IT-University of Copenhagen

Master Thesis

Funsheet

Integration of Sheet-Defined Functions in Excel using C#

Authors:
Simon Eikeland Timmermann
Jonas Druedahl Rask

Supervisor:
Prof. Peter Sestoft

A thesis submitted in fulfilment of the requirements for the degree of MSc of IT: Software Engineering

1st June 2014

IT-UNIVERSITY OF COPENHAGEN

Abstract

MSc of IT: Software Engineering

Funsheet
Integration of Sheet-Defined Functions in Excel using C#

by
Simon Eikeland Timmermann
Jonas Druedahl Rask

In this thesis the concept of sheet-defined functions, found in the research spreadsheet implementation Corecalc and Funcalc developed by Peter Sestoft, is integrated into Excel through an Excel XLL add-in named Funsheet. A series of benchmark experiments have been constructed and executed to explore the most effective technology to use for the implementation of Funsheet. A collection of design goals targeting both functional and non-functional requirements of Funsheet have been identified. Considering the benchmark results and the design goals, Excel-DNA has been chosen as the superior technology to use for the implementation of Funsheet. The thesis then describes the design and implementation of the Funsheet add-in. Finally, the thesis describe how the implementation of Funsheet was verified through a collection of tests.

Contents

Abbreviations

UDF	**U**ser **D**efined **F**unction
SDF	**S**heet **D**efined **F**unction
VBA	**V**isual **B**asic for **A**pplications
COM	**C**omponent **O**bject **M**odel
TG	**T**echnical **G**oal
QG	**Q**uality **G**oal

Chapter 1

Introduction

1.1 Foreword

This Master's thesis is written by Jonas Druedahl Rask and Simon Eikeland Timmermann in the period from January 2014 to June 2014. With this thesis, we finish our Master degree in software development at the IT University of Copenhagen.

This paper describes a framework for conducting performance benchmark experiments in Microsoft Excel. It also describes how we used different technologies to implement simple, user-defined functions (UDF) to identify the technology currently yielding the best performance. Finally, we describe the process of designing and implementing the required Excel UDFs to support the concept of sheet-defined functions (SDF) using parts of the Funcalc[1] project. The original working title of our solution was *Excelcalc*, but was later changed to *Funsheet*.

To read and understand this thesis the reader should be familiar with how spreadsheets work. It also requires knowledge of the programming language Microsoft C# and a basic understanding of computer science.

1.2 Motivation

Spreadsheets offer an array of functionality and are widely used today. Peter Sestoft is a professor at the IT University of Copenhagen, and has been researching the area for

some time. He developed and implemented Funcalc[1], a research prototype that implements core spreadsheet functionality and sheet-defined functions (SDF)[2]. Through experiments with prototypes, we explored possibilities of integrating the SDF implementation from Funcalc into Microsoft Excel. This would allow users to introduce more advanced and programming functionality to their spreadsheets, while still maintaining the familiar interface and declarative syntax of Excel.

1.3 Results

We used exploratory prototyping as our method to find the most suitable technology to base Funsheet upon. In chapter 4 we describe the development of a series of user-defined functions where each UDF or prototype is evaluated based on their performance.

In chapter 5 we establish that Funsheet should be implemented using Excel-DNA[3]. We have been through a lot of iterations, perfecting the design and implementation. The result is a complete implementation of an Excel add-in supporting sheet-defined functions, higher order functions and recursive function definitions. This proves that it is possible to transfer the concept of SDFs from Funcalc to Microsoft Excel and thereby enable end-users to exploit SDFs while still keeping all the complex functions and tools present in Excel.

1.4 Thesis Organization

This thesis is organized into chapters. Chapter 2 facilitates the background knowledge required to read and understand this paper. In chapter 3 we present related work, relevant to this thesis. In chapter 4 we present a process for conducting experiments along with a series of experiments. We benchmark these experiments and evaluate the results. Chapter 5 describes the rationale behind the chosen solution along with the design goals and actual design of Funsheet. Chapter 6 describes the implementation of the solution. Chapter 7 describes our testing efforts to verify the realization of our design goals. In chapter 8 we propose ideas for future work. In chapter 9 we state the final conclusions and discuss how our solution align with our initial goals.

1.5 Deliverables

Besides this thesis report, we have handed in the following:

- Funsheet source code
- Technology Experiments source code
- The Funsheet add-in
- Latex source code

Chapter 2

Background

In this chapter we will cover the core technologies used in this thesis.

2.1 Spreadsheet Technology

Spreadsheet applications are used daily by millions of people all over the world. It enables users to organize and analyse data in a tabular format. The applications enable users to transform data in different ways by applying built-in and customized functions.

A spreadsheet is organized as a two-dimensional grid in which the horizontal axis is labelled with letters and the vertical axis with numbers, both starting from the upper left corner. Every combination of a letter and number is called a cell, where the address of the cell is represented by the unique combination of the letter and number eg. B17. Cells can contain text, numbers, functions or a combination of these. A collection of cells is called a range and is referenced by a start cell and an end cell, eg. A3:C7, which denotes a range containing the cells A3-A7, B3-B7 and C3-C7.

If the user demands functionality that is not present in the spreadsheet application, most common applications contain ways to extend the functionality by writing user-defined functions. Often, the creation of user-defined functions requires the user to have some knowledge of programming.

Some well-known spreadsheet applications are Microsoft Office Excel, Gnumeric and OpenOffice Calc.

2.2 Sheet-Defined Functions

Most ordinary spreadsheet users do not have any programming experience which makes it difficult for them to extend spreadsheet applications. The article "A User-Centred Approach to Functions in Excel" Peyton Jones, Blackwell and Burnett introduces a concept called sheet-defined functions (SDF)[2]. Basically SDFs is a way to integrate user-defined functions (UDF) into the grid of the spreadsheet application. This enables users to extend spreadsheet applications using knowledge they already possess. It further enables users to compose SDFs and built-in functions into new SDFs, avoiding the need to rewrite functions, thus lowering the margin of error.

The idea is illustrated in figure 2.1 in which the top sheet shows how the SDF F2C is called. The bottom sheet shows the definition of the F2C. It consists of a cell defining an input field, a cell containing an intermediate result, and a cell combining the former two to act as the function. Besides being easy to understand for non-programmers, it allows for splitting complex functions up into many intermediate results which makes it perform better, makes it easier to maintain and less error prone.

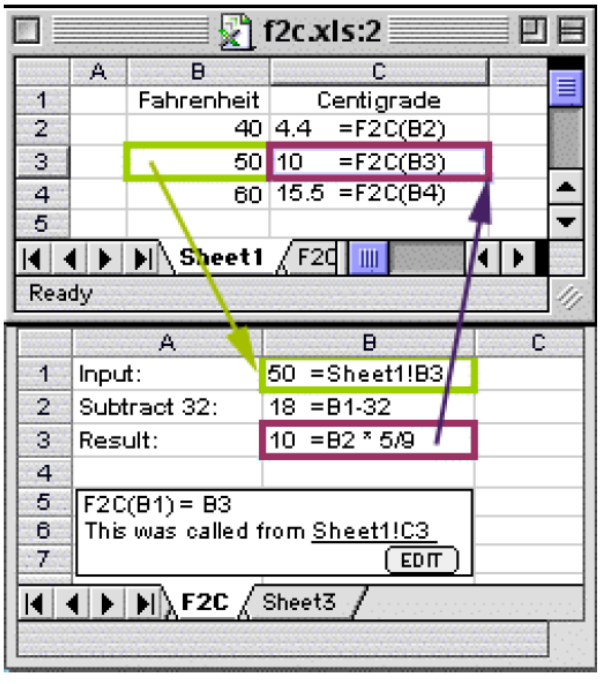

FIGURE 2.1: Invocation and definition of a SDF[2].

2.3 Corecalc and Funcalc

Corecalc is an implementation of core spreadsheet functionality in C#. Corecalc is a research project developed by Peter Sestoft[1]. It was developed to provide researchers with a platform for carrying out experiments and trying out new ideas in an open spreadsheet implementation. Besides being open-source, one of the advantages of Corecalc is that it comes with extensive documentation, which is in direct contrast to the mainstream spreadsheet applications.

Funcalc[1] is an extension of Corecalc. It introduces the concept of sheet-defined functions(SDFs), while retaining the familiar spreadsheet user interface. It works by translating SDFs at runtime to bytecode and caching this representation for future calls to the SDF.

In this thesis the term Funcalc corresponds to Corecalc with the Funcalc extension.

2.4 VBA

Visual Basic for Applications (VBA) is an implementation of Microsoft's programming language Visual Basic 6. It is built into most Microsoft Office applications along with an integrated editor for writing VBA code. It can be used to build user-defined functions (UDFs) in Excel and control many other aspects of the host application.

VBA is an interpreted language and therefore VBA code is never compiled. Naturally, this affects performance. Another limitation of VBA is the lack of multi-threading capabilities. Regardless of the limitations, VBA is still widespread and commonly used to extend Microsoft Office applications.

2.5 COM

Component Object Model (COM) is a binary-interface standard for software components introduced by Microsoft in 1993. COM defines a standard for compiling assemblies so they match a specific structure. This provides a way for sharing binary code across different applications and languages.

COM Interoperability, also called COM Interop, is a technology found in the .NET Common Language Runtime, that enables COM objects and .NET objects to interact. Microsoft Excel has the ability to consume COM objects. In this thesis we use COM and COM Interop to expose user-defined functions written in C# to Microsoft Excel.

When COM Interop is used, some performance overhead will occur, due to marshalling between processes.

2.6 XLL

XLL is used to create add-ins for Excel. XLL add-ins are written using the Excel C API, which offers the possibility to develop high-performance add-ins compared to using the COM approach. A downside of using the Excel C API is that the learning curve is quite steep compared to using COM.

2.7 Excel-DNA

Excel-DNA is an independent open-source project that aims to integrate .NET and Excel. It allows building native (.xll) add-ins for Excel using the .NET framework. Excel-DNA can take advantage of the COM Interop features from .NET, the native C XLL API, or a combination of both. The add-ins created with Excel-DNA can be compiled as a DLL file or an XLL file.

Excel-DNA works by hosting the .NET framework inside an XLL add-in. The .NET code is then injected into the hosted .NET framework and can thus run inside the add-in. Excel-DNA includes a .NET object model, resembling the C XLL API types to make it possible to bridge the gap between the C and the .NET border.

There is not a significant marshalling overhead when using Excel-DNA.

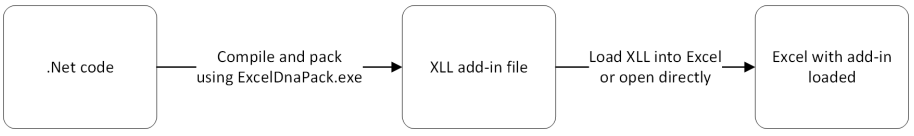

FIGURE 2.2: How .NET code interacts with Excel through Excel-DNA.

Chapter 3

Related Work

3.1 Introduction

In this chapter we present some research and related work that has acted as a foundation for our implementation of Funsheet. The related work has also greatly influenced our decisions and rationales.

3.2 The inception of sheet-defined functions

In 2003, Peyton Jones et al. first described the concept of sheet-defined functions[2]. They envisioned an extension to Excel allowing users to define functions directly in the sheets. Users should be able to do this by leveraging their existing knowledge of how spreadsheet applications work. By applying this user-centric approach to language design, Peyton Jones et al. managed to design a declarative and easy to use programming interface that integrates nicely with the existing concepts of Excel. However, this concept was never implemented by Peyton Jones et al.

Peyton Jones et al. describe the benefits of sheet-defined functions. Sheet-defined functions can be used to avoid tedious repetitions of complex and hard to read expressions by encapsulating these expressions in a sheet-defined function. Not only does this promote readability, but is also beneficial for maintainability. Instead of copying expressions, which introduces a maintenance burden if the expression needs to change, it is possible to only change the definition of the sheet-defined function. This way of achieving

centralized maintenance by encapsulation is very similar to the way software is built using imperative languages. It also enables the creation of dedicated SDF libraries that can be used to promote re-use across an organization and minimize the risk of errors. Peyton Jones et al. also pointed out that there is a potential for performance benefits by representing SDFs using an abstract syntax or even compiling SDFs into bytecode.

3.3 Corecalc and Funcalc: The realization

In 2006 Peter Sestoft started researching in the area of sheet-defined functions(SDF)[1]. He began the development of Corecalc, a spreadsheet implementation written in C#. The project aimed to provide an experimental platform to test different aspects of implementing a spreadsheet application. Later, Funcalc was developed by Peter Sestoft[4]. Funcalc is an extension to Corecalc, supporting the concept of sheet-defined functions. Funcalc enabled the definition of SDFs and runtime generation of bytecode to allow high performance when executed. Corecalc and Funcalc is a great platform for experimenting, but lacks most functions and tools found in Microsoft Excel. It is therefore not a viable alternative to Excel for users.

3.4 Excel-DNA

Excel-DNA is an open-source project aimed to ease the development of user-defined functions in Excel[3]. Excel-DNA was started by Govert van Drimmelen, who has developed and supported the framework from the project's beginning. Excel-DNA has evolved over time and has a large array of use cases including development of user-defined functions, custom user interfaces and macros. One interesting feature of Excel-DNA is the recent addition of runtime registration of delegates as user-defined functions in Excel.

The Excel-DNA project has a large and active community in which the author of Excel-DNA, Govert, actively participates and constructively helps developers answer their questions.

Chapter 4

Experiments

We aimed to conduct experiments on the efficiency of using user-defined functions (UDF) in Excel. The UDFs are based on prototypes implemented through three different technologies: COM Interop, VBA and Excel-DNA. We assess how big a performance penalty can be attributed to the data transfer from Excel to the UDFs versus the overhead of function calling. In some experiments we included the function in Microsoft Excel corresponding to the current UDF.

4.1 Hypotheses

Here we present some hypotheses regarding the performance of the three technologies:

1. We expect COM Interop to be slower in all cases in which we transfer great amounts of data back and forth between Excel and the .NET assembly. This is due to the heavy marshalling of objects between processes when using COM Interop[5].

2. We expect UDFs defined in VBA to have a larger overhead for function calling than COM Interop and Excel-DNA. We assume this is because of VBA being an interpreted language[6].

3. We expect UDFs defined in VBA to perform better than COM Interop and Excel-DNA in situations where much data is passed as arguments (eg. the Average function that takes large number of cells). We propose that this advantage is

due to the fact that VBA is being executed in-process and therefore avoids the overhead of marshalling data between processes[7].

4. We expect Excel-DNA to have a lower overhead with regard to calling functions and transferring data than COM Interop, because the data is passed through the native Excel C API[3]. Some marshalling will be required when the data is passed to managed C#.

5. We expect Microsoft Excel to have the best performance by avoiding the overhead that comes with the other technologies.

4.2 Experimental Setup

We define three user-defined functions (UDF). Each UDF will be implemented in each of the three technologies, resulting in a total of nine UDFs. There is an experiment for each UDF with different volumes of data. Each experiment consists of an individual Excel sheet, in order to minimize the risk of influences from other sheets. Each experiment is manually executed one hundred times by a human being and the results/running times are written down. The execution is carried out using Excel's "Workbook Forced Recalculation" function (CTRL-SHIFT-ALT-F9). The UDFs and the experiments have been designed with gauging overhead of function calls and data transfer in mind, as we believe these to be the main performance defining factors. Details of the setup of each UDF are described in the following sections.

4.2.1 UDF 1 Setup - Generate Random Number

The first UDF generates and returns a single random number between 0 and 1. We will use this UDF to measure the overhead of executing UDFs with each technology. This UDF was chosen because it is relatively simple while being naturally volatile. It is also easy to verify its volatility, by checking if the randomly generated numbers change. For the first UDF, we have created three experiments for each technology. One experiment executes the UDF 10,000 times, the next executes 100,000 times and the last 1 million times.

To setup the experiment for UDF 1, we start out by adding the actual add-in to the Excel sheet instance. For measuring start time, we insert the Excel built-in function `NOW` into a cell. `NOW` returns the current date and time, including milliseconds. Then a number of cells corresponding to the desired execution count are populated with calls to UDF 1. To make sure the start time is calculated before anything else, we make the cells calling UDF 1 depend on the start time, by adding the start time to the result of UDF 1. An end time cell is then defined using another call to Excel's `NOW` function. To ensure this cell is evaluated last, we create another dependency by having a new cell calculate the sum of all the UDF 1 calls using Excel's `SUM` function. After that we are able to insert a logical statement into the end time cell, evaluating the sum cell. These dependencies are depicted in figure 4.1.

	A	B	C	D
1	=ExcelInteropRandom()+$D$2	=ExcelInteropRandom()+$D$2		**Start time (mm:ss,000)**
2	=ExcelInteropRandom()+$D$2	=ExcelInteropRandom()+$D$2		=NOW()
3	=ExcelInteropRandom()+$D$2	=ExcelInteropRandom()+$D$2		
4	=ExcelInteropRandom()+$D$2	=ExcelInteropRandom()+$D$2		
5	=ExcelInteropRandom()+$D$2	=ExcelInteropRandom()+$D$2		=SUM(A1:B11)
6	=ExcelInteropRandom()+$D$2	=ExcelInteropRandom()+$D$2		
7	=ExcelInteropRandom()+$D$2	=ExcelInteropRandom()+$D$2		**End time (mm:ss,000)**
8	=ExcelInteropRandom()+$D$2	=ExcelInteropRandom()+$D$2		=IF(D5>10;NOW();0)
9	=ExcelInteropRandom()+$D$2	=ExcelInteropRandom()+$D$2		
10	=ExcelInteropRandom()+$D$2	=ExcelInteropRandom()+$D$2		**Total time (mm:ss,000)**
11	=ExcelInteropRandom()+$D$2	=ExcelInteropRandom()+$D$2		= D8 - D2

FIGURE 4.1: The `EXCELINTEROPRANDOM` is implemented using COM Interop. All the calls to `EXCELINTEROPRANDOM` depend on the start time. This ensures that the start time is calculated first. The end time depends on the sum of the `EXCELINTEROPRANDOM` function calls, which ensures that the end time is calculated last.

4.2.2 UDF 2 Setup - Calculate Square Root

The second UDF accepts a double as an argument, calculates and returns the square root of the argument. Like UDF 1, UDF 2 was also created to measure the overhead of executing UDFs with each technology. During the experiment process we realized, that random number generators could be implemented differently in C# and VBA. So to minimize the error margin on the benchmark results revolving function call overhead, we implemented UDF 2. This decision is further elaborated on in 4.3.4. For UDF 2, we created three experiments for each technology and one in which we use Excel's built-in square root function. One experiment executes the UDF 10,000 times, the next executes 100,000 times and the last 1 million times. The setup is similar to the one in UDF 1, except for the fact that it is calling a different UDF and is using Excel's square

root function. We decided on including Excel's built-in square root function to further visualize the overhead that exists. This decision is also further elaborated on in 4.3.4.

4.2.3 UDF 3 Setup - Calculate Average of Cell Range

The third UDF accepts a cell range as an argument, calculates and returns the average of the cells in the range. We will use this UDF to measure the performance penalty associated with transferring data as parameters with each technology. For the third UDF, we have created three experiments for each technology and one where we use Excel's built-in average function. One experiment where the UDF is called a single time with a cell range of 100,000, one with a cell range of 500,000 and finally one with a cell range of 1 million. The decision to include Excel's built-in function is to further visualize the overhead of data transfer. This is further elaborated on in 4.3.4.

The setup of UDF 3 varies a bit from the others. In this case there is only a single call to the UDF, but many cells acting as parameters. The order of calculation is enforced using the same technique as in UDF 1 and UDF 2, which is depicted in figure 4.2.

	A	B	C	D	E
1	5	5		**Start time (mm:ss,000)**	
2	5	5		=NOW()	
3	5	5			
4	5	5		**=IF(D2 > 1;ExcelDnaAverage(A1:B10);0)**	**<-- ExcelDnaAverage call**
5	5	5		**End time (mm:ss,000)**	
6	5	5		=IF(D4>4;NOW();0)	
7	5	5			
8	5	5			
9	5	5		**Elapsed time (ms)**	
10	5	5		=(D6-D2)	

FIGURE 4.2: The `EXCELDNAAVERAGE` is implemented using Excel-DNA. The call to `EXCELDNAAVERAGE` is dependent on the start time. This ensures that the start time is calculated first. To ensure that the end time is calculated last, it is made dependant on the result of the `EXCELDNAAVERAGE`.

4.3 Results and Evaluation

This section presents the results and our reflections of the experiments conducted with the three UDFs.

4.3.1 UDF 1

The measured performance results for this UDF was, from the on average slowest to the on average fastest: VBA, COM Interop and Excel-DNA. This is depicted in figure 4.3. UDF 1 was created in order to measure some of the overhead that exists when calling functions from the different technologies. We therefore aimed to create functions that are as simple as possible while calling them many times, in order to isolate the overhead as much as possible. In hypothesis 4 we expect Excel-DNA to perform better than COM Interop, due to its utilization the native C Api. This claim is supported by the results, where we can see that Excel-DNA on average performs each function call 2000 ns faster than COM Interop does.

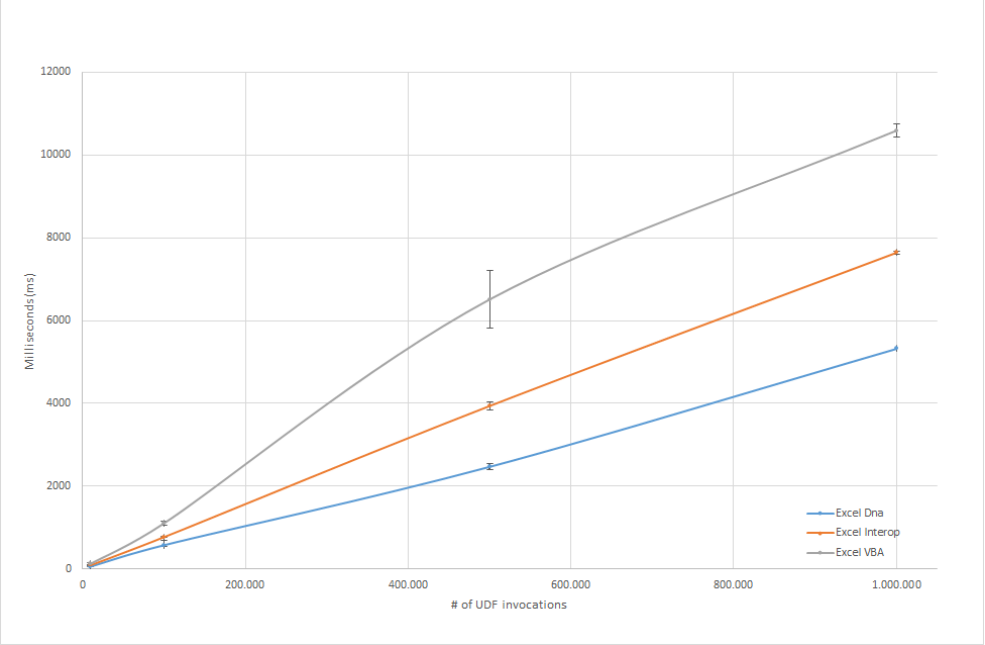

FIGURE 4.3: Random benchmark: Excel-DNA is the best performer of the three add-ins. Detailed data can be found in table 4.1.

4.3.2 UDF 2

The measured performance results for this UDF were from the on average slowest to the on average fastest: VBA, COM Interop, Excel-DNA, Excel. This is depicted in figure 4.4 Similar to UDF 1, UDF 2's purpose is to gauge the overhead when calling functions using COM Interop, VBA, Excel-DNA and Microsoft Excel. In hypothesis 2 we expect VBA, as an interpreted language, to have a larger overhead that COM

Interop and Excel-DNA. This hypothesis is supported by our results, where we see VBA having the slowest average performance in both the UDF 1 and UDF 2 experiments. Microsoft Excel's square root function with 1 million cells is 78 times faster than its VBA equivalent. We attribute the performance gap to the overhead of calling functions outside of Excel.

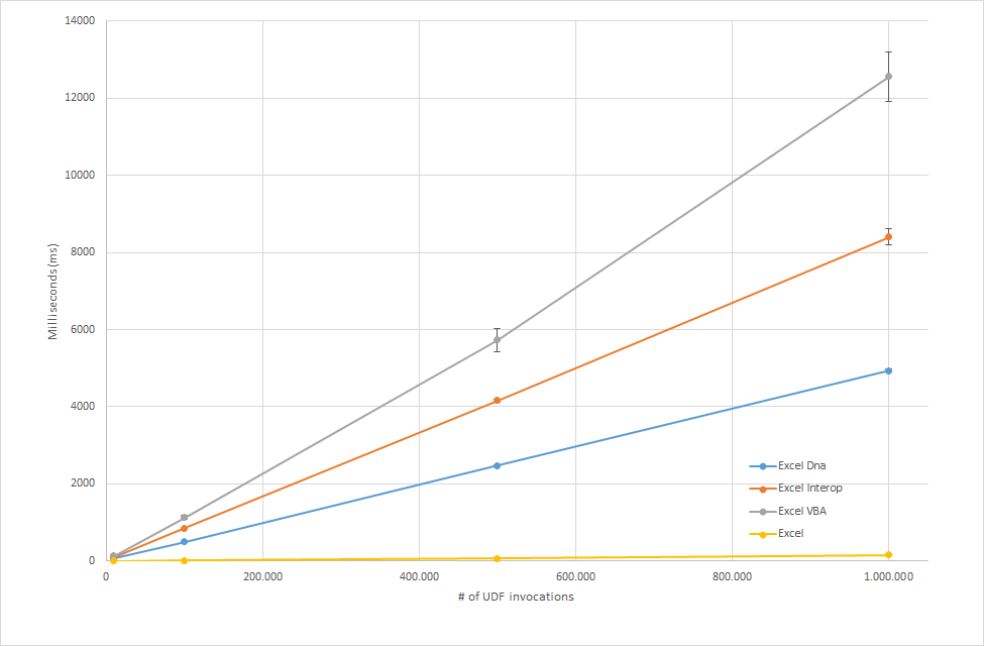

FIGURE 4.4: Square root benchmark: Excel-DNA is the best performer of the three add-ins. Detailed data can be found in table 4.2.

4.3.3 UDF 3

The measured performance results from this UDF was from the on average slowest to the on average fastest: Excel VBA, Excel-DNA and COM Interop. This is also depicted in figure 4.5 In hypothesis 1, we expect COM Interop to be slower when great amounts of data are transferred. This hypothesis is supported by the benchmark results which show that COM Interop is less efficient than VBA and Excel-DNA, when much data is transferred. Excel-DNA remains the most efficient throughout this experiment. We expect the reason that VBA retains some of its performance is due to it being in-process as opposed to COM Interop and Excel-DNA. In hypothesis 3 we state that we expect VBA to outperform the two other technologies when working with great amounts of data. This has clearly proven to be a faulty assumption, since Excel-DNA considerably

outperforms the two others, with COM Interop being the slowest performer. Microsoft Excel has the best performance in this experiment.

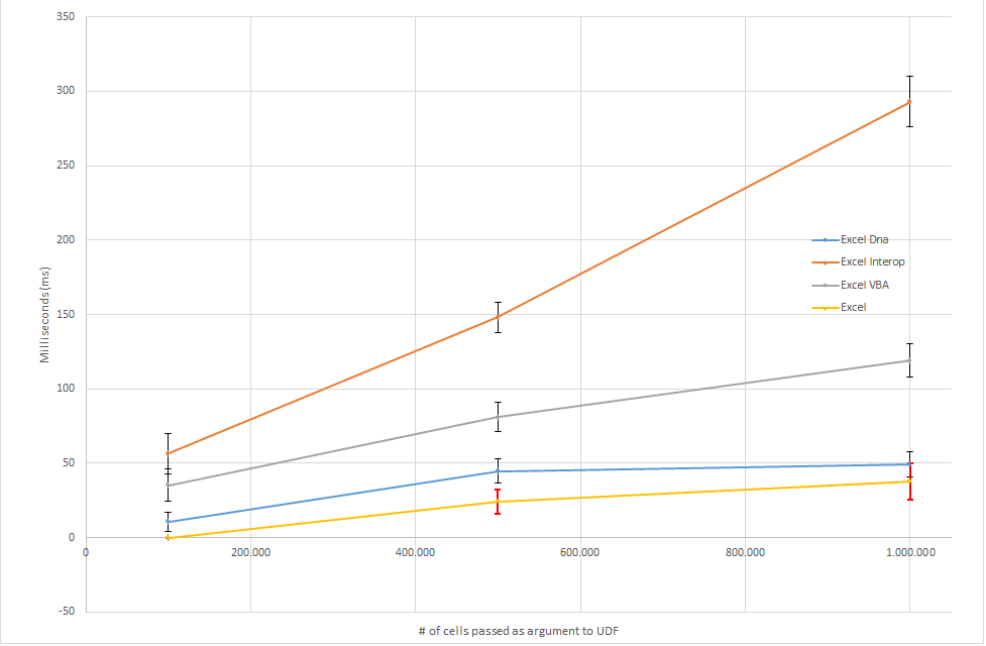

FIGURE 4.5: Average-benchmark: Excel-DNA is the best performer of the three add-ins. Detailed data can be found in table 4.3.

