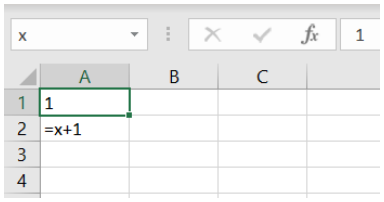


Figure 11: Here, A1 was named “x”.

4.2.6 External references

While formulas usually reference cells in the current worksheet, they can also reference cells in other worksheets, for example `Worksheet2!A1`. Those external references are

discovered during graph generation and added to the set of cells. External references are always considered Constants, regardless of whether they reference other cells.

4.3 User interface

The user interface is implemented in Windows Presentation Foundation (WPF) with the Metro UI toolkit.

Functionally, the interface is divided into three parts (see Figure 12). The top left part consists of the Excel file that has been loaded into the tool. The user may select one of the worksheets in the file. To display and get data from the spreadsheet, the SfSpreadsheet library[2] is used.

The bottom left part consists of a toolbox, which displays values for the currently selected cell and allows the user to set custom names. It also includes the log, which displays information, warnings and error messages.

The right part consists of tabs for graph generation (Figure 12), class generation (object-oriented approach only – Figure 13) and code generation (Figure 14).

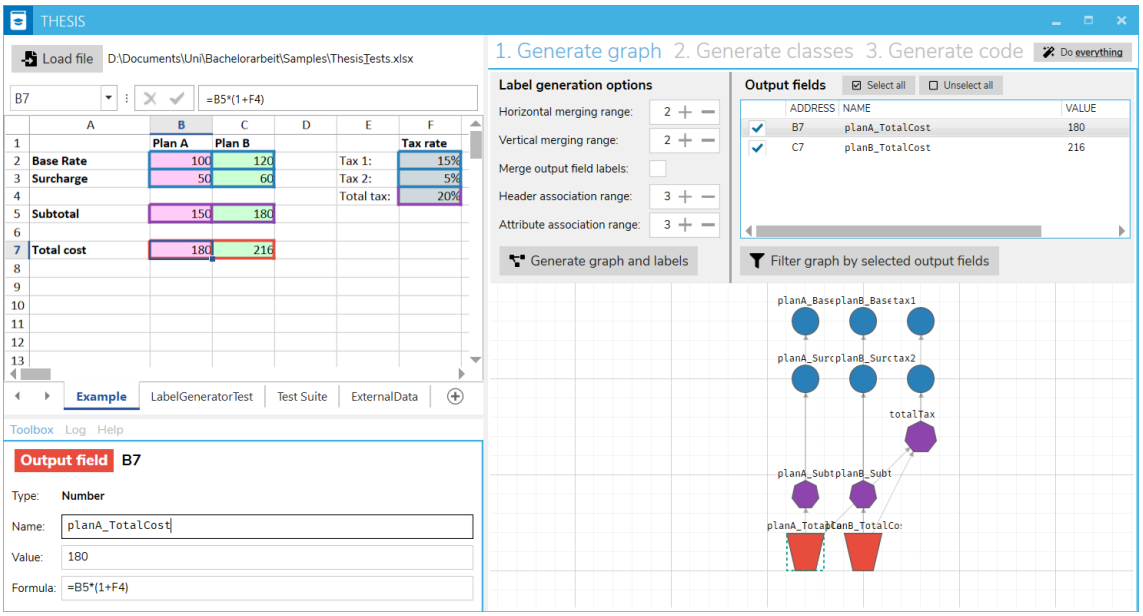


Figure 12: User interface with graph generation tab selected. The loaded spreadsheet is the one in Figure 8.

[2] https://help.syncfusion.com/wpf/sfspreadsheet/overview

4.3.1 Graph generation

In the graph generation tab, we can set parameters for label generation (see Section 2.4). There is also a list of OutputFields that allows for omission of certain OutputFields from generation.

Excluding OutputFields also filters out their children, if they are not used by any other OutputField. In large and complex spreadsheets, we often want to focus on certain aspects of the calculation logic. Selecting only the OutputFields we need is a good way to maintain clarity.

The results of the graph generation are rendered in a diagram, which displays cells as nodes. To draw the diagram, the SfDiagram library[3] is used.

4.3.2 Class generation

In the class generation tab, which is only visible in the object-oriented approach, generated classes are shown in a class diagram, which displays the cells in each class. For better readability, classes are assigned a random color, which is also used to color cells in the spreadsheet.