4.3.4 Evaluation

These experiments were designed to provide us with insight in how UDFs implemented using different technologies perform in regards to overhead of function calls and data transfer. We initially implemented UDF 1 and UDF 3, thinking they would suffice to gauge overhead of function calls and data transfer. In later iterations it occurred to us that the built-in random number generators in C# and VBA, might be of vastly different implementations. We feared that this could affect the benchmarking results in a negative way. We therefore created and benchmarked UDF 2, the square root function, which uses C#'s and VBA's square root method. It is fair to assume that square root is implemented on a hardware level, with a minimal amount of overhead in executing the function. It also helps us validate the results of UDF 1 (random number); if the results of the two UDF benchmarks are similar or close to, the overhead of generating a random number might not be that high, which means we can trust and use the UDF 1 results when deciding on what technology to choose. We also chose to include benchmarks for Microsoft Excel's built-in square root function, to further distinguish the overhead. In

the results of UDF 2, depicted in 4.4 it is very clear that some overhead in function calling exists. The results of 1,000,000 calls to the square root function, Microsoft Excel is 30 times faster than Excel-DNA, 51 times faster than COM Interop and 77 times faster than VBA. The benchmark results of UDF 1 and UDF 2 look very similar, which leads us to conclude that the potentially different implementations of a random number generator in C# and VBA had no noteworthy effect on the results.

In the experiments involving UDF 3, we tried to identify the overhead of data transfer when using UDFs in Microsoft Excel. Here we had some interesting benchmark results: Excel-DNA seem to start out with an overhead and gradually moves closer and closer to the mean performance of Microsoft Excel, as seen in 4.5. This is consistent with the idea that the main reason that the add-ins are losing performance, is the initial overhead of calling a function outside Microsoft Excel.

Table 4.1: UDF 1: Results for the random function

Technology and # of calls	Mean (ms)	St. dev.	Per call (ns)
Excel-DNA 10K	65.10	8.70	6510
Excel-DNA 100K	579.80	26.70	5798
Excel-DNA 500K	2470.90	94.92	4942
Excel-DNA 1 mio.	5318.80	23.63	5319
Excel COM Interop 10K	95.50	8.69	9550
Excel COM Interop 100K	775.10	13.89	7751
Excel COM Interop 500K	3939.30	103.85	7879
Excel COM Interop 1 mio.	7639.20	28.98	7639
Excel VBA 10K	130.60	18.30	13060
Excel VBA 100K	1106.60	41.69	11066
Excel VBA 500K	6510.60	691.66	13021
Excel VBA 1 mio.	10586.00	158.80	10586

Table 4.2: UDF 2: Results for the Sqrt function

Technology and # of calls	Mean (ms)	St. dev.	Per call (ns)
Excel 10K	N/A	N/A	N/A
Excel 100K	14.30	4.98	143
Excel 500K	72.70	13.47	145
Excel 1 mio.	160.20	30.32	160
Excel-DNA 10K	72.80	15.05	7280
Excel-DNA 100K	505.50	12.66	5055
Excel-DNA 500K	2473.60	12.35	4947
Excel-DNA 1 mio.	4936.40	37.30	4936
Excel COM Interop 10K	107.10	14.58	10710
Excel COM Interop 100K	848.20	10.38	8482
Excel COM Interop 500K	4162.10	34.21	8324
Excel COM Interop 1 mio.	8395.29	211.35	8395
Excel VBA 10K	141.40	17.12	14140
Excel VBA 100K	1129.50	57.09	11295
Excel VBA 500K	5733.30	301.12	11467
Excel VBA 1 mio.	12553.80	640.82	12554

Table 4.3: UDF 3: Results for the average function

Technology and # of cells	Mean (ms)	St. dev.
Excel 100K	N/A	N/A
Excel 500K	24.20	7.94
Excel 1 mio.	37.70	12.05
Excel-DNA 100K	10.80	6.46
Excel-DNA 500K	44.60	8.22
Excel-DNA 1 mio.	49.00	8.59
Excel COM Interop 100K	56.40	13.52
Excel COM Interop 500K	148.00	10.44
Excel COM Interop 1 mio.	293.10	17.10
Excel VBA 100K	35.30	10.96
Excel VBA 500K	81.10	9.94
Excel VBA 1 mio.	119.00	11.06

Chapter 5

Design

In this chapter we will define our technical and quality design goals. Based on these goals we will decide on which technology to use for developing Funsheet. Lastly, we elaborate on the system design of Funsheet in regard to our design goals.

5.1 Design Goals

We divide our design goals into technical and quality goals. We will measure the realization of the design goals in the Test chapter to verify their implementation.

5.1.1 Technical Goals

TG1 **Support for creating and calling SDFs in Funcalc from Excel.**
It should be possible to initiate the SDF creation process in Funcalc from the Excel instance. Funcalc supports a maximum of 9 arguments. Our implementation should reflect this and therefore allow the user to specify 0-9 arguments when defining an SDF. It should further be possible to call an SDF registered in Funcalc from Excel. Finally, it should be possible to use previously defined SDFs in a new SDF.

TG2 **Support for editing previously defined SDFs.**
It should be possible for a user to alter an SDF definition to update an existing

SDF. Note that this goal has not been implemented, but a possible solution is discussed in section 8.1 in chapter 8 concerning Future Work.

TG3 **Support for defining and calling closures.**

Closures are higher order functions in Funcalc allowing users to pass functions around as values. It should be possible to create and apply closures from Excel. It should be be possible to create a closure either from an SDF definition or from an existing closure. When creating closures it should be possible to make a subset of the arguments fixed.

TG4 **Support for calling built-in Microsoft Excel functions from Funsheet SDFs.**

In Microsoft Excel, users will most likely assume that the built-in functions will be available, when defining SDFs. Funcalc relies on the ability to access any function reachable from an SDF definition, but Excel provides an array of built-in functions that might not exist in Funcalc, eg. `ARABIC` (a specific Excel built-in function capable of converting a roman numeral to an arabic number). Therefore, the solution should include a mechanism to allow Funcalc to reach Excel built-in functions when they are used in the definition of an SDF.

TG5 **Support deletion of SDFs.**

If a cell containing an SDF is cleared or the formula is rendered invalid, the original SDF should be deleted. This deletion should be effective both in Funcalc and in Excel, meaning that SDF registrations in Funcalc should be removed and the possibility of calling the SDF from Excel should be removed.

TG6 **Support for specialization of closures.**

This provides performance gains in most situations, due to partial evaluation. This will, for example, allow a user to define a specialized closure, where some of the arguments are pre-compiled, so the system only has to evaluate the remaining arguments supplied at runtime.

TG7 **Support for defining recursive SDFs.**

As Funcalc already supports recursive SDFs, Funsheet should also implement support for this.

TG8 **It must be possible to save and reload SDFs.**

It should be possible to open a previously saved workbook in Excel containing SDF definitions. The SDFs should work even if they are mutually recursive.

TG9 **Support benchmarking of closures and specializations.**

There should exist a way to benchmark closures and specializations. It should be possible to select a closure or specialization and supply a desired number of calls. The benchmark feature should apply the closure or specialization the desired number of calls and present the average execution time. This should yield an accurate result that with a statistical significance depending on the number of executions.

5.1.2 Quality Goals

These goals cover all non-functional aspects of the system. The section is divided into four sections: Performance goals, maintainability goals, portability goals and usability goals.

5.1.2.1 Performance Goals

QG1 **Execution of SDFs should take at most 20% longer than execution of similar SDFs in Funcalc.**

The execution of SDFs from Excel that do not call any built-in Excel functions, should not suffer on performance.

5.1.2.2 Maintainability Goals

QG2 **Low coupling between Funsheet and Funcalc.**

This will ensure more seamless future development, for both solutions. We will strive to alter the Funcalc source code only when it is strictly necessary.

5.1.2.3 Portability Goals

QG3 **Funsheet should have a high degree of portability.**

Funsheet should be easy to distribute and install for end users. It should also

be fairly easy to distribute SDF libraries among users eg. distribution of specific SDFs within an organization.

5.1.2.4 Usability Goals

QG4 **Funsheet should add visual guidance to aid its use.**

It should be possible to view created SDFs and the generated bytecode for each. It should be possible to list the created closures. It should also be possible to get a list of the Excel functions that are currently in use through SDFs. Excel should use visual clues in order to clarify when sheets are containing SDF Definitions (function sheets). Lastly, the user interface should include support when debugging or tracing errors in the solution.

QG5 **Error handling should support the user.**

Errors should be shown in a meaningful way to the user. When possible, errors should conform to the Excel conventions for showing error, eg. if a function is called with incorrect argument types the #NAME? error should be propagated back to the user, because this is the expected behaviour of Excel. This relates to QG6, concerning conforming to Microsoft Excel.

QG6 **The solution should conform to Microsoft Excel's standards and conventions.**

All new features introduced by Funsheet, should conform to the existing user interface of Microsoft Excel and not introduce any fundamental changes. This should help flatten the learning curve of Funsheet for existing Excel users.

QG7 **All the usual Excel facilities must be available.**

Even though SDFs have been defined in a sheet, all Excel facilities should be available to the user as usual. With facilities we mean charts, tools, formula auditing, cell colors, etc.

5.2 Technology Choice

In this section we elaborate on technology options in regard to our design goals. In the end of this section we conclude on which technology we choose.

Our initial research and benchmarking indicated that we have three reasonable technology choices to base Funsheet on: VBA, COM Interop and Excel-DNA. Each comes with a set of advantages and disadvantages relating to our design goals.

Design goal QG1 states that performance is a key requirement. In the evaluation section 4.3 of the benchmark chapter, it is clear that Excel-DNA yields the best performance of the three technologies. In the results of the experiment concerning 500,000 calls of the random UDF in table 4.1, Excel-DNA is 1.6 times faster than COM Interop and 2.6 times faster than VBA. In the results of the experiment regarding 500,000 calls of the square root UDF in table 4.2, Excel-DNA is 1.7 times faster than COM Interop and 2.3 times faster than VBA. In the results of the last experiment with the Average UDF in table 4.3, Excel-DNA is 3.3 times faster than COM Interop and 1.8 times faster than VBA.

Design goal QG2 states that maintainability and low coupling is a key factor. Excel-DNA and COM Interop can both be easily used from C# (the same programming language that Funcalc is developed in). In contrast, VBA is unable to communicate directly with C#. VBA actually needs to employ COM Interop to communicate with a piece of C# code [8]. Using the same programming language across components enables a more seamless integration, since no translation or marshalling is required. Maintainability and future development might also be eased by enabling developers to use their existing skills and knowledge about C# to maintain and extend Funsheet. The fact that VBA needs to use COM Interop to integrate with Funcalc means there has to be an intermediate tier for the interface between VBA and Funcalc. This increases maintainability as it introduces an extra layer. In the portability design goal QG3 dealing with the ease of distribution and installation, a possible advantage of VBA compared to the COM Interop and Excel-DNA can be found. VBA is embedded into the Excel spreadsheet file and not contained in any external file. This can be seen as an advantage since the user simply has to open the spreadsheet and not be concerned about having the correct add-ins. We could argue that this should be seen as an advangtage, but not in Funsheet's case. The fact that Funsheet relies on Funcalc means that we at a minimum need to reference one external component (Funcalc) from the VBA code. So the idea of an encapsulated spreadsheet containing everything is not possible. Keeping the code in each spreadsheet can also turn out to be a mess in regards to maintainability (Quality goal QG2), because all released spreadsheets containing the Funsheet VBA code needs

to be changed individually or overwritten by the spreadsheet containing the updated VBA code. In contrast to this approach, COM Interop and Excel-DNA provides the option of maintaining and distributing a single add-in file from a centralized repository. This option does add a minor overhead since the user has to reference the add-in the first time Microsoft Excel is launched.

According to usability design goals QG4 and QG6, Funsheet should provide an intuitive and coherent user experience, while adding visual guidance for the features introduced in Funsheet. To achieve these goals, Funsheet need to be able to manipulate the user interface of Excel to some extent. VBA, COM Interop and Excel-DNA all offer the ability to manipulate Microsoft Excel's user interface. Excel-DNA encapsulates a lot of the user interface features from the COM Interop API to make it more available and easier to use.

Taking all the above points into consideration, we have decided to use Excel-DNA for the forthcoming implementation of Funsheet. It is the best performing technology of the three technologies, and it has the best architecture for enabling easy maintainability, portability and error handling, while still allowing for user interface manipulation.

5.3 Funsheet Design

This section covers the design of Funsheet, starting with an overview of the whole system on a component level and narrowing in and elaborating on the different components of the solution.

5.3.1 Design overview

We have identified six concerns or areas required to realize our design goals: Closures, Excel Registration, Funsheet Functions, Funcalc registration, Mapping and Conversion and User Interface, all illustrated in figure 5.1. We chose this separation to ensure a low coupling between the different concerns and a high cohesion among the components within each concern of the solution. This addresses our maintainability goal QG2 regarding future development.

The Excel Registration is concerned with the functionality responsible for SDF registration. We have collected all the functions necessary for implementing Funsheet in the Functions area. These functions are illustrated in figure 5.1 and their individual design is elaborated in section 5.3.5. We have encapsulated the functionality regarding closures in the Closures area. The Funsheet registration part is concerned with the registration of SDFs in Funcalc. The mapping and conversion area will handle the conversion of errors and values between Excel and Funcalc.

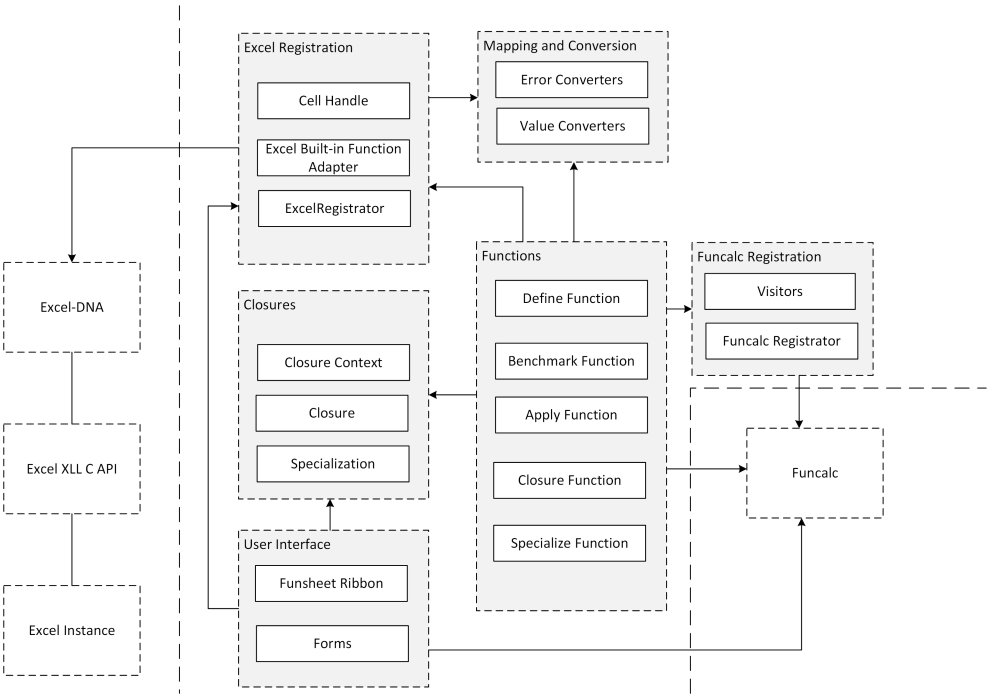

FIGURE 5.1: Overall design of Funsheet and how it integrates with Excel-DNA and Funcalc.

5.3.2 User Interface

In this section we will elaborate on our UI design.

To address design decision QG4 regarding visual guidance for Funsheet users, we will add a custom Funsheet ribbon menu in Microsoft Excel. A mockup of the ribbon user interface is shown in figure 5.2. This menu will allow the user to view the defined SDFs along with the generated bytecode. This feature already exists in Funcalc, so we believe it would be natural to include it in Funsheet. From the ribbon menu users should be able to list closures and specializations. It should also allow users to see the built-in Excel functions that are currently used by the SDFs in the workbook. Apart from providing

a sense of familiarity to former Funcalc users, this will also provide them with a general overview of the current state of the workbook.

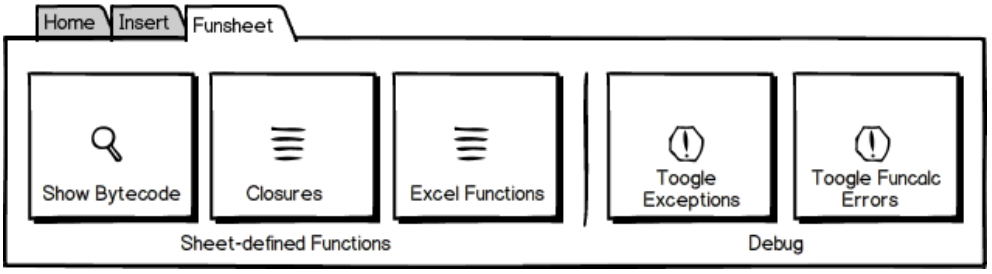

FIGURE 5.2: Mockup of the Funsheet ribbon menu.

To aid former Funsheet users and in general help users troubleshoot errors, we introduce another feature in the ribbon menu, which allows users to toggle Funcalc errors and .NET exceptions on and off. These will not be mutually exclusive, but exceptions will take precedence over Funcalc errors. This is useful in cases where the wrong type of arguments is used when calling an SDF. Microsoft Excel would give the #NAME? error, indicating that no such function exists. Funcalc would return the "#ERR: ArgType", which is more indicative of the source of the issue. Besides this scenario, the extended error handling is a positive addition to Funsheet when used as a platform for experiments, because developers and researchers can choose to have Funsheet display exceptions or the slightly more descriptive Funcalc errors when debugging or tracing problems in Funsheet.

We will refer to Excel sheets containing SDF definitions as *function sheets*. We assist users to identify function sheets, by pre-pending the sheet name with an @.

5.3.3 Monitor Cell Changes

This section addresses the system design aspects of the functionality goals that requires Funsheet to keep track of the state of specific cells. We need to keep track of cells where SDFs are defined, to be able to respond if such a cell is altered or cleared. The natural Microsoft Excel behaviour in a case where a cell is cleared, is that whatever content was there is now removed completely. So in the case where a cell contains an SDF definition that has been registered in both Funsheet and Microsoft Excel is cleared, the SDF needs to be unregistered in Funcalc and the possibility to call the SDF from Excel should

be removed. To achieve this, we considered several approaches. Here we present three alternatives, A, B and C.

5.3.3.1 Alternative A

First, we considered a polling component that is responsible for scanning through the current Microsoft Excel workbook at fixed time intervals. The polling component should have an internal representation of the workbook stored, which can be used on each scan to determine changes. This solution requires that the inner representation of the workbook is updated very frequently, which will add a performance overhead due to the excessive scanning of the workbook. A timer induced approach can also be confusing to users. Because the scans are executed asynchronously sudden changes may happen when a user is idle, or worse, when a user is in the process of editing a formula.

5.3.3.2 Alternative B

Another possible solution, also using an internal representation of the workbook, is to use Excel's C API. The API allows developers to wire a piece of code to certain events, such as key presses. Our first perception was, that cell editing in Microsoft Excel always ends with a press from the keyboard. A solution could therefore be to bind specific key presses to execute a workbook scan. This solution would avoid most futile, timer induced scans. However, this solution would require a very thorough check on many different keys, while most likely also adding many unnecessary scans. To further detract from this option, we learned that Microsoft Excel actually provides a button in the user interface for committing a cell, which is illustrated in figure 5.3.

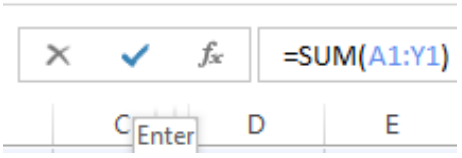

FIGURE 5.3: Microsoft Excel allows users to commit cell changes without using keyboard presses.

5.3.3.3 Alternative C

The last design option, which we ended up accepting, is to create a wrapper for Microsoft Excel's Real time Data(RTD) features. RTD is basically an implementation of the publish-subscribe pattern designed to enable cells to subscribe to an outside data source. Cells are then updated when new data is available through the publisher. A minor feature of RTD is the ability for a cell to unsubscribe from the publisher. This happens when an RTD cell is altered or cleared. Using the RTD approach will allow us to monitor cell changes with a minimum overhead, as no internal representation or thorough scans would be required.

The *Cell Handle* component in figure 5.1 represents this wrapper. We have chosen the term "handle" to avoid confusion as we do not use the RTD features for publish-subscribe in a traditional sense.

5.3.4 Error Handling

In this section we illustrate the design of the error handling in Funsheet. First we will review the different error values found in Funcalc and Microsoft Excel and describe where and how they are used. We then elaborate on how we design the conversion between errors in Funcalc and Microsoft Excel.

Microsoft Excel is equipped to show appropriate errors if formulas fail to evaluate. We need a strategy for handling errors originating from Funcalc, e.g. when Funcalc evaluates an SDF call to a Funcalc error. These errors should then be converted to an appropriate Microsoft Excel error.

Our first step was to research the various kind of errors in Funcalc and Microsoft Excel. We learned that Funcalc has errors corresponding to all Microsoft Excel errors except #NULL! and #DIV/0. An overview of all Microsoft Excel errors is shown in table 5.2 and Funcalc errors can be found in table 5.1. Funcalc introduces three new errors not found in Microsoft Excel: *argument count* error, *argument type* error and *too many arguments* error. Given this, our first notion was to research the possibility of creating custom errors, which would allow us to support both Funcalc errors together with Microsoft Excel errors. Unfortunately we discovered that this is not possible, as Microsoft Excel only provides a fixed collection of errors without any option for customization.

During our research we realized that Microsoft Excel perceives its own errors as error types as opposed to plain strings. This is important, because returning a string error message from Funcalc instead of an error, will lead to unforeseen consequences. A string error message would propagate to dependant cells as a string type in Excel instead of an error type which is the normal behaviour of Excel. Microsoft Excel also has a built-in function called IsError, which returns a boolean describing whether the input is an error or not. Using strings as errors would defeat the purpose of this function in the context of Funsheet. To conclude, returning custom errors in Excel is not supported and returning strings as errors would be extremely bad practice and interfere with Excel's calculation process.

We therefore thought of an alternative, which included returning a built-in Microsoft Excel error (#NAME?, #REF!, etc.) and then linking a comment to the cell containing the custom error message. In this way, we could support the error type convention of Microsoft Excel, while still conveying the custom error message as a comment. The downside of this is the fact that users are not used to having comments in their cells, unless they have inserted them themselves. Nor are they used to having errors displayed as comments. We must remember that Funsheet's objective is to bring new functionality to Microsoft Excel, so one can assume that potential users will be familiar with Microsoft Excel before using Funsheet. Thus, we think it would be wise to stick strictly to the conventions of Microsoft Excel. We believe there is a risk of confusing the user, if we start introducing new ways of showing errors.

We therefore decided on mapping the Funcalc errors to Microsoft Excel errors. Our decision on how to map the errors are shown in table 5.2 and table 5.1. Comparing table 5.2 containing the Microsoft Excel errors and table 5.1 containing the Funcalc errors, it is clear that there does not exists a one-to-one mapping between most errors. For example, errors such as the *#ERR: Too many arguments* error, the *#ERR: ArgCount* and the *#ERR: ArgType error* needs to be mapped to a more general Microsoft Excel error. This decision results in the loss of some information about the original Funcalc error, but we believe this to be a reasonable trade-off, e.g. we decided to map the Funcalc *ArgCount, ArgType* and *Too many arguments* errors to the Microsoft Excel #NAME? error. The rationale behind this is that when a function is called with the wrong amount of arguments and/or wrong types of arguments in Microsoft Excel, a #NAME? error is shown. This is because Excel either does not recognize the name of the function or the

types of arguments supplied to the function[9]. The complete mapping between Funcalc and Microsoft Excel errors is illustrated in figure 5.4.

As mentioned in section 5.3.2 regarding the User Interface, we implement a menu item that enables users to view the original Funcalc errors as well as .NET exceptions if the need for this should arise. This feature should only be used in troubleshooting with the aforementioned side effects in mind.

Funcalc Error	Description
#ERR: NumError	Returned in cases where a conversion to a number in Funcalc fails.
#ERR: ArgCount	Returned if a wrong number of arguments is supplied for an SDF.
#ERR: ArgType	Returned if a wrong type of argument is supplied for an SDF.
#ERR: Name	Returned if the requested SDF is not found.
#REF!	Returned if a reference is invalid.
#VALUE!	Returned if values are used the wrong way in a function.
#N/A	Returned if an indexing in a cell area is out of bounds. The #N/A error value is also used as a place holder for arguments in closures.
#ERR: Too many arguments	Returned if an SDF is called with too many arguments.

TABLE 5.1: Values and descriptions of all errors available in Funcalc.

MS Excel Error	Description
#DIV/0!	When attempting to divide by 0.
#N/A!	Missing data or either #N/A or NA() has been entered.
#NAME?	Text in a formula is not recognized.
#NULL!	Incorrect range operator used or ranges do not intersect.
#NUM!	A formula has invalid numeric data for the requested operation.
#REF!	A reference is invalid
#VALUE!	Wrong type of operand or function argument is used.

TABLE 5.2: Values and descriptions of all errors available in Microsoft Excel[10].

5.3.5 Funsheet Functions

This section explains the design of the user-defined functions which are the key enablers for integrating Funcalc in Excel. We have designed 5 UDFs to support the design goals: Define, Closure, Specialize, Apply and Benchmark. All these functions have a counterpart in Funcalc and our UDFs will serve as integration points between Excel

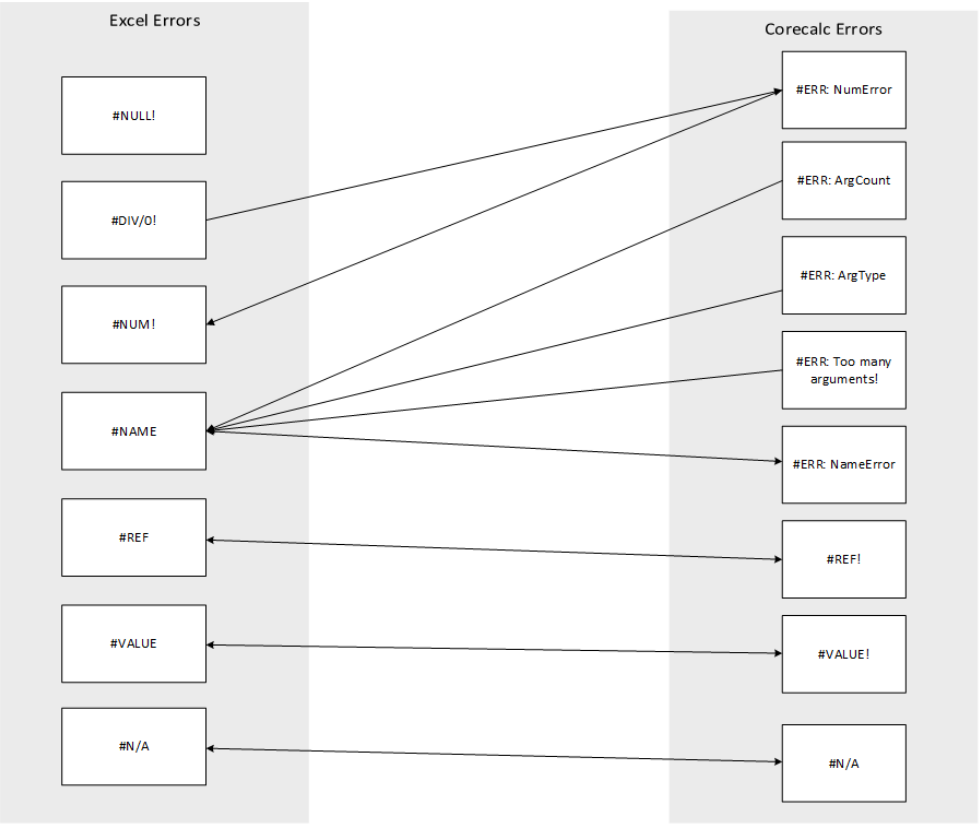

FIGURE 5.4: Mapping of Microsoft Excel errors and Funcalc errors.

and Funcalc. The Define function allows users to define new SDFs through Excel. The Closure function enables users to define new closures with the option to fix a subset of the arguments. Specialize will be able to compile a closure to obtain better performance through partial evaluation. Apply will be used to apply a closure or specialization. When using apply users will be able supply the missing arguments, if any. Benchmark will be used to benchmark closures or specializations. Benchmark accepts a reference to a closure or specialization taking no arguments (all arguments are fixed) and a number specifying the number of executions. Benchmark will then return the average execution time in nanoseconds.