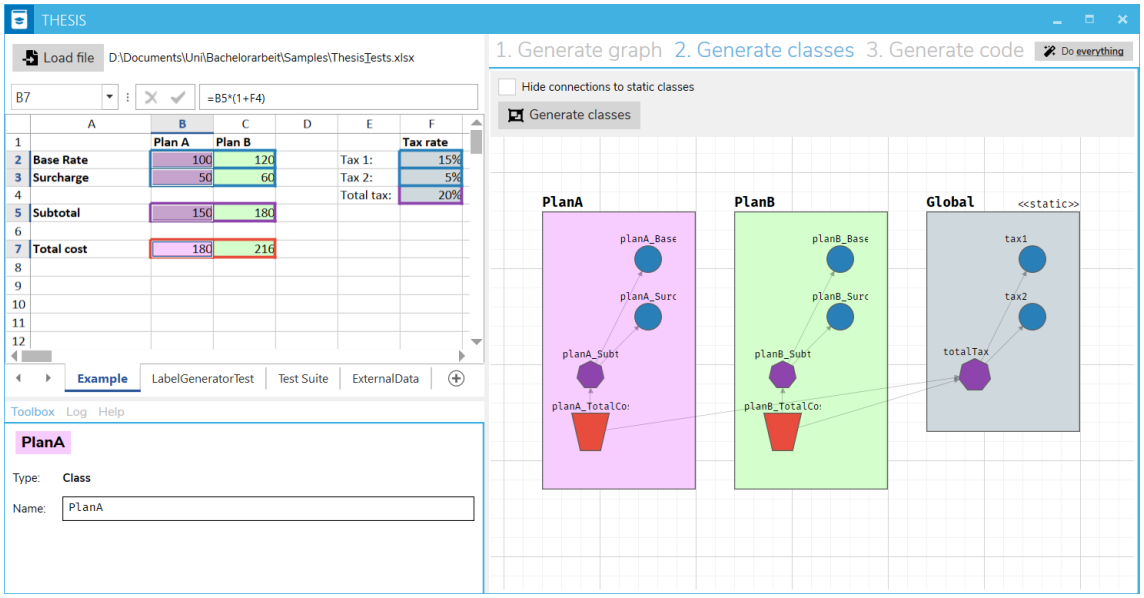


Figure 13: User interface with class generation tab selected (object-oriented approach only)

[3] https://help.syncfusion.com/wpf/sfdiagram/overview

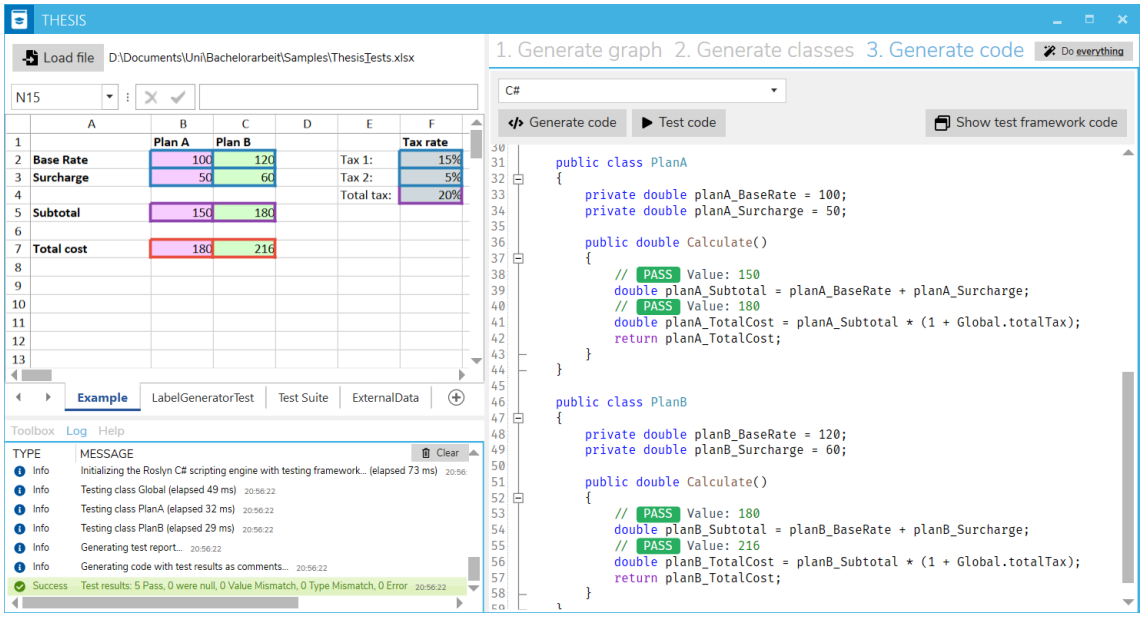


Figure 14: Automated testing

4.4 Automated testing

In the code generation tab, we can not only generate code, but **test** the code right away. Once again using the .NET Compiler SDK ("Roslyn") [11], we first load our testing framework into Roslyn's scripting engine. The testing framework consists of helper methods (`VLOOKUP` etc.) and classes (`EmptyCell`, `Matrix` etc.) that were described in Section 4.2.

Then, the generated code is loaded into the scripting engine and executed.