5.3.5.1 Define Function

In this section we will elaborate on the design of the Define function of Funsheet. We will explain its purpose, constraints and functionality flow.

The Define function will require a minimum of two arguments: A string representing the desired SDF name and the cell reference to an output cell. In addition it will take zero to nine cell references representing arguments to the SDF. The SDF name will be subject to two constraints: It cannot be identical to any built-in Microsoft Excel functions and the name must not contain numbers. The reason is, that it is not possible to overwrite existing Microsoft Excel functions and if a function name contains numbers, e.g. *LOG10*, Microsoft Excel would misinterpret the function as a cell reference (intersection of column LOG and row 10).

The Define function will be responsible for wiring up functionality in Microsoft Excel concerning the definition of SDFs. The function will be the foundation for the rest of the Funsheet functions, since they will all rely on defined SDFs. As illustrated in the design overview figure 5.1, the Define function will draw on several areas of Funsheet; Funcalc Registration, Excel Registration and Closures, in that order. When a user will attempt to call the Define function, Funsheet will use the visitors in the Funcalc registration area to build an internal representation of all direct or transitive dependencies from the output cell of the define formula. At this point, the *Excel Bult-in Function Adapter* component will handle cases where it is necessary for the SDF to call built-in Microsoft Excel functions. After building an internal representation of the dependant cells, the SDF will be registered in Funcalc. The next step will be to register the SDF in Microsoft Excel, using the Excel Registration area. This will enable the user to access the newly created functions in Microsoft Excel. Our first design draft actually defined this as the last step of the process, but during later design iterations and during the first steps of implementation, we realized that we also needed to register a closure from the newly created SDF, as the last part of the process. The rationale for this is to enable the cell containing the SDF definition to act as a closure. We will elaborate further on this in section 5.3.5.2 regarding the closure function. The Define function flow is illustrated in figure 5.5.

5.3.5.2 Closure Function

The Closure Function will allow the creation of new closures. A closure will consist of a reference to another closure, specialization or SDF. It will further consist of a collection of arguments. These arguments can be either fixed or open. Open arguments will be

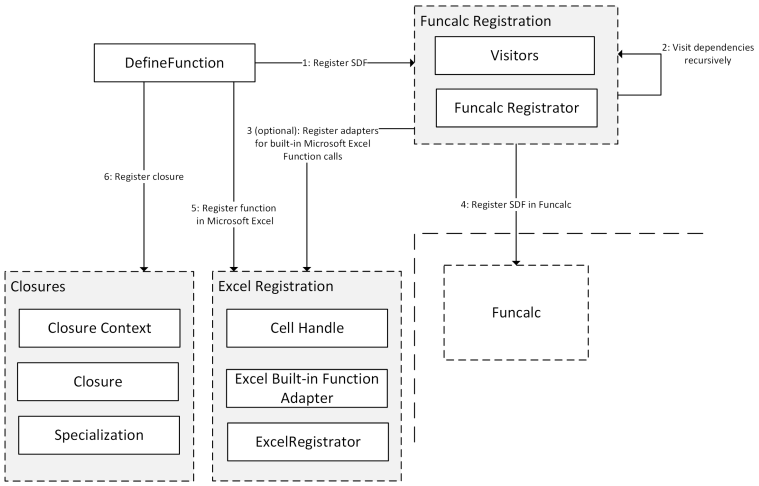

FIGURE 5.5: The flow inside Funsheet when the Define function is called.

declared by using the Excel #N/A error. This choice was made to reflect the closure function in Funcalc where the #N/A Funcalc error is used. It is important to note that although #N/A is an error, it is not used or perceived as an error in this context, but rather as an open argument. The Closure function uses the *Closure Context* component to create and save a new closure for later use.

In Funcalc, a closure can be created by supplying the SDF name as a string to the function. We have made a design decision to deviate from this convention and instead force users to supply a cell reference to the actual SDF. This decision was made because we do not have control of the dependency tree used for calculation in Excel. When a closure is defined using a string, Excel does not see any dependency from the closure to the SDF. This does not introduce problems as long as the SDF is defined before the closure is created. The problem arises when a workbook is saved, closed and re-opened. Excel will not catch the closure's required dependency on the SDF and will try to create the closure before the SDF is defined. This happens because the closure seems totally autonomous without actually being so. This problem does not exist in Funcalc because both the underlying dependencies and the calculation process can be controlled, as it is a pure C# implementation. Besides eliminating the problem we think that this design decision better aligns with the Microsoft Excel user experience where dependencies always are created using references.

To illustrate the design described above, consider a simple SDF called TwoSum, capable of adding two numbers and returning the result. The SDF is defined in the cell A1 and

accepts two arguments, both numbers. We can then use the following formula to create a closure from the SDF, where the first argument is fixed(5) and the second argument is open.

=CLOSURE(A1;5;NA())

To execute a closure, the Apply function (see 5.3.5.4) should be used.

5.3.5.3 Specialize Function

The Specialize function is used to compile a closure to yield better performance. The following formula shows how to specialize a closure if the closure is created in the cell A1:

=SPECIALIZE(A1)

When a closure is specialized, a new SDF will be created in Funcalc, but the SDF will not be registered as a callable function in Excel, which is the case when using the Define function. To execute a specialization, the Apply function (see 5.3.5.4) should be used. The *Closure Context* component will be responsible for registering the specialization and instructing Funcalc to compile the closure to an SDF.

5.3.5.4 Apply Function

The Apply function will accept a cell reference to an SDF, closure or specialization along with a collection of arguments to apply to the execution of the respective SDF, closure or specialization.

The Apply function will use the *Closure Context* component to obtain a closure or specialization. In the case where a cell reference to an SDF definition is supplied, the *Closure Context* component will be able to find a matching closure, because of the design decision to create a default closure when using the Define function as explained in section 5.3.5.1.

We will then delegate to the Apply function implemented by Funcalc, obtain the result from the Funcalc Apply function and present it to the user.

The Apply function will be designed and marked as thread-safe to allow Excel to use multiple processors in the calculation of formulas containing the Apply function.

5.3.5.5 Benchmark Function

The Benchmark function will take two arguments, the first of which is a closure, specialization or SDF cell reference. The second which is a number. It is important to note that the first argument needs to reference a function where all arguments have been fixed. This means that if the supplied argument references an SDF, the SDF must not accept any input (must be defined with zero input arguments). If the first argument to the Benchmark function is a closure or specialization all argument must be fixed to some value (no open arguments using #N/A). The number indicates the number of times the Benchmark function should execute the SDF, closure or specialization. The Benchmark function will return the average execution time in nanoseconds.

The Benchmark function will use the *Closure Context* to obtain a closure or specialization. It will then delegate to the Benchmark function of Funcalc, which will run the actual benchmark routine and return the average execution time. The result will be displayed to the user in Excel.

The Benchmark function is designed to disallow simultaneous calculations of the functions in Excel. This does not prevent Excel from doing other calculations parallel to the Benchmark function, but prevents more than one benchmark to run at a single point in time, which wouldn't be desirable. This is not fool-proof but, if it is important to reduce the "noise" created by other calculations, Excel can be configured to only do calculations on a single thread. To enable single-threaded calculation of formulas in Excel 2013, go to **File** and select **Options**. Select the **Advanced** tab. Uncheck the option **Enable multi-threaded calculation** as shown in figure 5.6.

5.4 Design Evaluation

In this section we evaluate the design of Funsheet in regards to the design goals stated in section 5.1. Starting out, we had several options for choosing a technology: VBA, COM Interop and Excel-DNA. We discussed their advantages and disadvantages in regards to

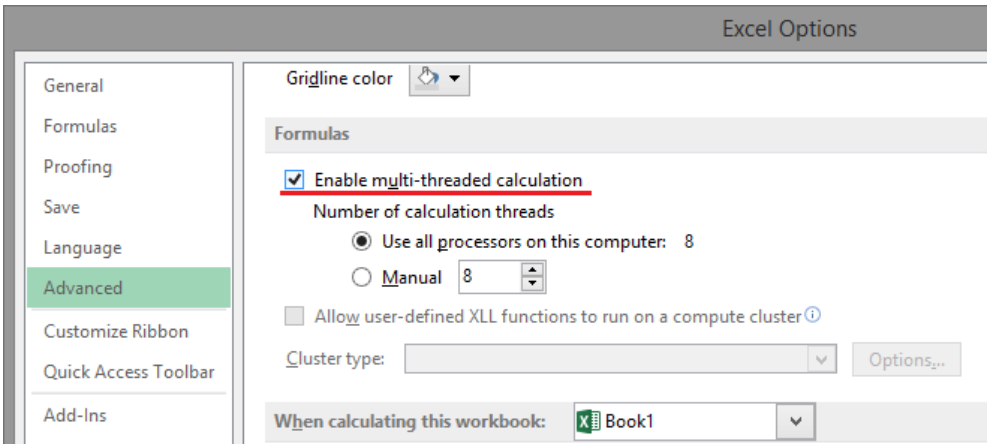

FIGURE 5.6: Enable or disable multi-threaded calculation in Excel 2013.

our design goals, and ended up choosing Excel-DNA as the technology to base Funsheet upon. From experimenting with the different technologies in chapter 4, we knew Excel-DNA was the best performing technology of the three. Hence choosing Excel-DNA was a high level design decisions we believe will lay the foundation for achieving design QG1 regarding performance.

All technical design goals except TG4, are aiming on using existing functionality from Funcalc. During the design process, we learned that different constraints apply when using Microsoft Excel as a frontend as opposed to the frontend of Funcalc. Funcalc has full control over its frontend, whereas Funsheet's control over Microsoft Excel is limited to the functionality of the Excel C API (and in some cases the COM Interop API). We strove to adapt the design of Funsheet to take these differences into account. An example of this is the design of the closure function, which was changed to comply with Excel's way of enforcing the dependency tree. We elaborate on the closure function in section 5.3.5.2. Design goal *TG4: Support for calling built-in Microsoft Excel functions from Funsheet SDFs* was an addition to the functionality provided by Funcalc that allows users to use built-in Excel functions in SDFs and it allows the users to substitute built-in Funcalc functions with their Excel counterpart, e.g. using `SUM` from Excel instead of the `SUM` from Funcalc.

We strove to address the technology agnostic design goal of maintainability, by separating concerns in the high level design and making it an explicit part of a goal (QG2) to alter as little of the source code of Funcalc as possible.

Our usability goals were all defined with a desire to conform to Microsoft Excel's standards and conventions. Funcalc error values without an obvious Excel counterpart were mapped to the most suitable existing Excel error and visual guidance was designed in the form of a ribbon menu in Excel, to maintain the Excel user experience.

Chapter 6

Implementation

In this section we describe our implementation of Funsheet. We start with an overview and general explanation of the solution. Then we go through our implementation in regards to achieving high portability and low coupling. Here we cover how Excel-DNA helps us to achieve a high degree of portability and maintainability and how we have minimized the number of changes to the Funcalc source code to allow the same code base to be used, both as a stand-alone application and as a Funsheet dependency. Then we will explain how we have implemented the conversion of values and errors between Funcalc and Excel. Next we explain how we have implemented the Real Time Data (RTD) wrapper, capable of acting when an associated cell is cleared. Finally, we explain how we implemented each of the user-defined functions of Funsheet.

6.1 System overview

This section gives an overview of the system at implementation level. All classes and their mutual relationships are illustrated in figure 6.1. A unidirectional association is depicted using a line ending in an arrow. A bidirectional association is depicted using a line with arrows in both ends. An aggregation is depicted using a line with a diamond in one end and an arrow in the opposite. Note that not all relationships are illustrated in figure 6.1, to avoid bloating the diagram. For example, the `HelperMethods`, `ErrorConverter` and `ExtensionMethods` classes are used excessively throughout the implementation. Also,

the user interface forms are shown as a single class, although we have implemented 4 forms to support the user interface.

The `ExcelCalcInitializer` class is used to initialize the workbook when a saved workbook is opened in Excel. The `RegisteredSdfContext` and `RegisteredSdf` classes are used to keep track of SDFs defined through Funsheet. All other classes within the *Functions* area contain the implementations of the Funsheet UDFs. We elaborate on each of these implementations later in this chapter. The `ExcelRegistrator` handles registration of SDFs as callable UDFs in Excel, at runtime. The `ExcelFunCallContext` and the `ExcelFunCall` are used to integrate built-in Excel functions into Funcalc, to make them available when defining SDFs. The `ExcelCellHandle` is our implementation of an Real Time Data (RTD) wrapper. The `CorecalcRegistrator` is responsible for registering SDFs in Funcalc and the `CorecalcDynamicCellVisitor` and `CorecalcDynamicExprVisitor` are used to traverse the dependency tree of an SDF definition in Excel and create a Funcalc representation from this. The `ClosureContext` and `Closure` classes are responsible for creating and registering closures. The `HelperMethods`, `ExtensionMethods` and `ErrorConverter` are utility classes that provide static methods used in all parts of Funsheet. `ExcelcalcRibbon` and `Forms` classes are used to provide a user interface for the user in Microsoft Excel. We elaborate on the specific forms in section 6.7.

6.2 Portability, Maintainability and Coupling Considerations

In this section we will cover some of the more important decisions that we made in regards to portability, maintainability and coupling, during implementation.

Excel-DNA is contributing to an increased portability as it contains a tool called `ExcelDNAPack`[11], which allows developers to pack resources directly related to an Excel-DNA plugin, into a single file. In our case this means that all Excel-DNA related references and resources are packed into one file. We have chosen to supply the Funcalc binary and the DLLs responsible for reverse engineer methods into bytecode. This decision was made to allow developers to update Funcalc and the bytecode generation assemblies without having to redistribute Funsheet. Besides that we also include the `Microsoft.Office.Interop.Excel`

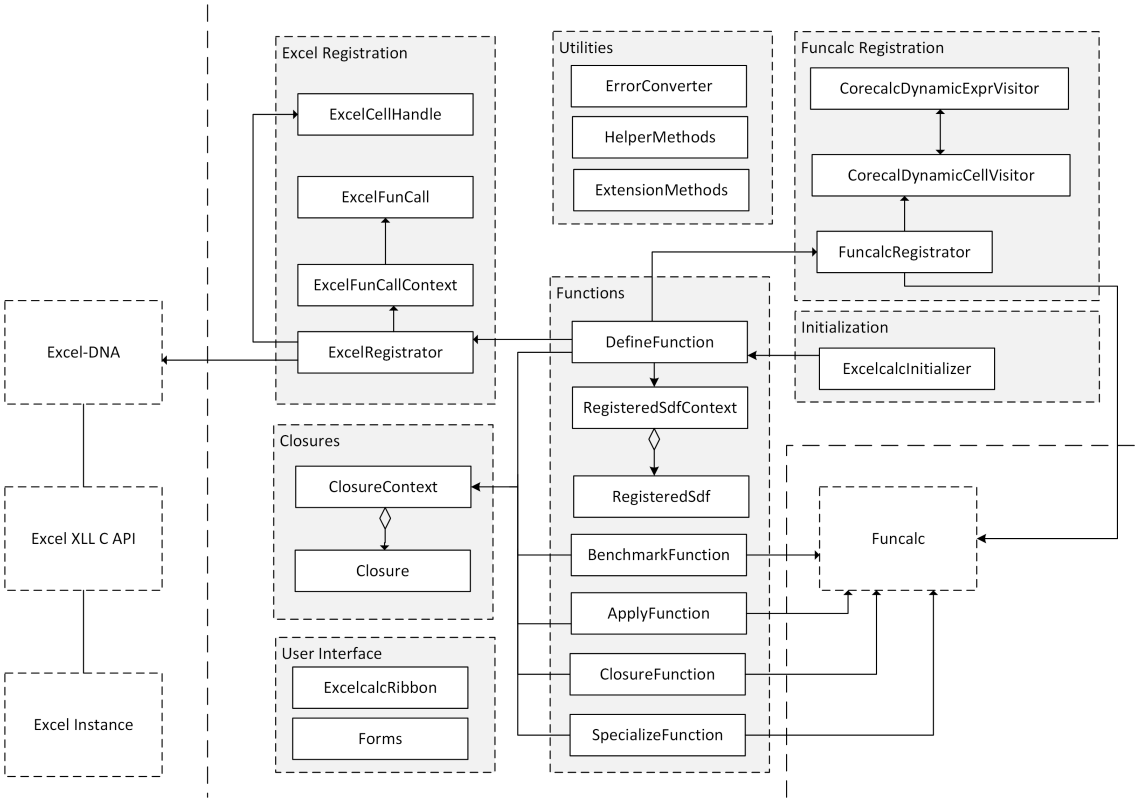

FIGURE 6.1: Class diagram of Funsheet.

DLL to be capable of responding to COM Interop events, as described in 6.6.2.8. This makes the Funsheet add-in highly portable. To install Funsheet refer to the installation guide in Appendix C.

We started implementing Funsheet with a desire to alter as little of the Funcalc source code as possible. After becoming acquainted with the Funcalc source code during the initial steps of this project, it occurred to us that we required looser access modifiers on the Funcalc members to achieve our goals of integrating Funcalc with Microsoft Excel without making major changes to the Funcalc source. So instead of identifying and changing all the necessary access modifiers inside the Funcalc source code, we decided to add the `InternalsVisibleTo`-attribute to the `AssemblyInfo.cs`-file of Funcalc, as illustrated in listing 6.1. By adding this attribute, all Funcalc members with the internal access modifier are exposed to Funsheet. This will allow better maintainability and further development of both Funsheet and Funcalc. A couple of small code changes were made to the Funcalc `Function` and `FunctionInfo` classes to allow overwriting of a built-in Funcalc function, e.g. if a user wishes to use the Excel built-in `SUM` instead of the one from Funcalc. These changes does not break Funcalc. A minor change to the existing

Funcalc grammar (described in section B.2.3) was implemented, to support punctuations in function names. All changes made to Funcalc including the aforementioned can be found in the change log in Appendix B.

```
1 [assembly: InternalsVisibleTo("ExcelCalc")]
```

LISTING 6.1: InternalsVisibleTo attribute added to the AssemblyInfo file of Funcalc to expose internals to Funsheet.

6.3 Value Conversion

In this section we will describe the implementation of value conversion and error conversion in Funsheet. We will describe the data types that are converted from Funcalc types and returned to Microsoft Excel as *Excel Types*. Data received in Funsheet from Microsoft Excel needs to be type cast and converted at runtime. To achieve this, we have implemented a helper method `ObjectToValue` that takes the ultimate base class `object` as an argument, type casts it and creates and returns its corresponding Funcalc value. We know what data we possible can receive from Microsoft Excel[12] in Excel-DNA and have designed the method accordingly. The helper method returns corresponding Funcalc values for the following C# data types: `double`, `string`, `object[,]` (a two dimensional object array), `ExcelReference` and the following Excel-DNA `ExcelError` enumerations: `ExcelError`, `ExcelMissing` and `ExcelEmpty`. The reverse conversion is implemented as an extension method that converts Value types and returns the corresponding Microsoft Excel types. An overview of the two-way conversion is illustrated in figure 6.2. There exists a bidirectional conversion relationships between all types, except for ExcelEmpty and `null`, which is a one-way conversion. This is because there are now null like Value types in Funcalc. Thus we are assuming to always receive a non-null Value type which can be converted and returned as a C# type.

6.4 Error Handling

In this section we will elaborate on the implementation of error handling in Funsheet, which is based on the error mapping conventions described in section 5.3.4 of the design chapter. Our initial implementation of the error handling consisted of two conversion

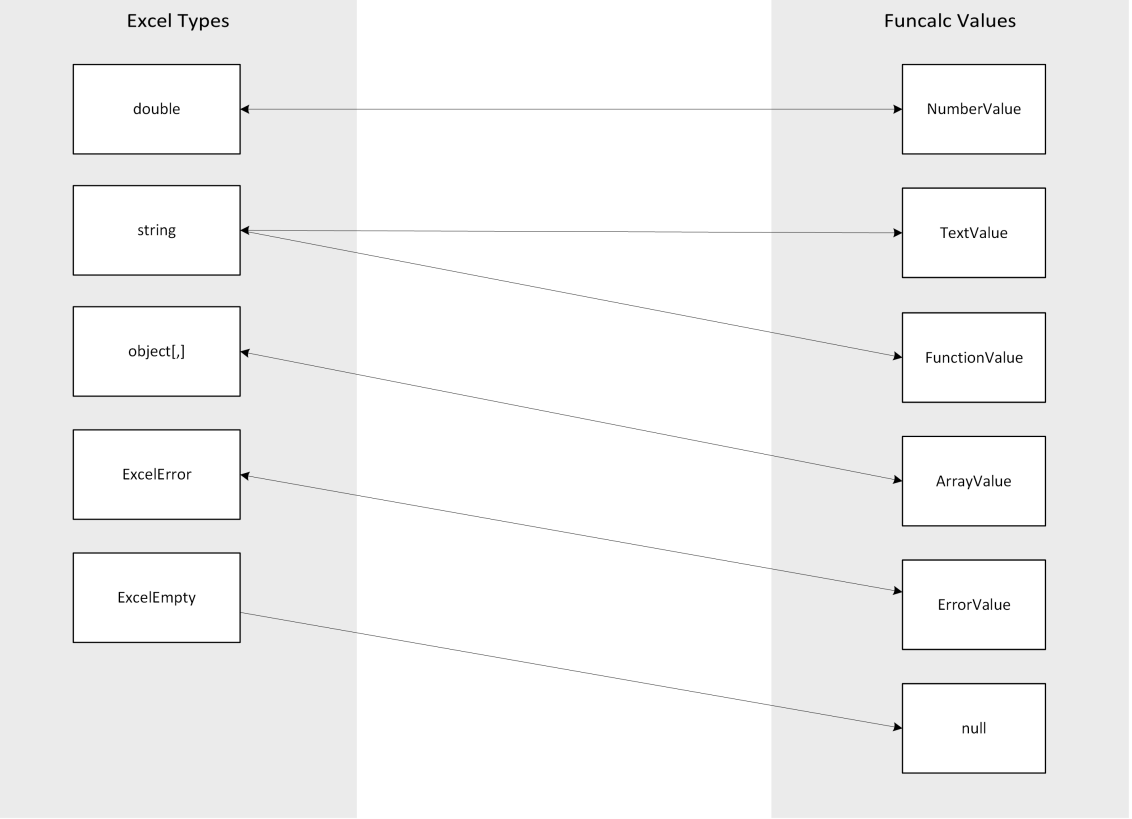

FIGURE 6.2: Value conversion in Funsheet between Microsoft Excel and Funcalc types.

methods located inside the `HelperMethods`-class (cf. figure 6.1). One method to convert from Microsoft Excel errors to Funcalc errors and another method for the reverse conversion. Each of the methods consisted of a series of conditionals, each conditional concerned with one type of error. This led to some very bloated methods. We decided to re-factor it into the separate class `ErrorConverter`, which is only concerned with error conversion. This class contains two dictionaries that are responsible for mapping from Microsoft Excel errors to Funcalc errors and vice versa. It is dependant on the Funcalc `ErrorValue` class, which is a subclass of Value. The `ErrorValue` class provides a public static `ErrorValue` property for each of the errors listed in table 5.1. The convert mechanic is implemented with two overloaded methods, one for each type of conversion.

In section 5.3.2 of the design chapter we stated that we wanted to add the opportunity for users to swap between errors being shown as Microsoft Excel errors, Funcalc errors and .NET exceptions. This has been implemented in the value converter extension method of the Value type (described in section 6.3). Funsheet's value conversion method `ToExcelObject` will use the default setting of the ribbon menu for determining which

error to show. If the ribbon menu is toggled to showing Funcalc errors, the Funcalc error will be converted into a string representation instead of a corresponding Microsoft Excel error. This logic is illustrated in listing 6.2. The ribbon menu also allows the display of .NET exceptions. The handling of this particular setting is distributed to Excel-DNA's `RegisterUnhandledExceptionHandler` method, to which we pass a delegate that returns the unhandled exceptions a string.

```
1  if (value is ErrorValue)
2  {
3          if (ExcelcalcRibbon.ShowFuncalcErrors)
4          {
5                  return value.ToString();
6          }
7          else
8                  {
9                  return ErrorConverter.Convert((ErrorValue)value);
10             }
11 }
```

LISTING 6.2: Snippet from the value converter extension method of the Value type, showing how Funsheet determines what errors to show, based on the state of the ribbon menu in Microsoft Excel.

6.5 RTD Wrapper

The Real Time Data (RTD) wrapper (design explained in section 5.3.3.3) is used to handle the case where a user clears a cell containing a DEFINE-, CLOSURE- or SPECIALIZE-formula. When such a cell is cleared it is picked up by Funsheet and suitable actions are carried out, e.g. when a DEFINE-formula is removed, the SDF registration in Funcalc is deleted and the ability to call the SDF from Excel is removed.

The RTD wrapper is implemented in the class `ExcelCellHandle`. It implements the `IExcelObservable` interface from the Excel-DNA assembly. To create an RTD wrapper the static method `CreateHandle` is used. The `CreateHandle` method is shown in listing 6.3. The method returns a handle based on a function name, e.g. DEFINE and the parameters used to call it. We chose to only use the SDF name, closure name or specialization name as parameter. It further needs an `ExcelFunc` which is a delegate that returns an `object`. This delegate is used to produce the string that the user should see in Excel, e.g. `FUN GOALSEEK AT #1` for an SDF called GOALSEEK. Finally it accepts an `IDisposable` which is the most interesting part of the RTD wrapper. The

`Dispose` method of this object is called when the cell containing the RTD wrapper is cleared.

```
1 public static object CreateHandle(string callerFunctionName, object
      callerParameters, ExcelFunc getReturnValue, ExcelAction onDispose)
2 {
3     return ExcelAsyncUtil.Observe(callerFunctionName, callerParameters, () =>
      new ExcelCellHandle(getReturnValue(), onDispose));
4 }
```

Listing 6.3: Static method used to generate a cell handle to return from a UDF.

When the RTD wrapper is returned to Excel from a user-defined function (UDF), Excel invokes the `Subscribe` method of the RTD wrapper, as shown in listing 6.4. This method calls the `OnNext` method of the underlying observable to provide the string that should be shown in Excel. When using the publish-subscribe pattern, the `OnNext` method could be called on a regular basis to update the value of the associated cell, but in our case we are only interested in providing a string value to the cell once. Finally the subscribe method returns a delegate wrapped in an `IDisposable`. This delegate is called when the cell is cleared.

```
1 public IDisposable Subscribe(IExcelObserver observer)
2 {
3     observer.OnNext(_returnValue);
4     return new SdfDisposer(() =>_disposeAction());
5 }
```

Listing 6.4: The `Subscribe` method is called from Excel when the RTD wrapper is written to a cell.

Listing 6.5 shows how the RTD wrapper is created in the Define UDF and returned to Excel. Note that because we use delegates to provide the return value and the dispose action, the respective functions are able to provide their own values and actions depending on their inputs and responsibilities.

```
1 private static object CreateCellObserver(string sdfName)
2 {
3     var cellText = RegisteredSdfContext.Sdfs[sdfName].SdfCellText;
4     return ExcelCellHandle.CreateHandle("DEFINE", sdfName, () => cellText, () =>
      UnregisterSdf(sdfName));
5 }
```

Listing 6.5: Creating and returning an RTD wrapper from the Define UDF.

6.6 Funsheet Functions

In the following sections we will describe the implementation details of the Funsheet UDFs. We start by explained how we use Excel-DNA to expose C# code to Excel. Next we explain the key parts of the implemented UDFs. For readers who want even more details on the implementation, refer to the source code documentation in Appendix A.

6.6.1 Excel-DNA usage

Excel-DNA makes it very easy to expose user-defined functions (UDF) written in C# to Excel, but a few rules must be followed. The class containing the methods representing the UDFs must be `static`. The methods representing UDFs must also be declared `static` themselves. Further a return type should be defined in the method signature, otherwise the method will be perceived as a command by Excel-DNA.

To mark a method as an UDF all that is needed is to decorate the method with the custom attribute `ExcelFunction` as shown in listing 6.6.

```
1 public static class SpecializeFunction
2 {
3     [ExcelFunction(Name = "SPECIALIZE", IsMacroType = false)]
4     public static object Specialize(string closureName)
5     {...
```

LISTING 6.6: Exposing a method as an UDF using the `ExcelFunction` custom attribute

In listing 6.6 some named parameters have been supplied to the custom attribute. The following list describes the named parameters of the `ExcelFunction`-attribute and their influence on the resulting Excel UDF.

- **Name** is used to specify the name of the UDF when used from Excel.
- **IsVolatile** is used to specify whether the UDF should be registered as volatile in Microsoft Excel.
- **IsMacroType** should be true if the UDF requires to read or manipulate data in Excel, other than what is returned from the UDF call. E.g. if some cells should be coloured.

- **IsThreadSafe** is used to avoid race conditions from happening in this UDF. When set to false only one thread can operate inside this UDF at a time.

An option when defining input parameters for a method representing an UDF is to use the `ExcelArgument`-attribute with its named parameter `AllowReference` set to true. This will let Excel-DNA pass an `ExcelReference` to the method which is a more versatile approach than just getting e.g. a string. An instance of `ExcelReference` contains information about the row, column, the value of the cell and more. An example of how to use the `ExcelArgument`-attribute is shown in listing 6.7

```
1 [ExcelFunction(Name = "DEFINE", IsMacroType = true)]
2 public static object Define(
3     string sdfName,
4     [ExcelArgument(AllowReference = true)]object output,
5     ...
```

LISTING 6.7: Using the `ExcelArgument`.

One important thing to note is that if a method is marked as a macro type and have at least one input parameter declared with the `ExcelArgument`'s `AllowReference` set to true, the UDF will be registered as volatile in Excel. This is an unfortunate side-effect that will be mentioned in section 6.6.2.

6.6.2 Define

The `Define` method of the `DefineFunction` class is used when the Define UDF is called from Excel. It is responsible for registering SDFs in Funcalc and in Microsoft Excel. It accepts a string containing the name of the SDF as the first argument, then an output cell and hereafter zero to nine input cells. The `Define` method has the `IsMacroType` attribute set to true, to allow it to access data in the Excel sheets. The `IsVolatile` is set to false, because the Define UDF should only be recalculated when one of its precedents is changed. The `IsThreadSafe` attribute is set to false, because we expect the user to define SDFs sequential and we need the UDF to be sequentially called if it happens when opening a workbook. This is discussed in section 6.6.2.8. The Define UDF will be used differently depending on the circumstances. We basically have two scenarios. One where the `DEFINE`-formula is typed in by a user and committed to Excel and one where an existing workbook containing `DEFINE`-formulas is opened. In the first part of

this section we will assume the first scenario. The latter scenario will be described in subsection 6.6.2.8.

The following sections describe how the Define UDF work. We describe the steps the UDF performs. These are: obtaining the formula of the cell from where the Define UDF was typed in and creating `ExcelReference` instances representing the output cell and the input cells; The population of a Funcalc representation of the workbook using the output cell `ExcelReference`; The registration of the SDF in Funcalc; The registering the SDF as a callable UDF in Excel; The creation of a `Closure` from the SDF; The promotion of a sheet to a function sheet; Returning an instance of our RTD wrapper to Microsoft Excel.

We then explain how the Define UDF deals with built-in Excel functions, and finally, we explain how we managed to respond to users, re-opening a previously saved workbook. Why this scenario needs explaining is not clear at first, but imagine a user, defining two SDFs, one dependent on the other. When saving the workbook and re-opening it, which SDF should be processed first? Because the dependencies among these live in Funsheet, there are no guarantees that Excel will build our desired dependency tree, but we explain how we overcame this.

6.6.2.1 Step 1: Obtaining cell formula and references from a cell formula

This section describes how we use Excel-DNA to obtain the formula of the cell in which the `DEFINE` UDF is typed in. Then, it describes how we parse the formula to a Funcalc `Cell`. Next, we explain how we create `ExcelReference` instances for the output cell and each of the input cells.

To obtain the cell formula we use the Excel C API through Excel-DNA. Excel-DNA exposes a method, `Excel`, on the `XlCall` class. This method accepts an integer, representing a function in the Excel C API, and an array of arguments. The `XlCall` class has static members for each function in the C API to make it easier to read and use. To obtain the formula the we first obtain an ExcelReference, representing the calling the UDF, using `XlCall.xlfCaller`(see listing 6.8). This returns an ExcelReference to the calling cell. From this we obtain the formula by using `XlCall.xlfGetFormula` and pass the `ExcelReference` instance as the second argument. We then use the `Parser` class

from Funcalc to produce a Funcalc `Formula` instance. The `Formula` class is a type of cell in Funcalc. The `Formula` contains all the arguments passed to the UDF in Excel as `CellRef` instances, which is another Funcalc type. From the information stored in each `CellRef`(output and inputs) of the `Formula` instance, we create `ExcelReference` instances as illustrated in listing 6.9.

Our first implementation did not rely on the Funcalc cell parser to create `ExcelReference` instances from the output and input cells. Instead the input parameters in the *Define* method were decorated with the `ExcelArgument` attribute and its `AllowReference` set to true. This turned the input arguments into `ExcelReference` instances before entering the method. During implementation we learned that the combination of using the `ExcelFunction` attribute with `IsMacroType` set to true and decorating at least one parameter with the `ExcelArgument` with the `AllowReference` set to true, will result in the UDF to become volatile. We resolved this by implementing the solution as described above, retrieving `ExcelReference` instances using the Funcalc parser.

```
1 var callingCell = XlCall.Excel(XlCall.xlfCaller) as ExcelReference;
2 var fullFormula = XlCall.Excel(XlCall.xlfGetFormula, callingCell) as string;
```

LISTING 6.8: Snippet from the `Define` function showing how an `ExcelReference` of the calling cell is created and then used to retrieve the formula of the cell. The `xlfCaller` and `xlfGetFormula` are integers indicating what function to call in the Excel C API.

```
1 var cellRef = expr.es[1] as CellRef;
2 int row = cellRef.raref.rowAbs ? cellRef.raref.rowRef : callingCell.RowFirst +
      cellRef.raref.rowRef;
3 int col = cellRef.raref.colAbs ? cellRef.raref.colRef : callingCell.ColumnFirst
      + cellRef.raref.colRef;
4
5 var outputRef = new ExcelReference(row, col);
```

LISTING 6.9: Snippet from the `Define` function showing how an `ExcelReference` of the output cell is constructed. `cellRef` refers to the second argument of the Funcalc `Formula`. The rows and columns are determined based on either absolute or relative references.

6.6.2.2 Step 2: Creating Funcalc representation and registering the SDF

In this section we describe how we visit all cells directly or transitively dependant on the output cell using the `CorecalcDynamicCellVisitor` and `CorecalcDynamicExprVisitor` classes. The two visitor classes are illustrated in a class diagram in figure 6.3. We also describe how the registration of the SDF in Funcalc is implemented.

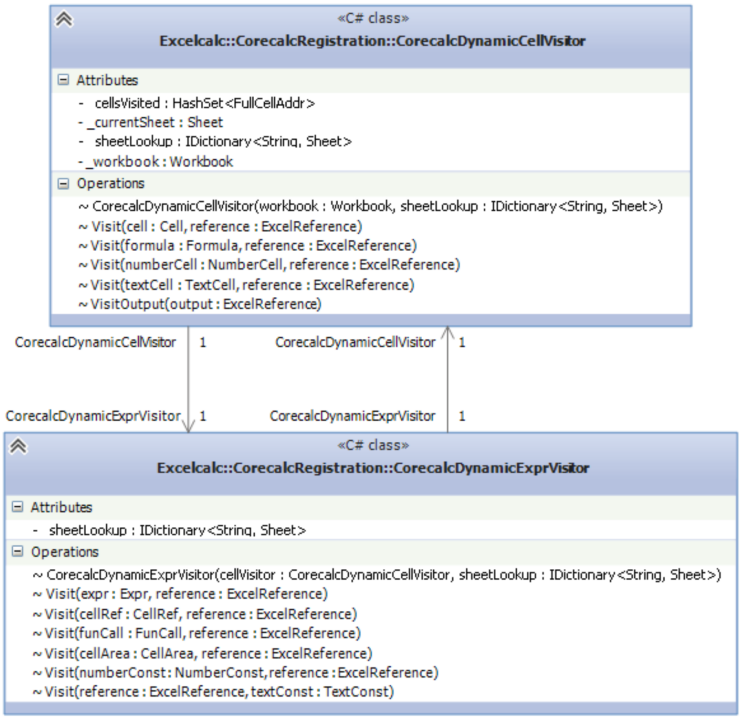

FIGURE 6.3: Design Class diagram illustrating the two visitors, the `CorecalyDynamicCellVisitor` and the `CorecalcDynamicExprVisitor`.

`Define` method calls the `RegisterSdf` on the `CorecalcRegistrator` passing the SDF name, reference to the output cell, references to the input cells and a reference to the calling cell. In the `RegisterSdf` the `CorecalcDynamicCellVisitor`'s `VisitOutput` method is invoked passing in the `ExcelReference` of the output cell. The visitor recursively builds a representation of the Excel workbook, containing only cells directly or transitively reachable from the output cell. The representation is stored in a Funcalc `Workbook` instance. The general process of the visitor is to recursively retrieve cell content, parse cell content and then visit precedents. The cell content is obtained and parsed the same way as described in section 6.6.2.1. We have implemented memoization using a HashSet containing the cell addresses of all visited cells to avoid exponential running times when visiting the cells. To determine which overloaded `Visit` method to use, our first implementation consisted of a series of conditionals. This was very prone to becoming bloated and one can imagine that if more `Cell`-types are added in the future, maintainability drops. To avoid this, we refactored the solution to use C#'s `dynamic` keyword for achieving runtime binding. An example of this is illustrated in listing 6.10. The late binding adds a small performance overhead, but enables for a more elegant and maintainable solution. When the `CorecalcDynamicCellVisitor` encounters either

a `NumberCell` or a `TextCell` it adds the cell to the internal workbook representation. If a cell is visited and parsed into a Funcalc `Formula` type, each expression in `Formula` is visited using the `CorecalcDynamicExprVisitor`.

After all relevant cells have been visited, the `CorecalcRegistrator` passes the Funcalc `Workbook` to the `SdfManager` in Funcalc. The `SdfManager` creates the new SDF based on information in the `Workbook`. Hereafter, the `Define` method calls the `RegisterSdf` on the `ExcelRegistrator` class, passing the SDF name, output `ExcelReference` and the list of input `ExcelReferences` as parameters. The `RegisterSdf` method then dynamically builds a delegate from the parameters, using the LINQ Expression API, which is used for runtime code generation. In our case, we use it to assemble the delegate dynamically to support a variable amounts of arguments for defining SDFs. The generated delegate is then registered in Microsoft Excel using the `RegisterDelegates` method of the `ExcelIntegration` class in Excel-DNA. This method accepts a list of delegates and a list of `ExcelFunctionAttribute` instances which contain the meta-data for each delegate(name, volatility, etc.).

```
1  internal void Visit(Cell cell, ExcelReference reference)
2  {
3          //Set current sheet
4      _currentSheet = _sheetLookup[reference.SheetId.ToString()];
5
6          //Dynamic redirect
7      Visit((dynamic)cell, reference);
8  }
9
10 internal void Visit(NumberCell numberCell, ExcelReference reference)
11 {
12     _currentSheet.SetCell(numberCell, reference.ColumnFirst, reference.RowFirst);
13     numberCell.MarkDirty();
14 }
```

LISTING 6.10: The top `Visit` method for visiting the abstract Funcalc class Cell uses the `dynamic` keyword to determine cell type at runtime. If the cell has been resolved to a `NumberCell` (inheriting from `Cell`) the bottom `Visit` method will automatically be called.

6.6.2.3 Step 3: Creating a default closure

After registering the SDF we chose to create a default closure where all arguments are open. The rationale of this decision is discussed in section 5.3.5.1. We implemented using the `RegisterClosure` method of the `ClosureContext` class and providing an `ExcelErrorNA` for each input, as shown in listing 6.11. The implementation of `ClosureContext` is explained in section 6.6.3.

```
1 //Register default closure for the SDF.
2 var closureNaInputs = Enumerable.Repeat((object)ExcelError.ExcelErrorNA,
      inputs.Count).ToArray();
3 ClosureContext.RegisterClosure(sdfCellText, closureNaInputs, false);
```

Listing 6.11: Creation of a default closure for the new SDF.

6.6.2.4 Step 4: Promoting to a function sheet

When an SDF is defined the sheet from which it is defined should be promoted to a function sheet, if this has not been done already. To indicate that a sheet is a function sheet we prepend the sheet name with an @. To accomplish this we execute an asynchronous macro from the `Define` method. We use an extension method on the calling cell's ExcelReference to obtain the sheet name. Then we execute a call to the Excel C API to rename the respective sheet, as shown in listing 6.12. We use an asynchronous macro because the `xlcWorkbookName` function cannot be called from an UDF, although the `IsMacroType` is set to true. The call must be executed from a true macro.

```
1  private static void PromoteToFunctionSheet(ExcelReference caller)
2  {
3      var sheet = caller.GetSheetName();
4
5      if (!sheet.StartsWith("@"))
6      {
7          ExcelAsyncUtil.QueueAsMacro(delegate
8          {
9              XlCall.Excel(XlCall.xlcWorkbookName, sheet, "@" + sheet);
10         });
11     }
12 }
```

Listing 6.12: Promoting a sheet to a function sheet

6.6.2.5 Step 5: Returning an RTD wrapper

Last, an RTD wrapper is created using the static `CreateHandle` method of the `ExcelCellHandle` class. The SDF is passed as a parameter for identification. This ensures that a new wrapper is not created if the same SDF is defined in several cells (this is more relevant for closures). The text that should be shown in the cell is passed into the method and a delegate capable of executing the `Unregister` method is passed in. The `UnregisterSdf` method will be called when the last cell containing this wrapper is cleared.

6.6.2.6 Un-registering an SDF

When the RTD wrapper triggers the `UnregisterSdf` method of the `DefineFunction` class the following actions are carried out: removal of the SDF in Funcalc, un-registering of the SDF in Excel and removal of the closures associated with the SDF.

Complete removal of the SDF, as a callable UDF, in Excel is not entirely possible, but we have a routine that almost accomplishes a complete removal. The SDF can be removed from Excel by calling `xlfUnregister` from the C API and passing in the registration id of the SDF. The registration id is an internal id issued by Excel. To obtain this id we call the `xlfEvaluate` function of the C API and passing in the SDF name as a parameter. The problem with this approach is that the SDF is still callable from Excel, but always return the #VALUE Excel error. To overcome this we re-register a hidden macro with the same name as the SDF and afterwards un-registers this. The SDF will still be visible to VBA code (although unusable), but is otherwise un-registered. Listing 6.13 shows the implementation of this.

```
1  internal static void UnregisterSdf(string sdfName)
2  {
3      ExcelAsyncUtil.QueueAsMacro(() =>
4      {
5          //Get XLL path
6          var xllName = XlCall.Excel(XlCall.xlGetName);
7
8          //Get SDF reg id and unregister
9          var regId = XlCall.Excel(XlCall.xlfEvaluate, sdfName);
10         XlCall.Excel(XlCall.xlfSetName, sdfName);
11         XlCall.Excel(XlCall.xlfUnregister, regId);
12
13         //Reregister SDF as hidden and unregister to clear the function wizard
14         var reregId = XlCall.Excel(XlCall.xlfRegister, xllName, "xlAutoRemove",
       "I", sdfName, ExcelMissing.Value, 2);
15         XlCall.Excel(XlCall.xlfSetName, sdfName);
16         XlCall.Excel(XlCall.xlfUnregister, reregId);
17     });
18 }
```

Listing 6.13: Un-registering an SDF as a callable UDF in Microsoft Excel.

6.6.2.7 Using built-in Excel functions in SDFs

In Funsheet it is possible to use selected Excel built-in functions in SDF definitions. The list of supported Excel functions is configured through the Funsheet configuration file. An example of such a configuration file is shown in listing 6.14.

```
1  <configuration>
2    <appSettings>
```

```
    <add key="SupportedExcelFunctions" value="ARABIC, SUM, QUARTILE.INC,
    DEC2BIN, BIN2DEC" />
  </appSettings>
</configuration>
```

LISTING 6.14: An example of a configuration file for Funsheet, with several supported Excel functions listed.

Each supported Excel function from the configuration file is registered in the `ExcelFunCallContext` class. When the visitors, mentioned in the previous section, are visiting cells and expressions they check whether a formula contains a supported Excel function. Listing 6.15 shows how the supported Excel functions are identified in the visitors.

```
internal void Visit(Formula formula, ExcelReference reference)
{
    if (formula.Expr is FunCall)
    {
        var funCall = formula.Expr as FunCall;
        if
    (ExcelFunCallContext.SupportedFunctions.Contains(funCall.function.name) &&
    !ExcelFunCallContext.RegisteredFunctions.Contains(funCall.function.name))
        {
            Function.Remove(funCall.function.name);
            ExcelFunCallContext.Register(funCall.function.name);

            //Make a new Formula the first time
            formula = Formula.Make(_workbook,
    FunCall.Make(funCall.function.name, funCall.es));
        }
    }
    ...
```

LISTING 6.15: Registration of a supported Excel built-in function.

When a supported Excel function is encountered in a visitor for the first time, the `Register` method of the `ExcelFunCallContext` is called and the Excel function name is passed to it. The `ExcelFunCallContext` class has a static constructor that creates a transient, dynamic assembly, as shown in listing 6.16.

```
static ExcelFunCallContext()
{
    //Register supported Excel functions
    RegisterSupportedFunctions();

    var domain = AppDomain.CurrentDomain;
    var assemblyName = new AssemblyName("ExcelFunctionCalls");
    var assemBuilder = domain.DefineDynamicAssembly(assemblyName,
    AssemblyBuilderAccess.Run);
    _moduleBuilder = assemBuilder.DefineDynamicModule("ExcelFunCalls");
}
```

LISTING 6.16: Static constructor of the `ExcelFunCallContext` class

When an Excel built-in function is registered, we create a delegate capable of calling back into Excel using the Excel C API and wrap the result in a Funcalc `Value` type. The delegate is registered as a Funcalc `Function` to allow SDFs to invoke the delegate.

In our first attempt to create a delegate capable of calling into Excel we encountered a limitation in the .NET framework, in which we could not use a string closure in a delegate inside a delegate. It seems numerous people have encountered this inconsistency or bug in the .NET framework, but no fix has been provided by Microsoft. Because of this we needed another approach. We implemented code generation using LINQ expression trees to create the necessary delegates. Although this solution is working it is not very readable and hard to debug. Listing 6.17 shows our solution to the delegate problem. It shows how we are generating code by using the LINQ Expression API. Line 9-15 show how we compile the LINQ expression tree to a method which we save in the dynamic assembly for later use.

```
1  public static Func<Value[], Value> GenerateExcelCall(string function, out
       MethodInfo methodInfo)
2  {
3      var parameter = Expression.Parameter(typeof(Value[]), "values");
4      var functionName = Expression.Constant(function, typeof(string));
5      var excelFunCallInstance =
       Expression.New(typeof(ExcelFunCall).GetConstructor(new[] { typeof(string)
       }), functionName);
6      var excelFunCall = Expression.Call(excelFunCallInstance,
       ExcelFunCall.ExcelCallMethodInfo, parameter);
7      var expression = Expression.Lambda<Func<Value[], Value>>(excelFunCall,
       parameter);
8
9      var typeBuilder = _moduleBuilder.DefineType(function + "Delegate",
       TypeAttributes.Public);
10     MethodBuilder methodBuilder = typeBuilder.DefineMethod(function,
       MethodAttributes.Public | MethodAttributes.Static,
       CallingConventions.Standard);
11     methodBuilder.SetReturnType(typeof(Value));
12     methodBuilder.SetParameters(typeof(Value[]));
13     expression.CompileToMethod(methodBuilder);
14     var dynamicType = typeBuilder.CreateType();
15     methodInfo = dynamicType.GetMethod(function);
16
17     return (Func<Value[],
       Value>)dynamicType.GetMethod(function).CreateDelegate(typeof(Func<Value[],
       Value>));
18 }
```

Listing 6.17: The process of generating a delegate the LINQ Expression API.

In line 5 of listing 6.17 we make expressions to construct a new `ExcelFunCall` instance in which the Excel function name is injected (this was our initial problem when using delegates). Then in line 6 we make an expression to call the `ExcelCall` method of the `ExcelFunCall` instance. We have encapsulated as much logic as possible in the `ExcelCall` method to ease the debugging process and make the code more maintainable

and readable. The `ExcelCall` method is shown in listing 6.18. The method is building a string from the Excel function name and the Funcalc `Value` array elements supplied to the method. In line 27, this string is fed to the `xlfEvaluate` function from the Excel C API. The `xlfEvaluate` evaluates a string as if it was a formula written in Excel. The result from calling `xlfEvaluate` is converted to a Funcalc `Value` and returned.

```
1  public Value ExcelCall(Value[] values)
2  {
3      var sb = new StringBuilder();
4      foreach (var value in values.Select(s => s.ToExcelObject()))
5      {
6          if (value is string)
7          {
8              sb.AppendFormat("\"{0}\";", value.ToString());
9          }
10         else if(value is double[,])
11         {
12             sb.Append("{");
13             foreach (var val in (double[,])value)
14             {
15                 sb.Append(val.ToString() + ";");
16             }
17             // removes the last ;
18             sb.Remove(sb.Length - 1, 1);
19             sb.Append("};");
20         }
21         else
22         {
23             sb.Append(value.ToString() + ";");
24         }
25     }
26     var excelFormula = string.Format("{0}({1})", FunctionName, values.Length !=
       0 ? sb.Remove(sb.Length - 1, 1).ToString() : string.Empty);
27     var result = XlCall.Excel(XlCall.xlfEvaluate, excelFormula);
28     return HelperMethods.ObjectToValue(result);
29 }
```

Listing 6.18: The ExcelCall method is using the Excel C API to call into Excel.

6.6.2.8 Re-opening a saved workbook

Funsheet needs to be able to handle the scenario where a workbook containing DEFINE-formulas, is saved and later re-opened. This requirement raised several problems in the implementation of Funsheet. We needed a way to execute some initialization code in between the opening of a workbook and the execution of the first DEFINE-formula, to pre-register the defined SDFs in the workbook. This requirement arose for two reasons. Firstly SDFs can be defined to be recursive or mutually dependent. The second reason is the fact that we cannot trust Excel's order of calculation[13].

First we turned to the Excel C API documentation[14] hoping to discover an event that is fired during the initialization of an Microsoft Excel sheet. It turned out to be a dead

end. Next we started to investigate the COM Interop API, which is a much richer API than the Excel C API, but does not perform as well. The performance is not a concern to us in this particular situation as the initialization code will only run once at the opening of a workbook.

The COM Interop exposes many events, including a `WorkbookOpen`-event which is triggered every time a workbook is opened, either from an existing Excel instance or by double-clicking a saved workbook file. After some experimentation with the `WorkbookOpen`-event it turned out that when opening a workbook, Excel first executes all the DEFINE-formulas and afterwards triggers the `WorkbookOpen`-event. This led us to the final implementation where we use thread synchronization to direct the flow of execution. We added a private bool `IsInitialized` and a private object `_lock` (acting as a lock) to the `DefineFunction`-class. In the case where a workbook is opened the `IsInitialized` has the value `false`. Each DEFINE-formula will execute sequential because they all start out by trying to acquire the lock. Because the `IsInitialized` is false, the Define UDF saves the information necessary to create the SDF at a later point in time. Further a placeholder for the SDF is registered as shown in listing 6.19.

```
1 lock(_lock)
2 {
3     if (!IsInitialized)
4     {
5         DefineQueue.Add(sdfName, new Tuple<ExcelReference, ExcelReference,
      List<ExcelReference>, string>(callingCell, outputRef, inputs, fullFormula));
6         var sdfCellText = CorecalcRegistrator.RegisterPlaceholder(sdfName,
      callingCell);
7         RegisteredSdfContext.AddToRegisteredSdfs(sdfName,
      callingCell.ColumnFirst, callingCell.RowFirst, fullFormula, sdfCellText);
8         ...
```

Listing 6.19: When a workbook is opened information about SDFs are saved and a placeholder is created for each.

When all DEFINE-formulas have been executed, the `WorkbookOpen`-event is triggered. From the handler of this event we execute an asynchronous macro that fully registers all the SDFs based on the information saved in the previous steps as shown in listing 6.20. It is necessary to use a macro because we need access to the sheets of the workbook while creating the SDFs. Finally `IsInitialized` is set to true, to return to normal operation.

```
1 void app_WorkbookOpen(Workbook Wb)
2 {
3     ExcelAsyncUtil.QueueAsMacro(delegate
4     {
5         DefineFunction.RegisterSdfsInQueue();
```

```
6       });
7   }
```

Listing 6.20: Register all SDFs using a macro.

One problem introduced by this solution was that if a new workbook is created the `WorkbookOpen`-event is never triggered, which resulted in `IsInitialized` never would be set to true. To overcome this we experimented with the other events of the COM Interop API, hoping to find an event that is called both when a new workbook is created and when a saved workbook is opened. Further it should be called consistently after the `WorkbookOpen`-event. After a lot of trial and error we found that the `WindowActivate`-event is always called and always after the `WorkbookOpen`-event. In the event handler of this event we implemented a small piece of code to set the `IsInitialized` to true.

6.6.3 Closure

The Closure UDF accepts a string value containing the name of the SDF, closure or specialization on which to base the new closure upon. It further accepts a collection of arguments to the new closure. Arguments can be open or fixed. Open arguments can be supplied in Excel using the `NA()`-function. Fixed arguments is supplied like regular arguments.

The Closure UDF passes all the arguments to the `RegisterClosure` method of the `ClosureContext` class. The `Closurecontext` class will determine whether an identical closure already have been created. It will determine whether it should reference an SDF or an existing closure or specialization. Listing 6.21 shows part of the `RegisterClosure` method. We are wrapping the closure name and all inputs to the closure inside an appropriate sub-class of the Funcalc `Expression` class instance, because we need to delegate all these to the `Closure` function in Funcalc afterwards. The Funcalc `Closure` function returns a `FunctionValue` which we save in the `ClosureContext`. The `FunctionValue` can be used later by the other UDFs.

Finally we return an RTD wrapper to monitor the cell containing the CLOSURE-formula.

```
1   internal static object RegisterClosure(string function, object[] inputs, bool
        returnObserver)
2   {
3       var closure = new Closure
```

```
4       {
5           ReferenceTo = function,
6           Inputs = inputs,
7           IsCompiled = false
8       };
9
10      var expressions = new List<Expr>();
11      if (Contains(function))
12      {
13          closure.Name = Closures[function].Fv.ToString();
14          expressions.Add(ValueConst.Make(Closures[function].Fv));
15      }
16      else
17      {
18          if (SdfInfos.ContainsKey(function.ToUpper()))
19          {
20              var sdfInfo = SdfInfos[function.ToUpper()];
21
        expressions.Add(TextConst.Make(HelperMethods.ObjectToValue(sdfInfo.name)));
22          }
23      }
24
25      for (int i = 0; i < inputs.Length; i++)
26      {
27          var input = inputs[i];
28          if (input is ExcelReference)
29          {
30              input = (input as ExcelReference).GetValue();
31          }
32
33          if (input is string)
34          {
35              expressions.Add(TextConst.Make(HelperMethods.ObjectToValue(input)));
36          }
37          else if (input is double)
38          {
39
        expressions.Add(NumberConst.Make(HelperMethods.ObjectToValue(input)));
40          }
41          else if (input is object[,])
42          {
43              expressions.Add(ValueConst.Make(HelperMethods.ObjectToValue(input)));
44          }
45          else if (input is ExcelError && (ExcelError)input ==
        ExcelError.ExcelErrorNA)
46          {
47              expressions.Add(Error.Make(ErrorValue.naError));
48          }
49          else
50          {
51              return ExcelError.ExcelErrorValue;
52          }
53      }
54
55      var value = _closureFunc(null, expressions.ToArray(), 0, 0);
56      ...
```

Listing 6.21: Part of the `RegisterClosure` method

It is possible to create the same closure in multiple cells. Note that the dispose method of the RTD wrapper will only be invoked when the last cell containing the closure is cleared.