Finally, the value of each variable is compared with the actual value in the spreadsheet, similar to unit tests in software engineering. The result of each comparison can be:

- **Pass**. The tested value matches the actual value in the spreadsheet.
- **Null**. The tested value was null, indicating an error.
- **Value mismatch**. The tested value was calculated, but differs from the actual value in the spreadsheet.
- **Type mismatch**. The tested value had a different type than expected, indicating a runtime error due to dynamic typing.
- **Error**. The code failed to compile or threw an exception during runtime.

5 Conclusion and future work

In this thesis, we examined the process of generating code from spreadsheets. We did this by considering a formal model for spreadsheets (Chapter 2) and methods to translate that model into programming concepts (Chapter 3). Finally, we examined a reference implementation in Chapter 4.

It was not the aim of this thesis to completely replicate a spreadsheet's entire formula engine, which includes a huge number of functions, rules, and edge cases. We have to consider boundaries of what can effectively be translated into code. For example, there are expressions like `=MIN(TRUE,TODAY())` [1] that are perfectly acceptable in spreadsheets, but make zero sense in real-world scenarios. These edge cases will currently either not compile or throw a run-time error when encountered.

For future work, we could strive to improve the code generation by replacing "spreadsheet concepts" with "programming concepts". The most obvious example would be matrices, the only data structure available within spreadsheets. While matrices are very flexible, they are often impractical in real world scenarios. Replacing matrices with proper data structures such as lists, dictionaries or even databases can drastically improve the quality of the code.

For example, instead of generating

```
Matrix inventory = Matrix.Of(Row.Of("Apples", 10),
                             Row.Of("Bananas", 5),
                             Row.Of("Oranges", 20));
```

we could create a dictionary

```
var inventory = new Dictionary<string, int>
{
   { "Apples", 10 },
   { "Bananas", 5 },
   { "Oranges", 20 },
};
```

Then, all `VLOOKUP` calls could be replaced by simple dictionary lookups, making the code much more readable.

[1] Incidentally, this formula will yield the date 1900/01/01, as TRUE and TODAY() are both casted into numbers (with TODAY() returning the number of days since 1900/01/00), evaluated by MIN, and casted back into a Date.

Another major issue in code generation from spreadsheets is that the quality of the output depends heavily on the quality of the input. A spreadsheet that contains major inconsistencies or even gross errors will yield bad, subpar code.

Thus, before generating code, we could try to implement systems that recognize and correct errors [12] as well as code smells [13] in spreadsheets.

Bibliography

[1] F. Hermans, B. Jansen, S. Roy, E. Aivaloglou, A. Swidan, and D. Hoepelman, "Spreadsheets are code: An overview of software engineering approaches applied to spreadsheets", in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5, 2016, pp. 56–65.

[2] F. Hermans, M. Pinzger, and A. van Deursen, "Automatically extracting class diagrams from spreadsheets", in *ECOOP 2010 – Object-Oriented Programming*, vol. 6183, 2010, pp. 52–75.

[3] J. Cunha, M. Erwig, and J. Saraiva, "Automatically inferring classsheet models from spreadsheets", in *2010 International Forum on Information Technology and Applications (IFITA)*, 2010, pp. 93–100.

[4] X. Wang, "Tabular abstraction, editing, and formatting", Dissertation, University of Waterloo, 1996. [Online]. Available: `https://uwspace.uwaterloo.ca/bitstream/handle/10012/10962/WANG_XINXIN_.pdf`.

[5] E. Koci, M. Thiele, O. Romero, and W. Lehner, "A machine learning approach for layout inference in spreadsheets", in *Proceedings of the 8th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*, 2016, pp. 77–88.

[6] E. Koci, M. Thiele, W. Lehner, and O. Romero, "Table recognition in spreadsheets via a graph representation", in *2018 13th IAPR International Workshop on Document Analysis Systems (DAS)*, 2018, pp. 139–144.

[7] Microsoft, *Excel (.xlsx) Extensions to the Office Open XML SpreadsheetML File Format*, 2019. [Online]. Available: `https://docs.microsoft.com/en-us/openspecs/office_standards/ms-xlsx/2c5dee00-eff2-4b22-92b6-0738acd4475e` (visited on Jun. 27, 2019).

[8] E. Aivaloglou, D. Hoepelman, and F. Hermans, "A grammar for spreadsheet formulas evaluated on two large datasets", in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2015, pp. 121–130.

[9] A. B. Kahn, "Topological sorting of large networks", *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962.

[10] H. G. Rice, "Classes of recursively enumerable sets and their decision problems", *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953.

[11] Microsoft, *The .NET Compiler Platform SDK*, 2017. [Online]. Available: https://docs.microsoft.com/de-de/dotnet/csharp/roslyn-sdk/ (visited on Sep. 13, 2019).

[12] S. G. Powell, K. R. Baker, and B. Lawson, "A critical review of the literature on spreadsheet errors", *Decision Support Systems*, vol. 46, no. 1, pp. 128–138, 2008.

[13] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting and refactoring code smells in spreadsheet formulas", *Empirical Software Engineering*, vol. 20, no. 2, pp. 549–575, 2015.