6.6.4 Specialize

The `Specialize` UDF is used to compile closures to make them perform better. UDF accepts a single string value. This should always reference an existing closure. We have decided to perceive the specializations as a kind of closure, therefore they are also managed through the `ClosureContext` class.

When using the UDF it calls the `RegisterCompiledClosure` method of the `ClosureContext`. Here we first check if all the arguments of the referenced closure are open. If this is the case the specialization would not yield any performance benefits because there is nothing to compile up front. Therefore we return the referenced closure instead, as shown in listing 6.22.

```
1  ...
2  if (closure.Fv.args.All(v => v == ErrorValue.naError))
3  {
4      //return observer to closure.
5      return ExcelCellHandle.CreateHandle("CLOSURE", closure.Name, () =>
       closure.Name, () => UnregisterClosure(closure.Name));
6  }
7  ...
```

Listing 6.22: Return the referenced closure if all arguments are open.

If the closure should be compiled we use the `SpecializeAndCompile` method of the `SdfManager` in Funcalc. To this method we supply the `FunctionValue` of the referenced closure. The `SpecializeAndCompile` method registers a new SDF and returns an `SdfInfo`. From the `SdfInfo` we create a `FunctionValue`. The newly created `FunctionValue` is then saved to be used by the other UDFs (See listing 6.23). Finally the `Specialize` UDF returns an RTD wrapper to the associated cell.

```
1  ...
2  var compiledClosure = new Closure();
3  compiledClosure.ReferenceTo = closureName;
4  compiledClosure.Inputs = closure.Inputs;
5  compiledClosure.IsCompiled = true;
6
7  compiledClosure.Fv = new
       FunctionValue(SdfManager.SpecializeAndCompile(closure.Fv), null);
8  compiledClosure.Name = compiledClosure.Fv.ToString();
9  Closures[compiledClosure.Name] = compiledClosure;
10
11 return ExcelCellHandle.CreateHandle("SPECIALIZE", compiledClosure.Name, () =>
       compiledClosure.Name, () => UnregisterClosure(compiledClosure.Name));
12 ...
```

Listing 6.23: Creating a specialization from an existing closure.

When the last cell containing a SPECIALIZE-formula to a specific closure is cleared, the specialization is deleted from Funcalc and its `FunctionValue` is removed from the `ClosureContext`.

6.6.5 Apply

The `APPLY` UDF is very simple in its implementation. It is marked as thread-safe because it is a *pure* function, which means it does not change the state of Funsheet. Because it is *pure* it is safe to allow parallel executions of this UDF. `APPLY` could potentially call a closure depending on an external function introducing state changes in Funsheet. But this is a hypothetical scenario and not very likely.

The UDF accepts a reference to a closure or specializtion along with a list of arguments. Listing 6.24 shows the core implementation of the `APPLY` UDF. First a closure is obtained from the `ClosureContext` class and the closure's `FunctionValue` is extracted. Then all the input arguments are converted from the Excel types to an array of the Funcalc `Value` type. Next we invoke the `APPLY` method of the `FunctionValue`. Finally we convert the resulting Funcalc `Value` to and Excel value and return this.

```
1 var fv = ClosureContext.GetClosure(closureName).Fv;
2 var vs = HelperMethods.ToValueArray(inputs);
3
4 if (fv == null)
5 {
6     return ErrorValue.argTypeError.ToExcelObject();
7 }
8
9 return fv.Apply(vs).ToExcelObject();
```

Listing 6.24: Implementation of the `Apply` UDF.

6.6.6 Benchmark UDF

The benchmark UDF was the simplest of all to implement. It has been marked as not thread-safe to ensure that only one `BENCHMARK`-formula is executing at a time. This was discussed in section 5.3.5.5 in the Design chapter.

Listing 6.25 shows the complete implementation. We have implemented a couple of checks that the closure or specialization exists. Then we simply delegate to the Funcalc implementation of `Benchmark` found in the `Function` class. The `FunctionValue` from

the closure or specialization is provided to the method and the double is wrapped in a `NumberValue` instance. Finally the return value from the `Benchmark` method is converted to a double using the extension method `ToExcelObject`.

```
1  [ExcelFunction(Name = "BENCHMARK", IsMacroType = false, IsThreadSafe = false)]
2  public static object Benchmark(string closureName, double iterations)
3  {
4      var closure = ClosureContext.GetClosure(closureName);
5      if(closure == null || closure.Fv == null)
6      {
7          return ErrorValue.argTypeError.ToExcelObject();
8      }
9      var result =  Function.Benchmark(closure.Fv, NumberValue.Make(iterations));
10     return result.ToExcelObject();
11 }
```

LISTING 6.25: Implementation of the `Benchmark` UDF

6.7 UI

In this section we cover the implementation of the UI enhancements to Microsoft Excel, introduced by Funsheet. This covers the implementation of design goal QG4 and QG6. Addressing design goal *QG6: The solution should conform to Microsoft Excel's standards and conventions*, we decided to implement the UI changes using the ribbon menu support in Excel-DNA. The first step was to make the `ExcelCalcRibbon` class (illustrated in class diagram in figure 6.1) and inherit from `ExcelRibbon`[15]. The class overrides the method `GetCustomUI`, which is inherited from the `ExcelRibbon`-class. This method lets us specify an XML file containing the ribbon specifications, such as text, images and actions triggered by the Funsheet ribbon. Ribbon menu customization using XML is supported by Microsoft[16]. In our case, Excel-DNA is acting as a wrapper for the existing extensibility. The actions specified in the XML file refer to methods inside the `ExcelCalcRibbon`-class. For example, the XML specification of the *Registered Closures* button illustrated in the Funsheet ribbon menu depicted in figure 6.5, is illustrated in listing 6.26. The action `onAction=\ShowClosures_onAction"` points to the `ShowClosures_onAction` method of the `ExcelcalcRibbon`-class. The methods creates a form containing a list of closures currently available in the current Excel workbook. To achieve better portability, we have included the XML file as an embedded resource in Funsheet as illustrated in figure 6.4 as opposed to keeping the file separate to the project.

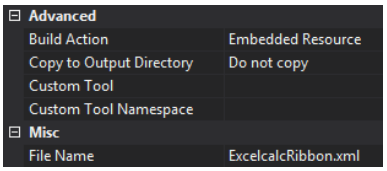

FIGURE 6.4: The XML file specifying the Funsheet ribbon is configured as an embedded resource in the Funsheet assembly.

The custom error handling that allow users to toggle between built-in Microsoft Excel errors, Funcalc errors and .NET exception (discussed in section 6.4), is controlled by the Funsheet ribbon. The decision on whether to show Funcalc errors is controlled by a public boolean value in the `ExcelCalcRibbon`-class, which is used by the value converter extension method to determine what error handling mechanism to use. This is illustrated in listing 6.2. When Funcalc errors are toggled on as illustrated in figure 6.6, the user will receive Funcalc errors as strings.

```
<box boxStyle="vertical" id="ExcelFunctions">
<button
description="Registered Closures"
enabled="true"
id="ShowClosures"
imageMso="ViewVisualBasicCode"
label="Registered Closures"
onAction="ShowClosures_onAction"
size="normal" />
```

LISTING 6.26: XML specifying text content, image and action for the *Registered Closures* button of the Funsheet ribbon.

FIGURE 6.5: The ribbon menu of Microsoft Excel 2013 customized with the Funsheet ribbon.

6.8 Evaluation

This chapter described the implementation of Funsheet. The implementation was based on the design and design goals, described in chapter 5. All design goals except *TG2:*

FIGURE 6.6: Funcalc errors has been toggled on in the Funsheet ribbon This enables the user to override the error mapping between Funcalc errors and Microsoft Excel errors introduced by Funsheet.

Support for editing previously defined SDFs, have been implemented. A possible implementation of TG2 is elaborated on in chapter 8, Future Work.

To achieve a maintainable implementation of Funsheet, we have used language features of C# and the .NET framework. For example, we used the `dynamic` keyword for runtime binding and used the The LINQ Expression API to avoid a bug in the .NET framework, which we discovered when we attempted to create delegates as described in section 6.6.2.7.

Section 6.6 explain how we implemented each of the functions of Funsheet. By implementing these, we have realized the technical design goals. Chapter 6: Test, confirms that all technical goals have been successfully implemented.

Lastly, we found the choice to use Excel-DNA (5.2) even better than we first realized. It helped us tremendously in the implementation of the Funsheet ribbon user interface. It also made it quite easy to implement the RTD wrapper by letting the wrapper inherit from the interface `IExcelObservable`.

Chapter 7

Test

This chapter describes how we defined and executed a series of tests. The purpose of the tests was to verify the implementation of the design goals we set out to realize have been achieved. We will cover each technical goal, except goal TG2, in test 1 through 7. TG2 was never implemented and is elaborated on in section 8.1 in chapter 8 concerning Future Work. In Test 8 we implemented three inter-connected SDFs to create and execute a Goal Seek function. This acts as a final test to verify the technical design goals. Quality goal QG1, regarding performance, is verified in Test 9. We have refrained from testing usability goals, as we believe a user study will be the optimal approach for testing this. We mention this in section 8.2 of chapter 8, Future Work. We indirectly tested parts of design goal QG4, concerning the display of an SDF's underlying bytecode to the user, in Test 1, 2, 4 and 5. Here we compared bytecode generated in Funsheet and Funcalc. All tests in this chapter were implemented in a Microsoft Excel sheet. A part of the tests was also implemented in Funcalc, for the purpose of comparing the generated byte code from each solution. All test cases was executed on a Dell XPS 13 laptop computer, with an Intel i7-3557U CPU @ 2.00GHz and 8GB of RAM running Windows 8 64 bit.

7.1 Organization of Tests

Each of the test cases we have defined are structured as follows:

- **Purpose**. A description of the test along with the purpose of executing the it. Will also contain references to the design goals being tested. Each design goal is covered by at least one test.
- **Implementation**. A detailed description of the steps necessary to implement and run the test.
- **Acceptance Criteria**. A list of criteria that should be met to pass the test execution.
- **Results**. The results of the test execution.
- **Evaluation**. Evaluation of the test execution and whether the specified acceptance criteria were met.

7.2 Tests

7.2.1 Test 1: Define SDF

7.2.1.1 Purpose

This test is concerned with Design Goal *TG1: Support for creating and calling SDFs in Funcalc from Excel.*

The purpose of this test is to verify the ability of Funsheet to correctly create and register SDFs in Microsoft Excel and that the generated bytecode in Funsheet is equal to the bytecode generated in Funcalc based on the same functions.

7.2.1.2 Implementation

We start out with defining three SDFs in Microsoft Excel as illustrated in figure 7.1. The defined SDFs are, `INCREMENT`, which increments a single input number parameter by one; `MYCONCAT`, which concatinates "Hello" with a single string input parameter; `MYMIN`, which returns the smallest number of the input range of numbers.

We define equivalent SDFs in Funcalc to compare the generated bytecode with the bytecode generated in Funsheet.

	A	B	C
1	**SDF #1: Number addition**	**SDF #2: String concatination**	**SDF #3: Range manipulation**
2	1	Mister	=MIN(Sheet2!A1:D1)
3	=A2+1	="Hello "&B2	=DEFINE("MYMIN";C2;A2)
4	=DEFINE("INCREMENT";A3;A2)	=DEFINE("MYCONCAT";B3;B2)	

FIGURE 7.1: All three SDFs for Test 1. Note that when defining SDFs with ranges, Funcalc does not allow to reference ranges from the same functions sheet as the SDF definition.

7.2.1.3 Acceptance Criteria

All the SDFs mentioned in the Implementation section must be defined and called without errors. They should yield the correct values based on the test vectors from table 7.1 in the results section.

Lastly, the Funsheet and Funcalc implementations bytecode should not differ. Note: the display of generated byte code is specified in design goal QG4, which is discussed in chapter 5.

7.2.1.4 Results

In table 7.1 the input values and their corresponding output values are illustrated. The generated bytecode from Funsheet and Funcalc is a match on all three SDFs. The bytecode for each of the three SDFs are illustrated in figure 7.2, figure 7.3 and in figure 7.4.

SDF	**Input**	**Output**
`INCREMENT`	10	11
`MYCONCAT`	"World"	"Hello World"
`MYMIN`	3873, 55, 250, 4, 95293	4

TABLE 7.1: Table showing the input and output from test 1.

```
IL_0000: ldarg      V_0
IL_0004: call       Double ToDoubleOrNan(Corecalc.Value)/Corecalc.Value
IL_0009: ldc.r8     1
IL_0012: add
IL_0013: call       Corecalc.Value Make(Double)/Corecalc.NumberValue
IL_0018: ret
```

FIGURE 7.2: The generated bytecode code from the `INCREMENT` function is the same in both Funsheet and Funcalc.

```
IL_0000: ldc.i4     2
IL_0005: call       Corecalc.TextValue FromIndex(Int32)/Corecalc.TextValue
IL_000a: ldarg      V_0
IL_000e: call       Corecalc.Value ExcelConcat(Corecalc.Value, Corecalc.Value)/
                    Corecalc.Function
IL_0013: ret
```

FIGURE 7.3: The generated bytecode code from the `MYCONCAT` function is the same in both Funsheet and Funcalc.

```
IL_0000: ldc.i4     1
IL_0005: newarr     Corecalc.Value
IL_000a: dup
IL_000b: ldc.i4     0
IL_0010: ldc.i4     0
IL_0015: call       Corecalc.ArrayView GetArrayView(Int32)/Corecalc.Funcalc.
                    CGNormalCellArea
IL_001a: stelem.ref
IL_001b: call       Double Min(Corecalc.Value[])/Corecalc.Function
IL_0020: call       Corecalc.Value Make(Double)/Corecalc.NumberValue
IL_0025: ret
```

FIGURE 7.4: The generated bytecode code from the `MYMIN` function is the same in both Funsheet and Funcalc.

7.2.1.5 Evaluation of Test 1

The results show that test 1 passed. The SDFs were successfully created and produced the expected results. The generated bytecode code from Funsheet proved to be equal to the bytecode generated in Funcalc. The test is passed.

7.2.2 Test 2: Define SDFs with a variable amounts of input argument

7.2.2.1 Purpose

This test addresses design goal *TG1: Support for creating and calling SDFs in Funcalc from Excel.* The purpose of this test is to ensure the support for creation of SDFs with an input argument count ranging from zero to nine. To maximize the testing effort, we have decided to carry out an equivalence partitioning and a boundary-value analysis to identify which partitions we should focus this test on. The equivalence partitioning and the boundary-value analysis are specified in subsection 7.2.2.2.

7.2.2.2 Equivalence Partitioning and Boundary-value analysis

We have identified three partitions that we assume the system handles: One partition that is impossible, which is less than zero arguments, one partition that covers the count from zero to nine input arguments and lastly one partition for ten or more arguments. We expect the last to return an error. These partitions are illustrated in figure 7.5. The boundary values in this case are 0, 1, 8, 9, 10 arguments. Based on that, we have decided to implement six test cases; one for each boundary value and we have also added an extra for the case of five arguments. Five is in the middle of the spectrum and we believe its is worth testing to give us a higher test coverage.

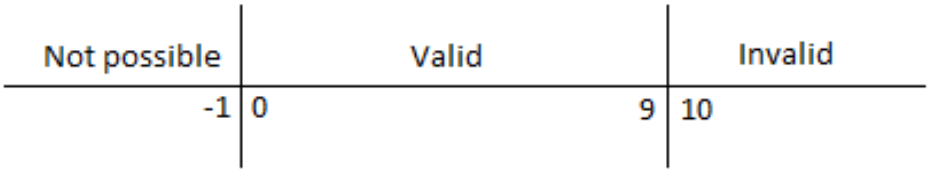

FIGURE 7.5: Three partitions on the test vectors of zero to nine arguments: One invalid partition, since it is not possible to input less than zero arguments, one that covers zero to nine arguments and is valid and one invalid partition which accounts for cases with ten or more arguments.

7.2.2.3 Implementation

The test implements six test cases, based on our boundary-value analysis illustrated in figure 7.5:
The SDFs in this test case will be simple. They will all do a simple addition operation that returns the sum of the input argument(s). The setup of the six SDFs is illustrated in figure 7.6.
We will implement the same SDFs in Funcalc to allow the comparison of bytecode.

7.2.2.4 Acceptance Criteria

All the SDFs mentioned in the Implementation section must be defined and called without error, except the last SDF with ten arguments. We expect that to return an error.
The SDFs should yield the correct values based on the test vectors from the table in the results section.

	A	B	C	D
1	1	=A1	=DEFINE("Zero";B1)	=Zero()
2	1	=A1	=DEFINE("One";B2;A1)	=One(1)
3	1			
4	1			
5	1			
6	1	=A1+A2+A3+A4+A5	=DEFINE("Five";B6;A1;A2;A3;A4;A5)	=Five(1;2;3;4;5)
7	1			
8	1			
9	1	=A1+A2+A3+A4+A5+A6+A7+A8	=DEFINE("Eight";B9;A1;A2;A3;A4;A5;A6;A7;A8)	=Eight(1;2;3;4;5;6;7;8)
10	1	=A1+A2+A3+A4+A5+A6+A7+A8+A9	=DEFINE("Nine";B10;A1;A2;A3;A4;A5;A6;A7;A8;A9)	=Nine(1;2;3;4;5;6;7;8;9)
11	1	=A1+A2+A3+A4+A5+A6+A7+A8+A9+A10+A11	=DEFINE("Ten";B11;A1;A2;A3;A4;A5;A6;A7;A8;A9;A10)	

FIGURE 7.6: Illustration of the SDF setup in Test 2. All the function calls in the D column are valid and were executed without errors. Notice that the SDF "Ten" is not called. This is because it never gets created because of the number of argument specified in its definition.

The Funsheet and Funcalc implementations byte code should not differ.

7.2.2.5 Results

In this section we present the result for test 2. In table 7.2 the input values and their corresponding output values are illustrated. The bytecode generated in Funsheet and Funcalc was equal.

Argument Count	**Expected Result**	**Result**
0	SDF Created	SDF Created
1	SDF Created	SDF Created
5	SDF Created	SDF Created
8	SDF Created	SDF Created
9	SDF Created	SDF Created
10	Excel error shown	Excel error shown

TABLE 7.2: Table showing the number of arguments an SDF is attempted defined with and whether it is accepted by the system. Test cases where the argument count is less than ten result in successfully created SDFs.

7.2.2.6 Evaluation of Test 2

The input partitions were processed as expected. SDF definitions with an argument count of between zero and nine were created successfully. An argument count of ten was rejected by Excel, as expected. The generated bytecode matched. The test passed.

7.2.3 Test 3: Delete SDF when cell containing SDF definition is cleared.

7.2.3.1 Purpose

This test is concerning the design goal *TG5: Support deletion of SDFs.*
The purpose of this test is to ensure that an SDF will be unregistered from Excel if the cell containing its definition is cleared.

7.2.3.2 Implementation

This test implements a single test case. One SDF Increment, which takes one number argument and adds 1 to it. The test setup is illustrated in figure 7.7.

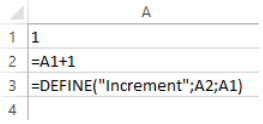

FIGURE 7.7: Setup of the `INCREMENT` SDF in Test 3.

7.2.3.3 Acceptance Criteria

The SDF mentioned in the Implementation section must be defined and called without error. Then we empty the cell containing the SDF definition. Hereafter the SDF should not be callable from Excel.

7.2.3.4 Results

After the SDF had been defined it was callable and worked as expected. After the SDF definition cell had been cleared, Microsoft Excel would would show a `#NAME?` error when the SDF was called, indicating that the function was unknown. This is illustrated in 7.10. The function auto suggestion in Excel was also missing the SDF name after the deletion. Microsoft Excel before deletion is illustrated in figure 7.8 and after deletion is illustrated in figure 7.9.

7.2.3.5 Evaluation of Test 3

The test executed as expected. The SDF was successfully deleted when the cell containing its definition was cleared. The test passed.

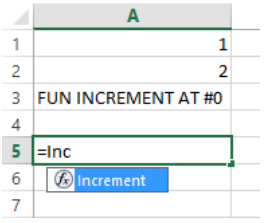

FIGURE 7.8: Before deletion. Microsoft Excel's auto suggestion for functions recognizes the recently defined SDF.

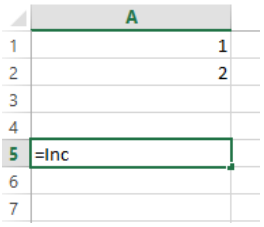

FIGURE 7.9: After deletion. Microsoft Excel's auto suggestion for functions no longer recognizes the deleted SDF.

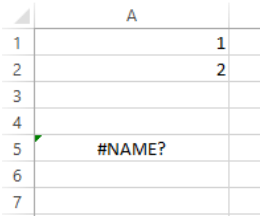

FIGURE 7.10: After deletion. Microsoft Excel gives the `#NAME?` error when the recently deleted SDF is attempted to be called. This error indicates that the function is not recognized by Microsoft Excel.

7.2.4 Test 4: Creating Closures

7.2.4.1 Purpose

This test addresses design goal *TG3: Support for defining and calling closures.*

The purpose of this test is to verify that closures can be created and registered in Microsoft Excel and afterwards called using the Apply UDF introduced by Funsheet.

7.2.4.2 Implementation

We will implement two closures based on the pre-defined SDF `INCREMENT`. This SDF accepts one number as an argument and returns the number incremented by one. One Closure will be created with `#N/A!` as a pre-defined argument, meaning that the Closure accepts one argument. The Excel implementation is shown in figure 7.11. The other

Closure will be created with *1* as a pre-defined argument, resulting in a Closure that takes zero arguments. The Excel implementation is shown in figure 7.12.
We will implement both Closures in both Funsheet and Funcalc to compare the generated bytecode.

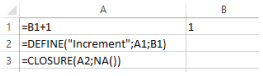

FIGURE 7.11: Closure defined with `#N/A!` as a pre-defined argument. Result is a Closure that accepts one argument.

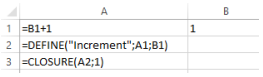

FIGURE 7.12: Closure defined with 1 as a pre-defined argument. Result is a closure that accepts zero arguments.

7.2.4.3 Acceptance Criteria

We assume the SDF from which the closure is created, has been created and registered successfully beforehand. Both Closures should be created and registered successfully. They should also produce the expected results when called.
No bytecode comparisons will be included in this test, as no bytecode is generated for Closures.

7.2.4.4 Results

Both Closures were successfully created. They both produced the expect results as illustrated in table 7.3. The generated bytecode in Funsheet and Funcalc matched.

Closure	Pre-defined Argument	Input	Output	Expected Output
#1	`#N/A!`	5	6	6
#2	1	None	2	2

TABLE 7.3: The input and output from test 4. Output and expected output are matching. Closure #1 has been defined with `#N/A!` as a pre-defined argument resulting in a Closure that accepts one argument. Closure #2 has been defined with 1 as a pre-defined argument resulting in a Closure that accepts zero arguments.

7.2.4.5 Evaluation of Test 4

Both Closures were created successfully and returned the expected results. The generated byecode also matched. The test passed.

7.2.5 Test 5: Specialize Closures.

7.2.5.1 Purpose

This test is regarding design goal *TG6: Support for specialization of closures.* In this test we will verify the specialization of closures. We will also verify that the produce the expected result, when executing them using the `APPLY` UDF.

7.2.5.2 Implementation

We will specialize the two closures defined in Test 4. One closure that takes a single number argument and another closure that takes no arguments. Both closures are based on the `INCREMENT` SDF, which accepts a number as an argument and returns the number increment by one.
We will implement two equal specializations in both Funcalc and Funsheet to compare the generated bytecode.

7.2.5.3 Acceptance Criteria

We assume that the closure and the SDF from which the closure is made, both have been created successfully beforehand. Both specializations should produce the expected results.

7.2.5.4 Results

Both specializations were successfully created. They both produces the expected results as illustrated in table 7.4.
The generated bytecode in Funsheet is equal to that of Funcalc. The bytecode for #1 specialization is shown in listing 7.13. The bytecode for #2 specialization is shown in listing 7.14.

```
IL_0000: ldarg      V_0
IL_0004: call       Double ToDoubleOrNan(Corecalc.Value)/Corecalc.Value
IL_0009: ldc.r8     1
IL_0012: add
IL_0013: call       Corecalc.Value Make(Double)/Corecalc.NumberValue
```

Specialization	Pre-defined Argument	Input	Output	Expected Output
#1	`#N/A!`	10	11	11
#2	5	None	6	6

TABLE 7.4: The input and output from test 5. Output and expected output are matching. Specialization #1 has been defined from a closure that accepts one argument. Specialization #2 has been defined from a Closure that accepts zero arguments.

```
IL_0018: ret
```

FIGURE 7.13: The generated bytecode code from the specialized closure which takes a single argument. The bytecode is identical in Funsheet and Funcalc.

```
IL_0000: ldc.r8     2
IL_0009: call       Corecalc.Value Make(Double)/Corecalc.NumberValue
IL_000e: ret
```

FIGURE 7.14: The generated bytecode code from the specialized closure which is predefined with a *1* as its only argument. The bytecode is equal in Funsheet and Funcalc.

7.2.5.5 Evaluation of Test 5

Both specializations were both created successfully and returned the expected results. The bytecode also matched. The test passed.

7.2.6 Test 6: Calling built-in Excel functions when function is not present in Funcalc.

7.2.6.1 Purpose

This test addresses design goal *TG4: Support for calling built-in Microsoft Excel functions from Funsheet SDFs* and design goal *QG5: Error handling should support the user.*

The goal of this test is to ensure that SDFs defined in Funsheet can use built-in Microsoft Excel functions and that the correct Excel error is shown, when a function is not available.

7.2.6.2 Implementation

This test will implement two SDFs that respectively will be using the `ARABIC` and `QUARTILE` functions from Microsoft Excel. None of these functions are, at the time of writing, supported by Funcalc. Hence Funsheet has to call the functions from Excel,

but only if the functions are specified on the *Supported Excel Functions*-list in Funsheet. This function list is explained in section 6.6.2.7. The `ARABIC` function converts a Roman number to an Arabic number. It accepts a string as input. The `QUARTILE` function returns a chosen quartile from a dataset. It accepts two inputs, the dataset and an integer from 0 to 4. 0 is the minimum value, 1 is the first median, 2 is the median value, 3 is the third quartile, 4 is the maximum value. We will start out by implementing an SDF called `MYARABIC`. It will take two string arguments and pass them to Excel's `ARABIC` function and add the results. The second SDF `MYMIN` will have its input parameter use the `QUARTILE` function with a hardcoded *0* as its second parameter. This way the minimum is always returned.

We will execute two test cases, one for each SDF. Before executing the test cases, we will ensure that only the `ARABIC` function exists on the *Supported Excel Functions*-list. This indicates, that the SDF relying on the `QUARTILE` function should show an Excel error when it is called from Excel. The implementation of the `MYARABIC` SDF is illustrated in figure 7.15. The implementation of the `MYMIN` SDF is illustrated in figure 7.16.

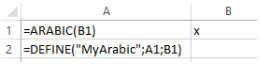

FIGURE 7.15: The implementation of the `MYARABIC` SDF which uses the built-in Excel function `ARABIC`.

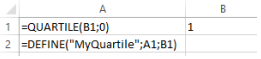

FIGURE 7.16: The implementation of the `MYMIN` SDF which uses the built-in Excel function `QUARTILE` to return the minimum value.

7.2.6.3 Acceptance Criteria

We expect that calling an SDF where the output cell either is directly or transitive dependant on a function that is not present in Funcalc nor is present on the *Support Excel Functions* list in Funsheet, will fail. If the function exists on the function list and is available from Excel, the SDF call should succeed.

Furthermore should the valid SDF (`MYARABIC`) return the expected output when executed.

7.2.6.4 Results

The `MYARABIC` SDF executed successfully and returned the correct result. The `MYMIN` resolved in a `#NAME?` error, which was expected.

SDF	Excel Function	Funsheet Support	Input	Expected Result	Result
`MYARABIC`	`ARABIC`	YES	“X”;“M”	1010	1010
`MYMIN`	`QUARTILE`	NO	4, 3, 200, 2, 77	`#NAME?`	`#NAME?`

TABLE 7.5: Inputs, outputs and expected outputs from test case 6. When an SDF relies on a function not available in Funcalc and not on the supported function list in Funsheet, Excel shows an error.

7.2.6.5 Evaluation of Test 6

As expected, Excel showed an error when the `MYMIN` SDF was called, due to the Excel function `QUARTILE` not being noted on the *Support Excel Functions* list in Funsheet. On the other hand, the `MYARABIC` SDF was executed successfully, returning the correct output. The test passed.

7.2.7 Test 7: Define Recursive SDF

7.2.7.1 Purpose

This test addresses design goal *TG7: Support for defning recursive SDFs.*
The purpose of this test is to recursively defining a recursive SDF.

7.2.7.2 Implementation

We wish to implement the `MYGCD` SDF for identifying the greatest common divisor (GCD) of two numbers. The implementation of the SDF relies on the `MOD` function which is supported by Funcalc. The SDF also relies on itself. This is illustrated in figure 7.17.

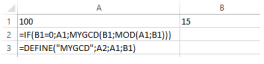

FIGURE 7.17: The implementation of the `GCD` SDF for finding the greatest common divisor between two numbers.

7.2.7.3 Acceptance Criteria

The SDF should be successfully created and registered. Furthermore it should produce the correct results.

7.2.7.4 Results

The initial creation of the SDF resulted in a `#NAME?` error in the output cell in Excel (Cell A2 in figure 7.17). This is because the SDF is non-existent on the time of creation. When the output cell is refreshed, after the SDF has been defined, the error disappears. This is working as intended. The SDF also returns the correct result, as illustrated in table 7.6.

SDF	First input	Second input	Expected Result	Result
`MYGCD`	12	90	6	6
`MYGCD`	-77	91	7	7

TABLE 7.6: Inputs, outputs and expected outputs from test case 7 using the global common divisor SDF `MYGCD`.

7.2.7.5 Evaluation of Test 7

The output cell of `MYGCD` relies on the SDF itself and was therefore resolved to an Excel error before the SDF had been defined. This was working as intended. The result section shows that the SDF returns the correct results. The test passed.

7.2.8 Test 8: Goal Seek

7.2.8.1 Purpose

This test will implement a realistic use case for an SDF, by defining a Goal Seek function. The implementation includes an auxiliary SDF and is much more complex that the SDFs defined in the previous tests. Conducting this might help uncover edge cases and scenarios untouched by the more simple tests. The Goal Seek function seeks to find a solution x to the equation $f(x) = r$ if one exists. The input to the Goal Seek function is a continuous function f (a closure), a target value r and an initial guess a at the value of x. The first few steps of the Microsoft Excel implementation are illustrated in figure 7.18.

Besides implementing the Goal Seek function, we wish to use it to determine x for the function `INCREMENT`, for a target value r of 100. The `INCREMENT` function accepts a number and increments it by one.

Lastly, we will also use this test for verifying design goal *TG8: It must be possible to save and reload SDFs*, which states that it should be possible to open saved workbooks containing mutual recursive SDFs.

	A	B	C	D	E	F
21	f=	=B5	taget=	100	a=	1
22	=DEFINE("GOALSEEK";D57;B21;D21;F21)					
23	=GOALSEEK(B21;100;96)	a0	b0	f(a0)-target	f(b0)-target	
24		=$F$21	=FINDEND($B$21;$D$21;B2	=APPLY($B$21;B24)-(	=APPLY(B21;C24)-D21	
25						
26		Invariant: f(ai)<=target<=f(bi)				
27		ai	bi	xi=(ai+bi)/2	f(x)-target	
28		=IF($D$24<=0;B24;C24)	=IF($D$24<=0;C24;B24)	=(B28+C28)/2	=APPLY($B$21;D28)-$D$21	
29		=IF($E28<=0;$D28;B28)	=IF($E28>=0;$D28;C28)	=(B29+C29)/2	=APPLY($B$21;D29)-$D$21	
30		=IF($E29<=0;$D29;B29)	=IF($E29>=0;$D29;C29)	=(B30+C30)/2	=APPLY($B$21;D30)-$D$21	
31		=IF(E30<=0;D30;B30)	=IF($E30>=0;$D30;C30)	=(B31+C31)/2	=APPLY($B$21;D31)-$D$21	
32		=IF($E31<=0;$D31;B31)	=IF($E31>=0;$D31;C31)	=(B32+C32)/2	=APPLY($B$21;D32)-$D$21	
33		=IF($E32<=0;$D32;B32)	=IF($E32>=0;$D32;C32)	=(B33+C33)/2	=APPLY($B$21;D33)-$D$21	
34		=IF($E33<=0;$D33;B33)	=IF($E33>=0;$D33;C33)	=(B34+C34)/2	=APPLY($B$21;D34)-$D$21	
35		=IF($E34<=0;$D34;B34)	=IF($E34>=0;$D34;C34)	=(B35+C35)/2	=APPLY($B$21;D35)-$D$21	
36		=IF($E35<=0;$D35;B35)	=IF($E35>=0;$D35;C35)	=(B36+C36)/2	=APPLY($B$21;D36)-$D$21	

FIGURE 7.18: First couple of iterations of the Goal Seek function.

7.2.8.2 Implementation

The Goal Seek function accepts the following arguments: a function value (a1), a target value (a2) and a start value (a3). The Goal Seek function's objective is to determine what input is needed for a1 to return the desired result a2. It relies the auxiliary function `FINDEND` to search for an x for which *f(x)* has the opposite sign of *f(a)*. The implementation of `Findend` is illustrated in figure 7.19.

We also implement the function `INCREMENT` as illustrated in column A in figure 7.1.

Our implementation uses a maximum of thirty iterations for finding the target value.

	A	B	C	D	E	F	G	H	I
1	GOALSEEK(f, r, a)		=D1+1	1					
2	=DEFINE("INC";C1;D1)								
3	FINDEND(f, target, a)								
4	=DEFINE("FINDEND";I5;B5;D5;F5)	=FINDEND(B5;100;1)							
5	f=	=CLOSURE(A2;NA())	target=	100	a=	1			=I8+0
6									
7	a	delta	a-delta	a+delta	f(a-delta)	f(a+delta)	OK(a-delta)	OK(a+delta)	
8	1	0,0001	=A8-B8	=A8+B8	=APPLY(B5;C8)-D5	=APPLY(B5;D8)-D5	=A9*E8<=0	=A9*F8<=0	=IF(G8;C8;IF(H8;D8;I9))
9	=APPLY(B5;F5)-D5	=B8*10	=A8-B9	=A8+B9	=APPLY(B5;C9)-D5	=APPLY(B5;D9)-D5	=A9*E9<=0	=A9*F9<=0	=IF(G9;C9;IF(H9;D9;I10))
10		=B9*10	=A8-B10	=A8+B10	=APPLY(B5;C10)-D5	=APPLY(B5;D10)-D5	=A9*E10<=0	=A9*F10<=0	=IF(G10;C10;IF(H10;D10;I11))
11		=B10*10	=A8-B11	=A8+B11	=APPLY(B5;C11)-D5	=APPLY(B5;D11)-D5	=A9*E11<=0	=A9*F11<=0	=IF(G11;C11;IF(H11;D11;I12))
12		=B11*10	=A8-B12	=A8+B12	=APPLY(B5;C12)-D5	=APPLY(B5;D12)-D5	=A9*E12<=0	=A9*F12<=0	=IF(G12;C12;IF(H12;D12;I13))
13		=B12*10	=A8-B13	=A8+B13	=APPLY(B5;C13)-D5	=APPLY(B5;D13)-D5	=A9*E13<=0	=A9*F13<=0	=IF(G13;C13;IF(H13;D13;I14))
14		=B13*10	=A8-B14	=A8+B14	=APPLY(B5;C14)-D5	=APPLY(B5;D14)-D5	=A9*E14<=0	=A9*F14<=0	=IF(G14;C14;IF(H14;D14;I15))
15		=B14*10	=A8-B15	=A8+B15	=APPLY(B5;C15)-D5	=APPLY(B5;D15)-D5	=A9*E15<=0	=A9*F15<=0	=IF(G15;C15;IF(H15;D15;I16))
16		=B15*10	=A8-B16	=A8+B16	=APPLY(B5;C16)-D5	=APPLY(B5;D16)-D5	=A9*E16<=0	=A9*F16<=0	=IF(G16;C16;IF(H16;D16;I17))
17		=B16*10	=A8-B17	=A8+B17	=APPLY(B5;C17)-D5	=APPLY(B5;D17)-D5	=A9*E17<=0	=A9*F17<=0	=IF(G17;C17;IF(H17;D17;NA()))

FIGURE 7.19: The Microsoft Excel implementation of the Goal Seek function's auxiliary funciton `FINDEND`.

7.2.8.3 Acceptance Criteria

The Goal Seek SDF must not take more than five seconds to execute. It should be verified against the Goal Seek function of Microsoft Excel, assuming the iteration count is set to thirty in Excel.

7.2.8.4 Results

The Goal Seek function was successfully implemented using SDFs. Using it to find the correct input value for the `INCREMENT` function yielded the same result, using both the Microsoft Excel implementation and the SDF implementation, which is illustrated in table 7.7.

To verify design goal TG8 regarding the opening of saved workbooks containing SDFs, we were able to successfully save, close and re-open the workbook containing the Goal Seek SDF implementation.

Goal Seek impl.	**function f**	**target value x**	**Initial guess a**	**Result**
Microsoft Excel	SDF `INCREMENT`	100	1	99
Funsheet	SDF `INCREMENT`	100	1	99

TABLE 7.7: Function f, target value x, initial guess a and result of using two different Goal Seek implementations running thirty iterations. Results were equal in both cases.

7.2.8.5 Evaluation of Test 8

When we began implementing the Goal Seek function using SDFs, we discovered that the definition of the `GOALSEEK` SDF had exponential running time. We were able to define the SDF in a couple of seconds, when limiting the iterations (illustrated in figure 7.18) to four. This gave us the notion that similar cells were covered multiple times in the cell visitors, used by the `DEFINE` function. To achieve linear running time with the cell visitors, we implemented memoization as described in section 6.6.2.2. Another challenge emerged, after we were able to define the `GOALSEEK` SDF in linear time. By saving the workbook containing the SDF definitions required for implementing the `GOALSEEK` function and opening it at later time, we discovered that Excel was not adhering to the implicit calculation order we had dictated when initially defining the SDFs one step at a time. this lead us to explicitly define *TG8: It must be possible to save and reload SDFs* as a design goal. The implementation of TG8 is discussed in section 6.6.2.8.

This test was by far the most complex and demanding in this chapter. It successfully verified that the implementation of our design goals can work in union and that the concept of sheet-defined functions has been integrating into Excel. The test passed.

7.2.9 Test 9: Performance

7.2.9.1 Purpose

This test case is addressing design goal *DG8: Execution of SDFs should take at most 20% longer than execution of similar SDFs in Funcalc.*

7.2.9.2 Implementation

We will implement an SDF `TRIAREA` that calculates the area of a triangle with side lengths *A1*, *B1* and *C1*. The Microsoft Excel SDF implementation is illustrated in figure 7.20. We then create a closure from the SDF with *A1*, *B1* and *C1* sat to 30, 40 and 50. We then pass the closure and the value *10,000,000* to the Funsheet UDF `BENCHMARK` (discussed in section 5.3.5.5). The `BENCHMARK` UDF will the return the average running time of 10,000,000 executions of the SDF in nanoseconds.

We will also use the `GOALSEEK` SDF implementation from section 7.2.8. As function value f we will provide a closure of the `INCREMENT` SDF with `#NA!` as a placeholder argument. Target value x will be 100 and the inital guess a will be 1. We will run the SDF 1,000,000 times using the `BENCHMARK` UDF.

We will implement the both SDFs in Funcalc and call them with `BENCHMARK` UDF. The `TRIAREA` SDF will be executed 10,000,000 and the `GOALSEEK` SDF will be executed 1,000,000.

	A	B
1	TRI AREA	
2	a	30
3	b	40
4	c	50
5	s	=(B2+B3+B4)/2
6	area	=SQRT(B5*(B5-B2)*(B5-B3)*(B5-B4))
7	=DEFINE("TriArea";B6;B2;B3;B4)	

FIGURE 7.20: The implementation of the `TRIAREA` SDF for calculating the area of a triangle with side lengths *A1*, *B1* and *C1*.

7.2.9.3 Acceptance Criteria

As stated in design goal *DG8: Execution of SDFs should take at most 20% longer than execution of similar SDFs in Funcalc*, the result of the `BENCHMARK` UDF in Funsheet should maximally be 20% higher than the equivalent benchmarking in Funcalc.

7.2.9.4 Results

The results shows that `TRIAREA` SDF defined in Funcalc is benchmarked to be 1.3 times slower than equivalent SDFs in Funsheet. The `GOALSEEK` SDF is only 1.09 times slower than its Funcalc equivalent. The results are illustrated in table 7.8.

Application	**SDF**	**Iterations**	**Benchmark result (ns)**
Funsheet	`TRIAREA`	10,000,000	48
Funcalc	`TRIAREA`	10,000,000	36
Funsheet	`GOALSEEK`	1,000,000	4545
Funcalc	`GOALSEEK`	1,000,000	4135

TABLE 7.8: The benchmark results in nanoseconds for the `TRIAREA` SDF and the `GOALSEEK` SDF.

7.2.9.5 Evaluation of Test 9

We stated in design goal QG1 that the execution of SDFs in Funsheet should take at most 20% longer than the execution of similar SDFs in Funcalc. From the results found in table 7.8, we show that the execution of the `TRIAREA` SDF in Funsheet takes 33% (1.3 times) longer than the equivalent SDF in Funcalc. However, the `GOALSEEK` SDF takes 9.9%(1.09 times) longer than the equivalent SDF in Funcalc. We therefore achieve our performance goal of not exceeding 20% execution time in regards to the `GOALSEEK` SDF, but fall short of achieving the goal in regards to the `TRIAREA` SDF.

The `BENCHMARK` UDF of Funsheet is implemented to solely rely on Funcalc to perform the benchmarking. The stopwatch that calculates the result in the `BENCHMARK` UDF, is initiated and stopped inside Funcalc. Funsheet has no control over this and is only responsible for passing the closure and iteration details to Funcalc. The `BENCHMARK` UDF is discussed in more detail in 5.3.5.5. We therefore speculate that Microsoft Excel has an impact on the increased execution time, since Excel is controlling the process in which the benchmarking is executed. In spite of the `TRIAREA` SDF exceeding the execution

limit of 20%, we evaluate this test as passed, due to the much more satisfactory result of the SDF `GOALSEEK`.

7.3 Evaluation

Concluding on our testing effort, we believe we have achieved a functional add-in that fulfills all our technical goals, except TG2, which is discussed in section 8.1 in Future Work. Test 9 concerning performance goal QG1, lead to conflicting results. One test case exceeded our SDF execution time limit of being, at most 20% slower, another test case was well within the time limits. In section 7.2.9.5, we discuss that the increased execution time might be due to Microsoft Excel controlling the process in which the benchmarking is executed. Hence we believe that further performance testing and research should be conducted to clarify the reasons behind these results. This is mentioned in section 8.5. We believe that Funsheet should be subject to more testing before being used in production. In section 8.2 and section 8.3 of chapter 8 Future Work, we discuss how a user study and a a real world case should be carried out to test Funsheet in a more realistic environment.

Chapter 8

Future Work

In this section we will discuss the possibilities for future work on Funsheet.

8.1 Editing of existing SDF definitions

In Funsheet the design goal *TG2 Support for editing previously defined SDFs* was not implemented. At the moment of writing, the only solution to changing the output or input of an existing SDF is to delete it and create a new one. This is not consistent with Microsoft Excel, where a formula automatically updates if its precedent cells are changed. A candidate solution to support editing of existing SDFs, could be to use hashing to identify changes in an SDF definition. The cell visitors (described in 6.6.2.2) would be responsible for building a hash from all the cells visited throughout the creation of an SDF. The hash would then be linked to the SDF name and every time the `DEFINE` function is called with an SDF definition containing a name of an existing SDF, the cell visitors would construct a hash and compare it to the existing hash. If the hashes are identical, nothing has been changed. If not, Funsheet should update the SDF.

8.2 Funsheet user study

In chapter 7 we validated the technical and performance design goals of Funsheet. We believe that conducting a user study with real world Excel users could verify that the usability goals of Funsheet has been realized. Besides, we could use it identify areas in need of improvement and uncover bugs. In section 8.3 we elaborate on a user study in a professional environment.

8.3 Funsheet production environment study

Funsheet and the concept of SDFs especially caters to professional Microsoft Excel users. Conducting a user study addressing this user segment would evaluate the Funsheet implementation in a real production environment, such as using Funsheet in banking. It could contribute to the future development of Funsheet and uncover use cases we have overlooked during the implementation and the testing of Funsheet. It would also help to determine the maturity of Funsheet.

8.4 Distributed computing with Funsheet

At the moment, the computational power of Funsheet is limited to the computer it is running on. We imagine a potential future extension to Funsheet, that allows it to interface with a distributed backend, such as a cloud platform. Concretely, it could be the introduction of a *define cloud function* that instead of creating the SDF in Funsheet locally, would redirect to a cloud provider running the distributed version of Funsheet. This could be an advantage in situations where more computing power is required. Using a cloud backend would introduce a transmission overhead, which could turn out to be negligible if the calculations are big enough.

8.5 The cause of higher execution times of benchmarks in Funsheet compared to Funcalc

It would be very interesting to investigate why Funsheet has a larger overhead on the benchmarking of functions compared to Funcalc. This is likely because Excel somehow is interfering the execution of the actual benchmarking. If it is possible to uncover why this overhead occurs, it could potentially be possible to avoid or reduce this overhead.

Chapter 9

Conclusion

In this thesis we described the development of Funsheet, a Microsoft Excel add-in, integrating the concepts from Funcalc into Excel.

We set out to introduce the concept of sheet-defined functions (SDF) in Microsoft Excel, in a functional and a user-friendly manner. We believe we have succeeded in this. We started experimenting with different technologies to identify the best performing ones for implementing user-defined functions (UDF) in Excel. From our experiment results, we conclude that UDFs built with Excel-DNA provides the highest performance. We analysed the features of Funcalc and the user experience of Microsoft Excel. From this analysis we formed a specification as a series of design goals. We developed a design to fulfill the design goals. We think we succeeded in creating a design in which all design goals were taken into account. We implemented a functioning and user-friendly Excel add-in to realize the design, while keeping it portable and maintainable. We succeeded in keeping the implementation true to the design. We developed a series of tests to verify that all design goals were satisfied. We succeeded in achieving all of our design goals, except one. We partly succeeded in realizing design goal QG1 concerning performance. This is discussed in section 9.4.

In the following sections we discuss the experiments, design, implementation and test more specifically.

9.1 Initial Experiments

We started out researching different technologies for implementing UDFs in Excel, to discover the ones currently available. From our research we found three alternatives:

VBA, COM Interop and Excel-DNA. We conducted a series of experiments to measure the different technologies for overhead of calling a UDF and performance when marshalling large amounts of data. Based on our results, we concluded that Excel-DNA performs far better than the other technologies, as stated in section 4.3.4.

9.2 Design

We analysed Funcalc to uncover the workings of the `DEFINE`, `CLOSURE`, `SPECIALIZE`, `APPLY` and `BENCHMARK` functions, to form our technical design goals. We examined the Microsoft Excel user interface and considered how to integrate and show information from Funcalc in Excel, while maintaining the Excel user experience. From our experiences we defined a series of usability goals. We formed a collection of quality goals to address our desire of creating a maintainable and portable solution. We think our design goals provided a strong foundation for a solid design, as discussed in the design evaluation 5.4.

Taking the results of the experiments and the design goals into consideration, we discussed the feasibility of each of the technology options. Based on our rationales in the discussion, we showed that Excel-DNA was the obvious choice to use in the development of Funsheet (see section 5.2).

Based on our design goals, we designed a solution. We used the separation of concerns principle to achieve a design with a high degree of maintainability and low coupling. We strove to make a minimum amount of changes to the original Funcalc source, in which we succeeded (see all changes in Appendix A). We ensured that all design goals were reflected in the design, and are thus confident of our design.

9.3 Implementation

We implemented the `DEFINE`, `CLOSURE`, `SPECIALIZE`, `APPLY` and `BENCHMARK` functions according to the design. All functional goals were fulfilled through the implementation of these functions, except one. The design goal TG2 concerning the ability to edit an existing SDF was not achieved. We discuss a possible implementation of this in section 8.1 of the Future Work chapter. Apart from this, we believe that we achieved a highly functional and complete realization of the functions.

To assist the functions in Funsheet, we implemented a consistent method of converting values and error values between Funcalc and Microsoft Excel. We succeeded in creating a versatile conversion system conforming to the design.
We have integrated a custom Excel ribbon into Excel to achieve a consistent user experience, while still providing the desired information from Funcalc. Through the implementation of the ribbon we realized our design goals concerning usability.

9.4 Test

We developed 9 tests to verify the implementation of the technical goals and the performance goal. We designed the tests such that the implementation of each technical goal was covered by at least one test, with the exception of *TG2: Support for editing previously defined SDFs*. The tests helped us to uncover bugs and poorly performing implementations. We think our tests of the technical goals has contributed to a higher quality of Funsheet by exposing errors that we were not initially aware of, as stated in section 7.3. We developed two test cases to verify design goal QG1 concerning performance. The test showed that Funsheet is slightly slower than our design goal allowed when benchmarking `TRIAREA`. The benchmarking of the more complex `GOALSEEK`, proved to be performining within the margins of our design goal, but still slower than Funcalc. We came to the conclusion that the reduction in performance was out of our control, as discussed in section 7.2.9.5 and section 7.3.
In section 7.3 we state that the Funsheet add-in is not ready to be used in production, but should undergo tests in a real world setting, e.g. be used in SDFs specific to banking, etc., and a user study should be conducted to see how the average Excel user reponds to the solution.

Appendix A

Source Code Documentation

A.1 Excelcalc.ApplyFunction Class Reference

This class contains the entry point for calling closures and specializations.

Static Public Member Functions

- static object **Apply** (string closureName, object input1, object input2, object input3, object input4, object input5, object input6, object input7, object input8, object input9)

  Applies the specified arguments to a closure or specialization.

A.1.1 Detailed Description

This class contains the entry point for calling closures and specializations.

A.1.2 Member Function Documentation

A.1.2.1 static object Excelcalc.ApplyFunction.Apply (string *closureName*, object *input1*, object *input2*, object *input3*, object *input4*, object *input5*, object *input6*, object *input7*, object *input8*, object *input9*) [static]

Applies the specified arguments to a closure or specialization.

Parameters

closureName	Name of the closure or specialization.
input1	The input1.
input2	The input2.
input3	The input3.
input4	The input4.
input5	The input5.
input6	The input6.
input7	The input7.
input8	The input8.
input9	The input9.

Returns

Result of executing the closure or specialization.

The documentation for this class was generated from the following file:

- C:/Projects/MasterThesis/Excelcalc/Excelcalc/**ApplyFunction.cs**

A.2 Excelcalc.BenchmarkFunction Class Reference

This class contains the entry point for benchmarking SDFs in Excel.

Static Public Member Functions

- static object **Benchmark** (string closureName, double iterations)

  Benchmarks the specified closure.

A.2.1 Detailed Description

This class contains the entry point for benchmarking SDFs in Excel.

A.2.2 Member Function Documentation

A.2.2.1 static object Excelcalc.BenchmarkFunction.Benchmark (string *closureName,* double *iterations*) [static]

Benchmarks the specified closure.

Parameters

closureName	Name of the closure.
iterations	Number of iterations. Used to get a more statistically significant result.

Returns

An integer of the time spent on each iteration in nano seconds.

The documentation for this class was generated from the following file:

- C:/Projects/MasterThesis/Excelcalc/Excelcalc/**BenchmarkFunction.cs**

A.3 Excelcalc.ClosureFunction Class Reference

This class contains the entry point for defining closures (higher order functions) in Excel.

Static Public Member Functions

- static object **Closure** (string function, object input1, object input2, object input3, object input4, object input5, object input6, object input7, object input8, object input9)

  Registers a closure with the specified arguments.

A.3.1 Detailed Description

This class contains the entry point for defining closures (higher order functions) in Excel.

A.3.2 Member Function Documentation

A.3.2.1 static object Excelcalc.ClosureFunction.Closure (string *function,* object *input1,* object *input2,* object *input3,* object *input4,* object *input5,* object *input6,* object *input7,* object *input8,* object *input9*) [static]

Registers a closure with the specified arguments.

Parameters

function	The SDF or closure to base the new closure on.
input1	The input1.
input2	The input2.
input3	The input3.
input4	The input4.
input5	The input5.
input6	The input6.
input7	The input7.
input8	The input8.
input9	The input9.

Returns

Returns a string representing the closure.

The documentation for this class was generated from the following file:

- C:/Projects/MasterThesis/Excelcalc/Excelcalc/**ClosureFunction.cs**

A.4 Excelcalc.Closures.Closure Class Reference

Represents a closure or compiled closure entity.

Inheritance diagram for Excelcalc.Closures.Closure:

Public Member Functions

- bool **Equals** (**Closure** other)

  Returns a bool indicating if two closures are equal. It compares the name of the SDF and the inuput from which the closure is based on.

Properties

- string **Name** `[get, set]`

  Gets or sets the name of the closure.

- bool **IsCompiled** `[get, set]`

Gets or sets a value indicating whether this instance is compiled.

- FunctionValue **Fv** `[get, set]`

  Gets or sets the FunctionValue that the closure is referencing. This will be autonomous if the closure is compiled.

- string **ReferenceTo** `[get, set]`

  A string indicating which SDF or closure the instance is referencing.

- object[] **Inputs** `[get, set]`

  Gets or sets the inputs to the closure.

- string **Details** `[get]`

  Gets the closure details to display in the Closure UI dialog.

Private Member Functions

- string **BuildDetails** ()

  Formats the closure details as a string in order to show it in the GUI

Private Attributes

- string **_details**

A.4.1 Detailed Description

Represents a closure or compiled closure entity.

A.4.2 Member Function Documentation

A.4.2.1 string Excelcalc.Closures.Closure.BuildDetails ( ) `[private]`

Formats the closure details as a string in order to show it in the GUI

Returns

A string containing the closure details.

A.4.2.2 bool Excelcalc.Closures.Closure.Equals (Closure *other*)

Returns a bool indicating if two closures are equal. It compares the name of the SDF and the inuput from which the closure is based on.

Parameters

other	An object to compare with this object.

Returns

True if the instances are equal. False otherwise.

A.4.3 Member Data Documentation

A.4.3.1 string Excelcalc.Closures.Closure._details [private]

A.4.4 Property Documentation

A.4.4.1 string Excelcalc.Closures.Closure.Details [get]

Gets the closure details to display in the Closure UI dialog.

A.4.4.2 FunctionValue Excelcalc.Closures.Closure.Fv [get], [set]

Gets or sets the FunctionValue that the closure is referencing. This will be autonomous if the closure is compiled.

A.4.4.3 object [] Excelcalc.Closures.Closure.Inputs [get], [set]

Gets or sets the inputs to the closure.

A.4.4.4 bool Excelcalc.Closures.Closure.IsCompiled [get], [set]

Gets or sets a value indicating whether this instance is compiled.

A.4.4.5 string Excelcalc.Closures.Closure.Name [get], [set]

Gets or sets the name of the closure.

A.4.4.6 string Excelcalc.Closures.Closure.ReferenceTo [get], [set]

A string indicating which SDF or closure the instance is referencing.
The documentation for this class was generated from the following file:

- C:/Projects/MasterThesis/Excelcalc/Excelcalc/Closures/**Closure.cs**

A.5 Excelcalc.Closures.ClosureContext Class Reference

Used to register and manage closures and specializations (compiled closures).

Static Package Functions

- static object **RegisterClosure** (string function, object[] inputs, bool returnObserver)

  Registers a closure in ExcelCalc and Corecalc.

- static object **RegisterCompiledClosure** (string closureName)

  Registers and compiles an already registered closure in Corecalc. It can then be used from Excelcalc.

- static bool **Contains** (string closureName)

  Has the closure been registered?

- static **Closure GetClosure** (string closureName)

  Retrieves the closure. Returns null if not found.

Properties

- static IDictionary< string,
  Closure > **Closures** `[get, set]`
- static IDictionary< string,
  SdfInfo > **SdfInfos** `[get, set]`

Static Private Member Functions

- static **ClosureContext** ()

  *Initializes the **ClosureContext** (p. 95) class.*

- static void **UnregisterClosure** (string closureName)

  Unregisters the closure from ExcelCalc and Corecalc.

Static Private Attributes

- static readonly Applier **_closureFunc**

A.5.1 Detailed Description

Used to register and manage closures and specializations (compiled closures).

A.5.2 Constructor & Destructor Documentation

A.5.2.1 static Excelcalc.Closures.ClosureContext.ClosureContext ( ) [static], [private]

Initializes the **ClosureContext** (p. 95) class.

A.5.3 Member Function Documentation

A.5.3.1 static bool Excelcalc.Closures.ClosureContext.Contains (string *closureName*) [static], [package]

Has the closure been registered?

Parameters

closureName	

Returns

boole

A.5.3.2 static Closure Excelcalc.Closures.ClosureContext.GetClosure (string *closureName*) [static], [package]

Retrieves the closure. Returns null if not found.

Parameters

closureName	Name of the closure.

Returns

Closure (p. 92)

A.5.3.3 static object Excelcalc.Closures.ClosureContext.RegisterClosure (string *function*, object[] *inputs*, bool *returnObserver*) [static], [package]

Registers a closure in ExcelCalc and Corecalc.

Parameters

function	The function to reference in the closure.
inputs	The inputs.

Returns

An Excel cell observer representing the closure.

A.5.3.4 static object Excelcalc.Closures.ClosureContext.RegisterCompiledClosure (string *closureName*) `[static]`, `[package]`

Registers and compiles an already registered closure in Corecalc. It can then be used from Excelcalc.

Parameters

closureName	

Returns

Excel cell observer representing the compiled closure.

A.5.3.5 static void Excelcalc.Closures.ClosureContext.UnregisterClosure (string *closureName*) [static], [private]

Unregisters the closure from ExcelCalc and Corecalc.
Parameters

closureName	

A.5.4 Member Data Documentation

A.5.4.1 readonly Applier Excelcalc.Closures.ClosureContext._closureFunc [static], [private]

A.5.5 Property Documentation

A.5.5.1 IDictionary<string, Closure> Excelcalc.Closures.-ClosureContext.Closures [static], [get], [set], [package]

A.5.5.2 IDictionary<string, SdfInfo> Excelcalc.Closures.-ClosureContext.SdfInfos [static], [get], [set], [package]

The documentation for this class was generated from the following file:

- C:/Projects/MasterThesis/Excelcalc/Excelcalc/Closures/**ClosureContext.cs**

A.6 Excelcalc.CorecalcRegistration.CorecalcDynamic-CellVisitor Class Reference

A cell visitor used to visit the Corecalc cell types and build up the workbook based on input from Excel.

Package Functions

- **CorecalcDynamicCellVisitor** (Workbook workbook, IDictionary< string, Sheet > sheetLookup)

*Initializes a new instance of the **CorecalcDynamicCellVisitor** (p. 98) class.*

- void **VisitOutput** (ExcelReference output)

  *Visits the output cell. Called from **CorecalcRegistrator** (p. 106)*

- void **Visit** (Cell cell, ExcelReference reference)

  Sets the current sheet based on the current Excel reference. Redirects the cell to the correct cell visitor method.

- void **Visit** (Formula formula, ExcelReference reference)

  Visits the specified formula cell. This method is also able to detect Excel functions and create/register these as needed.

- void **Visit** (NumberCell numberCell, ExcelReference reference)

  Visits the specified number cell.

- void **Visit** (TextCell textCell, ExcelReference reference)

  Visits the specified text cell.

Private Attributes

- readonly **CorecalcDynamicExprVisitor _exprVisitor**
- readonly Workbook **_workbook**
- readonly IDictionary< string, Sheet > **_sheetLookup**
- Sheet **_currentSheet**
- HashSet< FullCellAddr > **_cellsVisited**

A.6.1 Detailed Description

A cell visitor used to visit the Corecalc cell types and build up the workbook based on input from Excel.

A.6.2 Constructor & Destructor Documentation

A.6.2.1 Excelcalc.CorecalcRegistration.CorecalcDynamicCellVisitor.-CorecalcDynamicCellVisitor (Workbook *workbook,* IDictionary< string, Sheet > *sheetLookup*) [package]

Initializes a new instance of the **CorecalcDynamicCellVisitor** (p. 98) class.

Parameters

workbook	The workbook.
sheetLookup	A dictionary of the sheet names and sheets used by the SDF to be registered.

A.6.3 Member Function Documentation

A.6.3.1 void Excelcalc.CorecalcRegistration.CorecalcDynamicCellVisitor.Visit (Cell *cell*, ExcelReference *reference*) [package]

Sets the current sheet based on the current Excel reference. Redirects the cell to the correct cell visitor method.

Parameters

cell	The cell.
reference	The Excel reference.

A.6.3.2 void Excelcalc.CorecalcRegistration.CorecalcDynamicCellVisitor.Visit (Formula *formula*, ExcelReference *reference*) [package]

Visits the specified formula cell. This method is also able to detect Excel functions and create/register these as needed.

Parameters

formula	The formula cell.
reference	The Excel reference.

A.6.3.3 void Excelcalc.CorecalcRegistration.CorecalcDynamicCellVisitor.Visit (NumberCell *numberCell*, ExcelReference *reference*) [package]

Visits the specified number cell.

Parameters

numberCell	The number cell.
reference	The Excel reference.

A.6.3.4 void Excelcalc.CorecalcRegistration.CorecalcDynamicCellVisitor.Visit (TextCell *textCell*, ExcelReference *reference*) [package]

Visits the specified text cell.

Parameters

textCell	The text cell.
reference	The Excel reference.

A.6.3.5 void Excelcalc.CorecalcRegistration.CorecalcDynamicCellVisitor.VisitOutput (ExcelReference *output*) [package]

Visits the output cell. Called from **CorecalcRegistrator** (p. 106)

Parameters

output	The output.

A.6.4 Member Data Documentation

A.6.4.1 HashSet<FullCellAddr> Excelcalc.CorecalcRegistration.CorecalcDynamicCellVisitor._cellsVisited [private]

A.6.4.2 Sheet Excelcalc.CorecalcRegistration.CorecalcDynamicCellVisitor._currentSheet [private]

A.6.4.3 readonly CorecalcDynamicExprVisitor Excelcalc.CorecalcRegistration.CorecalcDynamicCellVisitor._exprVisitor [private]

A.6.4.4 readonly IDictionary<string, Sheet> Excelcalc.CorecalcRegistration.CorecalcDynamicCellVisitor._sheetLookup [private]

A.6.4.5 readonly Workbook Excelcalc.CorecalcRegistration.CorecalcDynamicCellVisitor._workbook [private]

The documentation for this class was generated from the following file:

- C:/Projects/MasterThesis/Excelcalc/Excelcalc/CorecalcRegistration/**CorecalcDynamicCellVisitor.cs**

A.7 Excelcalc.CorecalcRegistration.CorecalcDynamicExprVisitor Class Reference

An expression visitor used to visit the Corecalc expression types and build up the workbook based on input from Excel.

Package Functions

- **CorecalcDynamicExprVisitor** (**CorecalcDynamicCellVisitor** cellVisitor, IDictionary< string, Sheet > sheetLookup)

  Initializes a new instance of the ***CorecalcDynamicExprVisitor*** *(*p. *102)* *class.*

- void **Visit** (Expr expr, ExcelReference reference)

  Redirects to the correct expression visitor method.

- void **Visit** (CellRef cellRef, ExcelReference reference)

  Visits the specified cell reference expression.

- void **Visit** (FunCall funCall, ExcelReference reference)

  Visits the specified funtion call expression.

- void **Visit** (CellArea cellArea, ExcelReference reference)

  Visits the specified cell area expression.

- void **Visit** (NumberConst numberConst, ExcelReference reference)

  Visits the specified number constant expression.

- void **Visit** (TextConst numberConst, ExcelReference reference)

  Visits the specified number constant expression.

Private Attributes

- readonly **CorecalcDynamicCellVisitor _cellVisitor**
- readonly IDictionary< string,
  Sheet > **_sheetLookup**

A.7.1 Detailed Description

An expression visitor used to visit the Corecalc expression types and build up the workbook based on input from Excel.

A.7.2 Constructor & Destructor Documentation

A.7.2.1 Excelcalc.CorecalcRegistration.CorecalcDynamicExprVisitor.-CorecalcDynamicExprVisitor (CorecalcDynamicCellVisitor *cellVisitor*, IDictionary< string, Sheet > *sheetLookup*) [package]

Initializes a new instance of the **CorecalcDynamicExprVisitor** (p. 102) class.

Parameters

cellVisitor	The cell visitor.
sheetLookup	A dictionary of the sheet names and sheets used by the SDF to be registered.

A.7.3 Member Function Documentation

A.7.3.1 void Excelcalc.CorecalcRegistration.CorecalcDynamicExprVisitor.Visit (Expr *expr*, ExcelReference *reference*) [package]

Redirects to the correct expression visitor method.

Parameters

expr	The expression
reference	The Excel reference.

A.7.3.2 void Excelcalc.CorecalcRegistration.CorecalcDynamicExprVisitor.Visit (CellRef *cellRef*, ExcelReference *reference*) [package]

Visits the specified cell reference expression.

Parameters

cellRef	The cell reference expression.
reference	The Excel reference.

A.7.3.3 void Excelcalc.CorecalcRegistration.CorecalcDynamicExprVisitor.Visit (FunCall *funCall*, ExcelReference *reference*) [package]

Visits the specified funtion call expression.

Parameters

funCall	The function call expression.
reference	The Excel reference.

A.7.3.4 void Excelcalc.CorecalcRegistration.CorecalcDynamicExprVisitor.Visit (CellArea *cellArea*, ExcelReference *reference*) [package]

Visits the specified cell area expression.

Parameters

cellArea	The cell area expression.
reference	The Excel reference.

A.7.3.5 void Excelcalc.CorecalcRegistration.CorecalcDynamicExprVisitor.Visit (NumberConst *numberConst*, ExcelReference *reference*) [package]

Visits the specified number constant expression.

Parameters

numberConst	The number constant expression.
reference	The Excel reference.

A.7.3.6 void Excelcalc.CorecalcRegistration.CorecalcDynamicExprVisitor.Visit (TextConst *numberConst*, ExcelReference *reference*) [package]

Visits the specified number constant expression.

Parameters

numberConst	The number constant expression.
reference	The Excel reference.

A.7.4 Member Data Documentation

A.7.4.1 readonly CorecalcDynamicCellVisitor Excelcalc.CorecalcRegistration.CorecalcDynamicExprVisitor._cellVisitor [private]

A.7.4.2 readonly IDictionary<string, Sheet> Excelcalc.CorecalcRegistration.CorecalcDynamicExprVisitor._sheetLookup [private]

The documentation for this class was generated from the following file:

- C:/Projects/MasterThesis/Excelcalc/Excelcalc/CorecalcRegistration/**CorecalcDynamicExprVisitor.cs**

A.8 Excelcalc.CorecalcRegistration.CorecalcRegistrator Class Reference

This class is the entry point for registering and unregistering SDFs from Excel in Corecalc.

Static Package Functions

- static string **RegisterSdf** (string sdfName, ExcelReference output, List< ExcelReference > inputs, ExcelReference callingCell)

  Registers the SDF in Corecalc.

- static void **UnregisterSdf** (string sdfName)

  Unregisters the SDF in Corecalc.

- static string **RegisterPlaceholder** (string sdfName, ExcelReference callingCell)

Properties

- static Workbook **Workbook** `[get]`

  The Workbook is made static as the Workbook constructor removes all SDFs.

Static Private Attributes

- static **Workbook _workbook**

A.8.1 Detailed Description

This class is the entry point for registering and unregistering SDFs from Excel in Corecalc.

A.8.2 Member Function Documentation

A.8.2.1 static string Excelcalc.CorecalcRegistration.CorecalcRegistrator.RegisterPlaceholder (string *sdfName,* ExcelReference *callingCell*) `[static]`, `[package]`

A.8.2.2 static string Excelcalc.CorecalcRegistration.CorecalcRegistrator.RegisterSdf (string *sdfName,* ExcelReference *output,* List< ExcelReference > *inputs,* ExcelReference *callingCell*) `[static]`, `[package]`

Registers the SDF in Corecalc.

Parameters

sdfName	Name of the SDF.
output	The output cell reference.
inputs	The input cell references.
callingCell	The calling cell reference.

Returns

The string to show in the cell of the defined SDF

A.8.2.3 static void Excelcalc.CorecalcRegistration.CorecalcRegistrator.UnregisterSdf (string *sdfName*) [static], [package]

Unregisters the SDF in Corecalc.

Parameters

sdfName	Name of the SDF.

A.8.3 Member Data Documentation

A.8.3.1 Workbook Excelcalc.CorecalcRegistration.CorecalcRegistrator._workbook [static], [private]

A.8.4 Property Documentation

A.8.4.1 Workbook Excelcalc.CorecalcRegistration.CorecalcRegistrator.Workbook [static], [get], [package]

The Workbook is made static as the Workbook constructor removes all SDFs.

The workbook.

The documentation for this class was generated from the following file:

- C:/Projects/MasterThesis/Excelcalc/Excelcalc/CorecalcRegistration/**CorecalcRegistrator.cs**

A.9 Excelcalc.DefineFunction Class Reference

This class contains the entry point for defining SDFs in Excel using Excel-DNA.

Static Public Member Functions

- static object **Define** (string sdfName, object output, object input1, object input2, object input3, object input4, object input5, object input6, object input7, object input8, object input9)

  Creates the specified SDF. The Define method is automatically registered as an UDF in Excel at startup.

Static Package Functions

- static void **RegisterSdfsInQueue** ()
- static void **SetIsInitialized** (bool value)

  Sets the is initialized while holding the exclusive lock.

Static Private Member Functions

- static **DefineFunction** ()

  *Initializes the **DefineFunction** (*p. *108)* *class.*

- static void **PromoteToFunctionSheet** (ExcelReference caller)

  Promotes the sheet of the calling cell to a function sheet by prepending the sheet name with the '@' symbol. The action of changing the name is executed asynchroniously as a macro.

- static object **CreateCellObserver** (string sdfName)

  Creates the cell observer. The cell observer is used to monitor the cell calling the DEFINE UDF for deletion.

- static void **UnregisterSdf** (string sdfName)

  Unregisters the SDF.

Static Private Attributes

- static bool **IsInitialized** = false
- static object **_lock** = new object()
- static Dictionary< string,

  Tuple< ExcelReference,

  ExcelReference, List

  < ExcelReference >, string > > **DefineQueue**

A.9.1 Detailed Description

This class contains the entry point for defining SDFs in Excel using Excel-DNA.

A.9.2 Constructor & Destructor Documentation

A.9.2.1 static Excelcalc.DefineFunction.DefineFunction ( ) [static], [private]

Initializes the **DefineFunction** (p. 108) class.

A.9.3 Member Function Documentation

A.9.3.1 static object Excelcalc.DefineFunction.CreateCellObserver (string *sdfName*) [static], [private]

Creates the cell observer. The cell observer is used to monitor the cell calling the DEFINE UDF for deletion.

Parameters

sdfName	Name of the SDF.

Returns

Returns an instance of **Excelcalc.ExcelRegistration.ExcelCellHandle** (p. 112) that implements IExcelObservable

A.9.3.2 static object Excelcalc.DefineFunction.Define (string *sdfName*, object *output*, object *input1*, object *input2*, object *input3*, object *input4*, object *input5*, object *input6*, object *input7*, object *input8*, object *input9*) [static]

Creates the specified SDF. The Define method is automatically registered as an UDF in Excel at startup.

Parameters

sdfName	Name of the SDF to be created.

output	Content of the output cell.
input1	Content of the input cell 1. Optional.
input2	Content of the input cell 1. Optional.
input3	Content of the input cell 1. Optional.
input4	Content of the input cell 1. Optional.
input5	Content of the input cell 1. Optional.
input6	Content of the input cell 1. Optional.
input7	Content of the input cell 1. Optional.
input8	Content of the input cell 1. Optional.
input9	Content of the input cell 1. Optional.

Returns

Returns an object which can be any of the types that Excel is able to interpret.

Exceptions

System.Argument-Exception	

A.9.3.3 static void Excelcalc.DefineFunction.PromoteToFunctionSheet (ExcelReference *caller*) [static], [private]

Promotes the sheet of the calling cell to a function sheet by prepending the sheet name with the '@' symbol. The action of changing the name is executed asynchroniously as a macro.

Parameters

caller	The calling cell.

A.9.3.4 static void Excelcalc.DefineFunction.RegisterSdfsInQueue ( ) [static], [package]

A.9.3.5 static void Excelcalc.DefineFunction.SetIsInitialized (bool *value*) [static], [package]

Sets the is initialized while holding the exclusive lock.

Parameters

value	if set to `true` [value].

A.9.3.6 static void Excelcalc.DefineFunction.UnregisterSdf (string *sdfName*) [static], [private]

Unregisters the SDF.

Parameters

sdfName	Name of the SDF.

A.9.4 Member Data Documentation

A.9.4.1 object Excelcalc.DefineFunction._lock = new object() [static], [private]

A.9.4.2 Dictionary<string, Tuple<ExcelReference, ExcelReference, List<ExcelReference>, string> > Excelcalc.DefineFunction.DefineQueue [static], [private]

Initial value:

```
=
        new Dictionary<string, Tuple<ExcelReference, ExcelReference, List<ExcelReference>, string>>()
```

A.9.4.3 bool Excelcalc.DefineFunction.IsInitialized = false [static], [private]

The documentation for this class was generated from the following file:

- C:/Projects/MasterThesis/Excelcalc/Excelcalc/**DefineFunction.cs**

A.10 Excelcalc.ExcelRegistration.ExcelCellHandle Class Reference

This class implements a custom cell observer that is disposed when the cell is deleted from Excel.

Inheritance diagram for Excelcalc.ExcelRegistration.ExcelCellHandle:

Classes

- class **SdfDisposer**

  Private inner class used to wrap the dispose action.

Public Member Functions

- **ExcelCellHandle** (object returnValue, ExcelAction onDispose)

  Initializes a new instance of the ExcelCellHandle class.

- IDisposable **Subscribe** (IExcelObserver observer)

  Subscribes the specified observer.

Static Public Member Functions

- static object **CreateHandle** (string callerFunctionName, object callerParameters, ExcelFunc getReturnValue, ExcelAction onDispose)

  Creates the handle.

Private Attributes

- readonly ExcelAction **_disposeAction**
- readonly object **_returnValue**

A.10.1 Detailed Description

This class implements a custom cell observer that is disposed when the cell is deleted from Excel.

A.10.2 Constructor & Destructor Documentation

A.10.2.1 Excelcalc.ExcelRegistration.ExcelCellHandle.ExcelCellHandle (object *returnValue*, ExcelAction *onDispose*)

Initializes a new instance of the ExcelCellHandle class.

Parameters

returnValue	The return value.
onDispose	The on dispose.

A.10.3 Member Function Documentation

A.10.3.1 static object Excelcalc.ExcelRegistration.ExcelCellHandle.-CreateHandle (string *callerFunctionName,* object *callerParameters,* ExcelFunc *getReturnValue,* ExcelAction *onDispose*) [static]

Creates the handle.

Parameters

caller-Function-Name	Name of the caller function.
caller-Parameters	The caller parameters.
getReturn-Value	The get return value.
onDispose	The on dispose.

Returns

A.10.3.2 IDisposable Excelcalc.ExcelRegistration.Excel-CellHandle.Subscribe (IExcelObserver *observer*)

Subscribes the specified observer.

Parameters

observer	The observer.

Returns

A.10.4 Member Data Documentation

A.10.4.1 readonly ExcelAction Excelcalc.ExcelRegistration.ExcelCellHandle._disposeAction [private]

A.10.4.2 readonly object Excelcalc.ExcelRegistration.ExcelCellHandle._returnValue [private]

The documentation for this class was generated from the following file:

- C:/Projects/MasterThesis/Excelcalc/Excelcalc/ExcelRegistration/**ExcelCellHandle.cs**

A.11 Excelcalc.ExcelRegistration.ExcelFunCall Class Reference

Used to contain calls from an SDF into Excel.

Public Member Functions

- **ExcelFunCall** (string function)

  *Initializes a new instance of the **ExcelFunCall** (p. 115) class.*

- Value **ExcelCall** (Value[] values)

  *Calls the Excel function with the **FunctionName** (p. 116) and returns the result.*

Static Public Attributes

- static MethodInfo **ExcelCallMethodInfo** = typeof(**ExcelFunCall**).GetMethod("ExcelCall")

Properties

- string **FunctionName** [get, set]

  Gets or sets the name of the Excel function to be called from this instance.

A.11.1 Detailed Description

Used to contain calls from an SDF into Excel.

A.11.2 Constructor & Destructor Documentation

A.11.2.1 Excelcalc.ExcelRegistration.ExcelFunCall.ExcelFunCall (string *function*)

Initializes a new instance of the **ExcelFunCall** (p. 115) class.

Parameters

function	The name of the Excel function to be called.

A.11.3 Member Function Documentation

A.11.3.1 Value Excelcalc.ExcelRegistration.ExcelFunCall.ExcelCall (Value[] *values*)

Calls the Excel function with the **FunctionName** (p. 116) and returns the result.

Parameters

values	The Corecalc values to use for the call.

Returns

A value representing the result of the call to the Excel function

A.11.4 Member Data Documentation

A.11.4.1 MethodInfo Excelcalc.ExcelRegistration.ExcelFunCall.ExcelCallMethodInfo = typeof(ExcelFunCall).GetMethod("ExcelCall") [static]

A.11.5 Property Documentation

A.11.5.1 string Excelcalc.ExcelRegistration.ExcelFunCall.FunctionName [get], [set]

Gets or sets the name of the Excel function to be called from this instance.

The name of the function.

The documentation for this class was generated from the following file:

- C:/Projects/MasterThesis/Excelcalc/Excelcalc/ExcelRegistration/**ExcelFunCall.cs**

A.12 Excelcalc.ExcelRegistration.ExcelFunCallContext Class Reference

Used to keep track of which Excel functions are supported and which have been used from the registered SDFs.

Static Public Member Functions

- static Func< Value[], Value > **GenerateExcelCall** (string function, out MethodInfo methodInfo)

  Generates the Excel call delegate using the Linq.Expressions API. The delegate is creating an instance of ***ExcelFunCall*** *(p. 115) and invoking the ExcelCall method of that instance.*

Static Package Functions

- static void **Register** (string function)

  Creates a delegate for calling the Excel function. Registers the specified Excel function call delegate in Corecalc using the Function and FunctionInfo classes.

Static Package Attributes

- static ISet< string > **RegisteredFunctions** = new HashSet<string>()

  The supported Excel functions where a delegate for calling the function have been created.

- static ISet< string > **SupportedFunctions**

  The supported Excel functions.

Static Private Member Functions

- static **ExcelFunCallContext** ()

  Initializes the ***ExcelFunCallContext*** *(p. 117) class. A transient module is created in this assembly to contain all the Excel call delegates currently used in this run.*

- static void **RegisterSupportedFunctions** ()

Static Private Attributes

- static readonly ModuleBuilder **_moduleBuilder**

A.12.1 Detailed Description

Used to keep track of which Excel functions are supported and which have been used from the registered SDFs.

A.12.2 Constructor & Destructor Documentation

A.12.2.1 static Excelcalc.ExcelRegistration.ExcelFunCallContext.ExcelFunCallContext ( ) [static], [private]

Initializes the **ExcelFunCallContext** (p. 117) class. A transient module is created in this assembly to contain all the Excel call delegates currently used in this run.

A.12.3 Member Function Documentation

A.12.3.1 static Func<Value[], Value> Excelcalc.ExcelRegistration.ExcelFunCallContext.GenerateExcelCall (string *function*, out MethodInfo *methodInfo*) [static]

Generates the Excel call delegate using the Linq.Expressions API. The delegate is creating an instance of **ExcelFunCall** (p. 115) and invoking the ExcelCall method of that instance.

Parameters

function	The name of the supported Excel function.
methodInfo	The method information as an out parameter.

Returns

A.12.3.2 static void Excelcalc.ExcelRegistration.ExcelFunCallContext.Register (string *function*) [static], [package]

Creates a delegate for calling the Excel function. Registers the specified Excel function call delegate in Corecalc using the Function and FunctionInfo classes.

Parameters

function	The function.

A.12.3.3 static void Excelcalc.ExcelRegistration.ExcelFunCallContext.RegisterSupportedFunctions ( ) [static], [private]

A.12.4 Member Data Documentation

A.12.4.1 readonly ModuleBuilder Excelcalc.ExcelRegistration.ExcelFunCallContext._moduleBuilder [static], [private]

A.12.4.2 ISet<string> Excelcalc.ExcelRegistration.ExcelFunCallContext.RegisteredFunctions = new HashSet<string>() [static], [package]

The supported Excel functions where a delegate for calling the function have been created.

A.12.4.3 ISet<string> Excelcalc.ExcelRegistration.ExcelFunCallContext.SupportedFunctions [static], [package]

The supported Excel functions.

The documentation for this class was generated from the following file:

- C:/Projects/MasterThesis/Excelcalc/Excelcalc/ExcelRegistration/**ExcelFunCallContext.cs**

A.13 Excelcalc.ExcelcalcInitializer Class Reference

This class is used to enable Excelcalc to open saved workbooks.

Inheritance diagram for Excelcalc.ExcelcalcInitializer:

Public Member Functions

- void **AutoClose** ()

- void **AutoOpen** ()

  This method is fired when Excelcalc is discovered by Excel at startup. It hooks up delegates to the window activate event and to the workbook open event. These are used to preload SDFs from a previously saved workbook.

Private Member Functions

- void **app_WindowActivate** (Workbook Wb, Window Wn)

  *Signals the **DefineFunction** (p. 108) class to set the IsInitialized to true. This is to ensure that a new empty workbook is ready to register SDFs.*

- void **app_WorkbookOpen** (Workbook Wb)

  Executes a macro to register all SDF placeholders in the workbook. This is to ensure that a user can re-open a saved workbook.

Private Attributes

- Application **app** = ExcelDnaUtil.Application as Application

A.13.1 Detailed Description

This class is used to enable Excelcalc to open saved workbooks.

A.13.2 Member Function Documentation

A.13.2.1 void Excelcalc.ExcelcalcInitializer.app_WindowActivate (Workbook *Wb*, Window *Wn*) [private]

Signals the **DefineFunction** (p. 108) class to set the IsInitialized to true. This is to ensure that a new empty workbook is ready to register SDFs.

Parameters

Wb	The wb.
Wn	The wn.

A.13.2.2 void Excelcalc.ExcelcalcInitializer.app_WorkbookOpen (Workbook *Wb*) [private]

Executes a macro to register all SDF placeholders in the workbook. This is to ensure that a user can re-open a saved workbook.

Parameters

Wb	The wb.

A.13.2.3 void Excelcalc.ExcelcalcInitializer.AutoClose ( )

A.13.2.4 void Excelcalc.ExcelcalcInitializer.AutoOpen ( )

This method is fired when Excelcalc is discovered by Excel at startup. It hooks up delegates to the window activate event and to the workbook open event. These are used to preload SDFs from a previously saved workbook.

A.13.3 Member Data Documentation

A.13.3.1 Application Excelcalc.ExcelcalcInitializer.app = ExcelDnaUtil.Application as Application `[private]`

The documentation for this class was generated from the following file:

- C:/Projects/MasterThesis/Excelcalc/Excelcalc/**ExcelcalcInitializer.cs**

A.14 Excelcalc.RegisteredSdf Class Reference

Entity class to hold the information of a specific SDF registered in Excel.

Properties

- int **Col** `[get, set]`

  Gets or sets the column number.

- int **Row** `[get, set]`

  Gets or sets the row number.

- string **FullFormula** `[get, set]`

  Gets or sets the full formula.

- string **SdfCellText** `[get, set]`

  Gets or sets the SDF cell text generated from Corecalc.

A.14.1 Detailed Description

Entity class to hold the information of a specific SDF registered in Excel.

A.14.2 Property Documentation

A.14.2.1 int Excelcalc.RegisteredSdf.Col [get], [set]

Gets or sets the column number.

The col.

A.14.2.2 string Excelcalc.RegisteredSdf.FullFormula [get], [set]

Gets or sets the full formula.

The full formula.

A.14.2.3 int Excelcalc.RegisteredSdf.Row [get], [set]

Gets or sets the row number.

The row.

A.14.2.4 string Excelcalc.RegisteredSdf.SdfCellText [get], [set]

Gets or sets the SDF cell text generated from Corecalc.

The SDF cell text.

The documentation for this class was generated from the following file:

- C:/Projects/MasterThesis/Excelcalc/Excelcalc/**RegisteredSdf.cs**

A.15 Excelcalc.RegisteredSdfContext Class Reference

This class is used to keep track of all registered SDFs in Excel.

Static Package Functions

- static void **AddToRegisteredSdfs** (string sdfName, int column, int row, string fullFormula, string sdfCellText)

  *Adds a **RegisteredSdf** (p. 122) to the Sdfs dictionary.*

- static bool **SdfAlreadyExists** (string sdfName, int column, int row, string fullFormula)

  Check if the SDF already exists. If the formula or coordinates are different, it will return false.

Static Package Attributes

- static readonly Dictionary
  < string, **RegisteredSdf** > **Sdfs** = new Dictionary<string, **RegisteredSdf**>()

A.15.1 Detailed Description

This class is used to keep track of all registered SDFs in Excel.

A.15.2 Member Function Documentation

A.15.2.1 static void Excelcalc.RegisteredSdfContext.AddToRegisteredSdfs (string *sdfName,* int *column,* int *row,* string *fullFormula,* string *sdfCellText*) [static], [package]

Adds a **RegisteredSdf** (p. 122) to the Sdfs dictionary.

Parameters

sdfName	Name of the SDF.
column	The column number.
row	The row number.
fullFormula	The full formula.
sdfCellText	The SDF cell text.

A.15.2.2 static bool Excelcalc.RegisteredSdfContext.SdfAlreadyExists (string *sdfName,* int *column,* int *row,* string *fullFormula*) [static], [package]

Check if the SDF already exists. If the formula or coordinates are different, it will return false.

Parameters

sdfName	Name of the SDF.
column	The column.
row	The row.

fullFormula	The full formula.

Returns

Returns a boolean indicating whether the SDF exists.

A.15.3 Member Data Documentation

A.15.3.1 readonly Dictionary<string, RegisteredSdf> Excelcalc.-RegisteredSdfContext.Sdfs = new Dictionary<string, RegisteredSdf>() [static], [package]

The documentation for this class was generated from the following file:

- C:/Projects/MasterThesis/Excelcalc/Excelcalc/**RegisteredSdfContext.cs**

A.16 Excelcalc.ExcelRegistration.ExcelRegistrator Class Reference

This class is the entry point for doing all registration of SDFs in Excel. This includes setting up a function name in Excel on-the-fly, register a cell observer to monitor the definition and unregistering the function again.

Static Package Functions

- static void **RegisterSdf** (string sdfName, ExcelReference output, List< ExcelReference > inputs)

  Registers the SDF in Excel as an Excel UDF.

- static void **UnregisterSdf** (string sdfName)

  Unegisters the SDF asynchroniously from a macro.

Static Private Member Functions

- static Delegate **GenerateSdfCallDelegate** (string sdfName, int numberOf-Arguments)

  Generates the SDF call delegate to be fired each time the registered SDF is called from Excel.

Static Private Attributes

- static readonly MethodInfo **_toExcelObjectMethodInfo** = typeof(**ExtensionMethods**).GetMethod("ToExcelObject", new[]{typeof(Value)})
- static readonly MethodInfo **_toValueArrayMethodInfo** = typeof(**HelperMethods**).GetMethod("ToValueArray", new[] { typeof(object[]) })
- static readonly MethodInfo **_applyMethodInfo** = typeof(SdfInfo).GetMethod("Apply", new[] { typeof(Value[]) })

A.16.1 Detailed Description

This class is the entry point for doing all registration of SDFs in Excel. This includes setting up a function name in Excel on-the-fly, register a cell observer to monitor the definition and unregistering the function again.

A.16.2 Member Function Documentation

A.16.2.1 static Delegate Excelcalc.ExcelRegistration.ExcelRegistrator.GenerateSdfCallDelegate (string *sdfName,* int *numberOfArguments*) [static], [private]

Generates the SDF call delegate to be fired each time the registered SDF is called from Excel.

Parameters

sdfName	Name of the SDF.
numberOfArguments	The number of arguments.

Returns

A delegate representing the call to the SDF.

A.16.2.2 static void Excelcalc.ExcelRegistration.ExcelRegistrator.RegisterSdf (string *sdfName,* ExcelReference *output,* List< ExcelReference > *inputs*) [static], [package]

Registers the SDF in Excel as an Excel UDF.

Parameters

sdfName	Name of the SDF.
output	The output cell reference.
inputs	The input cel references.

A.16.2.3 static void Excelcalc.ExcelRegistration.ExcelRegistrator.-UnregisterSdf (string *sdfName*) [static], [package]

Unegisters the SDF asynchroniously from a macro.

The SDF is not completely deleted but is converted to a hidden macro. It is still possible to call the macro from VBA. This doesn't prevent recreation of the SDF later on in any way.

Parameters

sdfName	Name of the SDF.

A.16.3 Member Data Documentation

A.16.3.1 readonly MethodInfo Excelcalc.ExcelRegistration.Excel-Registrator._applyMethodInfo = typeof(SdfInfo).Get-Method("Apply", new[] { typeof(Value[]) }) [static], [private]

A.16.3.2 readonly MethodInfo Excelcalc.ExcelRegistration.-ExcelRegistrator._toExcelObjectMethodInfo = typeof(ExtensionMethods).GetMethod("ToExcelObject", new[]{typeof(Value)}) [static], [private]

A.16.3.3 readonly MethodInfo Excelcalc.ExcelRegistration.-ExcelRegistrator._toValueArrayMethodInfo = typeof(HelperMethods).GetMethod("ToValueArray", new[] { typeof(object[]) }) [static], [private]

The documentation for this class was generated from the following file:

- C:/Projects/MasterThesis/Excelcalc/Excelcalc/ExcelRegistration/**Excel-Registrator.cs**

A.17 Excelcalc.Util.ExtensionMethods Class Reference

Extension methods used to support Excelcalc.

Static Public Member Functions

- static string **GetSheetName** (this ExcelReference reference)

  Gets the name of the sheet from the current ExcelReference instance.

- static string **GetSheetName** (this string workBookAndSheet)

  Exstracts the sheet name from a string consisting of the Workbook name and Sheet name.

- static object **ToExcelObject** (this Value value)

  Converts a Corecalc Value to an Excel object.

Private Attributes

- const string **Pattern** = @"\[\w+\.?\w*\](@?\w+)"

A.17.1 Detailed Description

Extension methods used to support Excelcalc.

A.17.2 Member Function Documentation

A.17.2.1 static string Excelcalc.Util.ExtensionMethods.GetSheetName (this ExcelReference *reference*) [static]

Gets the name of the sheet from the current ExcelReference instance.
Parameters

reference	The reference.

Returns

The sheet name.

A.17.2.2 static string Excelcalc.Util.ExtensionMethods.GetSheetName (this string *workBookAndSheet*) [static]

Exstracts the sheet name from a string consisting of the Workbook name and Sheet name.

Parameters

workBook-AndSheet	The work book and sheet.

Returns

A.17.2.3 static object Excelcalc.Util.ExtensionMethods.ToExcelObject (this Value *value*) [static]

Converts a Corecalc Value to an Excel object.

Parameters

value	The value.

Returns

A.17.3 Member Data Documentation

A.17.3.1 const string Excelcalc.Util.ExtensionMethods.Pattern = @"\[\w+\.?\w*\](@?\w+)" [private]

The documentation for this class was generated from the following file:

- C:/Projects/MasterThesis/Excelcalc/Excelcalc/Util/**ExtensionMethods.cs**

A.18 Excelcalc.Util.HelperMethods Class Reference

Contains a collection of helper methods used by Excelcalc.

Static Public Member Functions

- static Value[] **ToValueArray** (params object[] args)

  Converts an array of argument objects to a Value array.

- static Value **ObjectToValue** (object value)

  Converts an object from Excel to a Corecalc Value.

Static Package Functions

- static Cell **Parse** (string cellContent, int col, int row)

  Parses the specified cell content.

A.18.1 Detailed Description

Contains a collection of helper methods used by Excelcalc.

A.18.2 Member Function Documentation

A.18.2.1 static Value Excelcalc.Util.HelperMethods.ObjectToValue (object *value*) [static]

Converts an object from Excel to a Corecalc Value.

Parameters

value	The value.

Returns

A Corecalc Value.

A.18.2.2 static Cell Excelcalc.Util.HelperMethods.Parse (string *cellContent*, int *col*, int *row*) [static], [package]

Parses the specified cell content.

Parameters

cellContent	Content of the cell.
col	The col.
row	The row.

Returns

Returns a specific cell class derived from the Corecalc Cell type. Returns null if the parsing fails.

A.18.2.3 static Value [] Excelcalc.Util.HelperMethods.ToValueArray (params object[] *args*) [static]

Converts an array of argument objects to a Value array.

Parameters

args	The arguments.

Returns

A Value array.

The documentation for this class was generated from the following file:

- C:/Projects/MasterThesis/Excelcalc/Excelcalc/Util/**HelperMethods.cs**

A.19 Excelcalc.SpecializeFunction Class Reference

This class contains the entry point for compiling closures into compiled closures called specializations.

Static Public Member Functions

- static object **Specialize** (string closureName)

  Compiles the closure with the specified name.

A.19.1 Detailed Description

This class contains the entry point for compiling closures into compiled closures called specializations.

A.19.2 Member Function Documentation

A.19.2.1 static object Excelcalc.SpecializeFunction.Specialize (string *closureName*) [static]

Compiles the closure with the specified name.

Parameters

closureName	Name of the closure.

Returns

A string representing the compiled closure.

The documentation for this class was generated from the following file:

- C:/Projects/MasterThesis/Excelcalc/Excelcalc/**SpecializeFunction.cs**

A.20 Excelcalc.UI.ExcelcalcRibbon Class Reference

Contains callback methods used by the Excelcalc Ribbon UI.

Inheritance diagram for Excelcalc.UI.ExcelcalcRibbon:

Public Member Functions

- void **ShowSDFs_onAction** (IRibbonControl control)

  Shows the SDF bytecode dialog.

- void **ShowClosures_onAction** (IRibbonControl control)

  Shows the dialog listing the closures and specializations.

- void **AboutFuncalc_onAction** (IRibbonControl control)

  Shows the About Funcalc dialog.

- void **SupportedExcelFunctions_onAction** (IRibbonControl control)

  Shows the supported excel functions dialog.

- void **RegisteredExcelFunctions_onAction** (IRibbonControl control)

  Shows the registered excel functions dialog.

- void **AboutExcelcalc_onAction** (IRibbonControl control)

  Shows the About Excelcalc dialog.

- void **ToggleExceptions_onAction** (IRibbonControl control, bool isActive)

  Toggles the exceptions on and off. Exceptions have the highest precedence if enabled.

- void **ToggleFuncalcErrors_onAction** (IRibbonControl control, bool isActive)

  Toggles the Funcalc errors on and off. Funcalc errors have lower precedence than exceptions if both are enabled.

- override string **GetCustomUI** (string RibbonID)

  Gets the custom UI from an embedded XML resource.

Static Public Attributes

- static bool **ShowFuncalcErrors**

Private Attributes

- readonly UnhandledExceptionHandler **_exceptionHandler**

  The exception delegate that will be run on exceptions when the showing of exceptions is enabled.

A.20.1 Detailed Description

Contains callback methods used by the Excelcalc Ribbon UI.

A.20.2 Member Function Documentation

A.20.2.1 void Excelcalc.UI.ExcelcalcRibbon.AboutExcelcalc_onAction (IRibbonControl *control*)

Shows the About Excelcalc dialog.

A.20.2.2 void Excelcalc.UI.ExcelcalcRibbon.AboutFuncalc_onAction (IRibbonControl *control*)

Shows the About Funcalc dialog.

A.20.2.3 override string Excelcalc.UI.ExcelcalcRibbon.GetCustomUI (string *RibbonID*)

Gets the custom UI from an embedded XML resource.
Returns

An XML string describing the Excelcalc Ribbon UI.

A.20.2.4 void Excelcalc.UI.ExcelcalcRibbon.RegisteredExcelFunctions_onAction (IRibbonControl *control*)

Shows the registered excel functions dialog.

A.20.2.5 void Excelcalc.UI.ExcelcalcRibbon.ShowClosures_onAction (IRibbonControl *control*)

Shows the dialog listing the closures and specializations.

A.20.2.6 void Excelcalc.UI.ExcelcalcRibbon.ShowSDFs_onAction (IRibbonControl *control*)

Shows the SDF bytecode dialog.

A.20.2.7 void Excelcalc.UI.ExcelcalcRibbon.SupportedExcelFunctions_onAction (IRibbonControl *control*)

Shows the supported excel functions dialog.

A.20.2.8 void Excelcalc.UI.ExcelcalcRibbon.ToggleExceptions_onAction (IRibbonControl *control,* bool *isActive*)

Toggles the exceptions on and off. Exceptions have the highest precedence if enabled.

A.20.2.9 void Excelcalc.UI.ExcelcalcRibbon.ToggleFuncalcErrors_onAction (IRibbonControl *control,* bool *isActive*)

Toggles the Funcalc errors on and off. Funcalc errors have lower precedence than exceptions if both are enabled.

A.20.3 Member Data Documentation

A.20.3.1 readonly UnhandledExceptionHandler Excelcalc.UI.ExcelcalcRibbon._exceptionHandler [private]

Initial value:

```
= ex =>
                {
                    var exc = ex as Exception;
                    var sb = new StringBuilder();
                    sb.AppendLine("Exception: ");
                    sb.Append(exc.ToString());
                    Exception inner = exc.InnerException;
                    while (inner != null)
                    {
                        sb.AppendLine("Inner Exception: ");
                        sb.Append(inner.ToString());
                        inner = inner.InnerException;
                    }
```

```
            return sb.ToString();
        }
```

The exception delegate that will be run on exceptions when the showing of exceptions is enabled.

A.20.3.2 bool Excelcalc.UI.ExcelcalcRibbon.ShowFuncalcErrors [static]

The documentation for this class was generated from the following file:

- C:/Projects/MasterThesis/Excelcalc/Excelcalc/UI/**ExcelcalcRibbon.cs**

A.21 Excelcalc.Util.ErrorConverter Class Reference

Static Package Functions

- static ErrorValue **Convert** (ExcelError error)

  Converts an Excel error to its equivalent Corecalc error.

- static ExcelError **Convert** (ErrorValue error)

  Converts a Corecalc error to its equivalent Excel error.

Static Private Member Functions

- static **ErrorConverter** ()

  Contains mappings between Corecalc errors and Excel errors and vice versa.

Static Private Attributes

- static readonly Dictionary
  < ExcelError, ErrorValue > **_toErrorValue**
- static readonly Dictionary
  < ErrorValue, ExcelError > **_toExcelError**

A.21.1 Constructor & Destructor Documentation

A.21.1.1 static Excelcalc.Util.ErrorConverter.ErrorConverter ( )

[static], [private]

Contains mappings between Corecalc errors and Excel errors and vice versa.

A.21.2 Member Function Documentation

A.21.2.1 static ErrorValue Excelcalc.Util.ErrorConverter.Convert (ExcelError *error*) [static], [package]

Converts an Excel error to its equivalent Corecalc error.

Parameters

error	The error.

Returns

A.21.2.2 static ExcelError Excelcalc.Util.ErrorConverter.Convert (ErrorValue *error*) [static], [package]

Converts a Corecalc error to its equivalent Excel error.

Parameters

error	The error.

Returns

A.21.3 Member Data Documentation

A.21.3.1 readonly Dictionary<ExcelError, ErrorValue> Excelcalc.Util.ErrorConverter._toErrorValue [static], [private]

A.21.3.2 readonly Dictionary<ErrorValue, ExcelError> Excelcalc.Util.ErrorConverter._toExcelError [static], [private]

The documentation for this class was generated from the following file:

- C:/Projects/MasterThesis/Excelcalc/Excelcalc/Util/**ErrorConverter.cs**

A.22 Excelcalc.Util.ExtensionMethods Class Reference

Extension methods used to support Excelcalc.

Static Public Member Functions

- static string **GetSheetName** (this ExcelReference reference)

  Gets the name of the sheet from the current ExcelReference instance.

- static string **GetSheetName** (this string workBookAndSheet)

Exstracts the sheet name from a string consisting of the Workbook name and Sheet name.

- static object **ToExcelObject** (this Value value)

  Converts a Corecalc Value to an Excel object.

Private Attributes

- const string **Pattern** = @"\[\w+\.?\w*\](@?\w+)"

A.22.1 Detailed Description

Extension methods used to support Excelcalc.

A.22.2 Member Function Documentation

A.22.2.1 static string Excelcalc.Util.ExtensionMethods.GetSheetName (this ExcelReference *reference*) [static]

Gets the name of the sheet from the current ExcelReference instance.
Parameters

reference	The reference.

Returns

The sheet name.

A.22.2.2 static string Excelcalc.Util.ExtensionMethods.GetSheetName (this string *workBookAndSheet*) [static]

Exstracts the sheet name from a string consisting of the Workbook name and Sheet name.
Parameters

workBook-AndSheet	The work book and sheet.

Returns

A.22.2.3 static object Excelcalc.Util.ExtensionMethods.ToExcelObject (this Value *value*) `[static]`

Converts a Corecalc Value to an Excel object.

Parameters

value	The value.

Returns

A.22.3 Member Data Documentation

A.22.3.1 const string Excelcalc.Util.ExtensionMethods.Pattern = @"\[\w+\.?\w*\](@?\w+)" [private]

The documentation for this class was generated from the following file:

- C:/Projects/MasterThesis/Excelcalc/Excelcalc/Util/**ExtensionMethods.cs**

A.23 Excelcalc.Util.HelperMethods Class Reference

Contains a collection of helper methods used by Excelcalc.

Static Public Member Functions

- static Value[] **ToValueArray** (params object[] args)

  Converts an array of argument objects to a Value array.

- static Value **ObjectToValue** (object value)

  Converts an object from Excel to a Corecalc Value.

Static Package Functions

- static Cell **Parse** (string cellContent, int col, int row)

  Parses the specified cell content.

A.23.1 Detailed Description

Contains a collection of helper methods used by Excelcalc.

A.23.2 Member Function Documentation

A.23.2.1 static Value Excelcalc.Util.HelperMethods.ObjectToValue (object *value*) [static]

Converts an object from Excel to a Corecalc Value.

Parameters

value	The value.

Returns

A Corecalc Value.

A.23.2.2 static Cell Excelcalc.Util.HelperMethods.Parse (string *cellContent*, int *col*, int *row*) [static], [package]

Parses the specified cell content.

Parameters

cellContent	Content of the cell.
col	The col.
row	The row.

Returns

Returns a specific cell class derived from the Corecalc Cell type. Returns null if the parsing fails.

A.23.2.3 static Value [] Excelcalc.Util.HelperMethods.ToValueArray (params object[] *args*) [static]

Converts an array of argument objects to a Value array.

Parameters

args	The arguments.

Returns

A Value array.

The documentation for this class was generated from the following file:

- C:/Projects/MasterThesis/Excelcalc/Excelcalc/Util/**HelperMethods.cs**

Appendix B

Funcalc Changelog

B.1 Primer

Changes to the Funcalc code base have been made to enable the integration Funcalc into Funsheet.
It is important to note that all changes made to the Funcalc assembly are non-breaking changes. This means that the changes does not affect the functionality or behavior of the stand-alone Funcalc application.

B.2 Changes

B.2.1 Exposing Funcalc internals to Funsheet

To allow Funsheet to reach Funcalc methods and members marked with the *internal* access modifier, the following line have been appended to the *AssemblyInfo.cs* file in Funcalc:

```
[assembly: AssemblyVersion("0.11.12.0")]
[assembly: AssemblyFileVersion("0.11.12.0")]
[assembly: InternalsVisibleTo("ExcelCalc")]
```

FIGURE B.1: Changes made to AssemblyInfo.cs.

B.2.2 Allow Funsheet to override pre-defined Funcalc functions

To allow Funsheet to substitute pre-defined functions in Funcalc with calls to a complementary built-in function in Excel a couple of changes have been made to the Funcalc source code.

In the constructor of the Function class the registration of a function have been allowed to overwrite an existing function. See figure B.3.

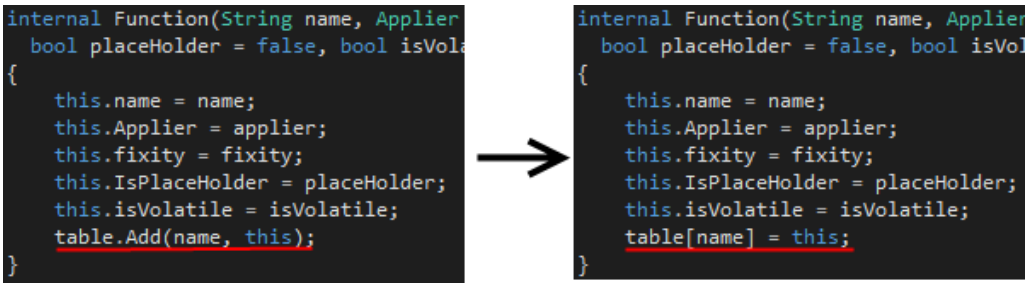

FIGURE B.2: Changes made to the constructor of the Function class.

In the constructor of the FunctionInfo class the registration of a function definition have been allowed to overwrite an existing function definition. See figure B.2.

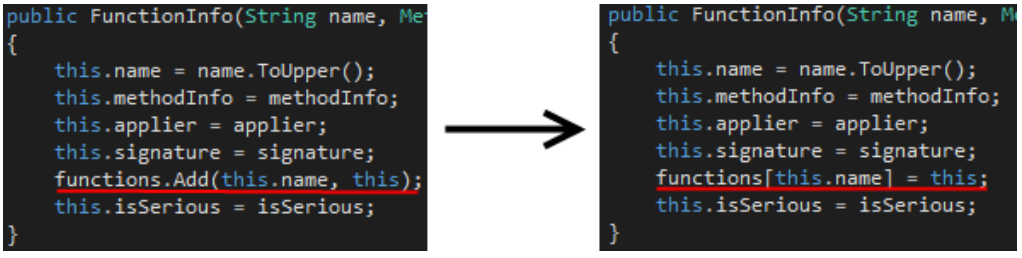

FIGURE B.3: Changes made to the constructor of the FunctionInfo class.

B.2.3 Grammar change to support newer Excel functions

Newer Excel functions contain a punctuation ex. *QUARTILE.INC*. To support these, changes have been made to the grammar stored in the file *Spreadsheet.ATG*. Figure B.4 shows the specific change.

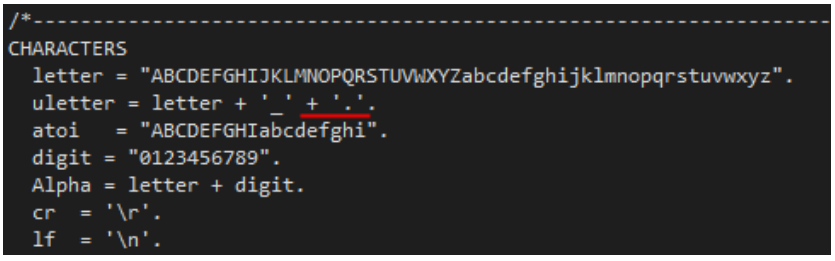

FIGURE B.4: Changes made to the grammar.

Using the Coco/R compiler generator, a new *Scanner.cs* file have been generated to reflect the changes in the grammar.

B.2.4 Ensuring deletion of deprecated specializations

In Funcalc the specializations are not removed completely when deleted from a sheet. This leads to an error if a user creates a specialization, deletes it and tries to recreate the specialization again. The following code shows the change made to the Funcalc source code to fix this small bug.

```
1  private static void Unregister(SdfInfo info)
2  {
3      sdfNameToInfo.Remove(info.name);
4      sdfDelegates[info.index] = sdfDeleted[info.arity];
5      sdfInfos[info.index] = null;
6      //Workaround to restore deleted specializations without getting #FUNERR:
       Function deleted.
7      var compiledClosure = specializations.FirstOrDefault(kvp => kvp.Value ==
       info);
8      if(compiledClosure.Key != null)
9      {
10         specializations.Remove(compiledClosure.Key);
11     }
12 }
```

Listing B.1: Line 7-11 added to ensure correct deletion of specializations

Appendix C

Funsheet Installation Guide

This chapter covers everything needed to run Funsheet. The guide has been created for the English version of Excel. Version with other languages might have different names for the visual elements referred to during this guide.
Note: The Funsheet had its name changed during implementation from Excelcalc. This is the reason many of the files have the Excelcalc pre-fix.

C.1 Pre-requisites

Microsoft Excel 2013 32-bit is required to run Funsheet.
Funsheet requires the following files to be located in the same folder to run: `Microsoft.Office.Interop.Excel.dll`, `ClrTest.Reflection.ILReader.dll`, `ClrTest.Reflection.ILVisualizer.dll`, `Excelcalc-AddIn.xll.config`, `Excelcalc-AddIn-packed.xll`, `Funcalc.exe`.

C.2 Install Funsheet

This section covers the installation of Funsheet. It is divided into 9 steps.

Step 1 Click the *File* tab in the ribbon menu of Microsoft Excel. The File menu appears.

Step 2 Click the *Options* button in the menu. The *Excel Options* window appears.

Step 3 Click the *Add-Ins* tab in the left side of the *Excel Options* window. The content of the window changes.

Step 4 Ensure that *Excel Add-ins* has been chosen in the dropdown menu in the bottom of the window.

Step 5 Click the *Go...* button. The current window is replaced with the *Add-Ins* window.

Step 6 Click the *Browse...* button. A file explorer appears.

Step 7 Browse to the folder where *Excelcalc-AddIn-packed.xll* is located.

Step 8 Choose *Excelcalc-AddIn-packed.xll* and click the *OK* button in the file explorer. The file explorer closes and you return to the *Add-Ins* window.

Step 9 Click the *OK* button of the *Add-Ins* window. You return to the Excel sheet. Notice the new *EXCELCALC* tab in the ribbon menu.

C.3 Uninstall Funsheet

This section covers the uninstallation of Funsheet. It is divided into 7 steps.

Step 1 Click the *File* tab in the ribbon menu of Microsoft Excel. The File menu appears.

Step 2 Click the *Options* button in the menu. The *Excel Options* window appears.

Step 3 Click the *Add-Ins* tab in the left side of the *Excel Options* window. The content of the window changes.

Step 4 Ensure that *Excel Add-ins* has been chosen in the dropdown menu in the bottom of the window.

Step 5 Click the *Go...* button. The current window is replaced with the *Add-Ins* window.

Step 6 Remove the check mark from the *Excelcalc Add-In* entry in the list.

Step 7 Click the *OK* button in the *Add-Ins* window. You return to the Excel sheet. Notice the *EXCELCALC* tab in the ribbon menu has disappeared.

Bibliography

[1] Peter Sestoft. *Spreadsheet Implementation Technology. Basics and Extensions.* 2014. ISBN 978-0-262-52664-7.

[2] Simon L. Peyton Jones, Alan F. Blackwell, and Margaret M. Burnett. A user-centred approach to functions in Excel. *SIGPLAN Notices*, 38(9):165–176, 2003.

[3] Excel-DNA. Excel-dna home page. `http://excel-dna.net/`. [Online; accessed 02-February-2014].

[4] Peter Sestoft and Jens Zeilund. Sheet-defined functions: implementation and initial evaluation version 1.1 of 2013-01-16. 2013. IT-University of Copenhagen.

[5] Microsoft. Interop Marshaling Overview. `http://msdn.microsoft.com/en-us/library/eaw10et3(v=vs.90).aspx`, . [Online; accessed 02-February-2014].

[6] Stephen Bullen Rob Bovey, Dennis Wallentin and John Green. *Professional Excel Development*. Addison-Wesley Professional, 2009. ISBN 978-0-321-50879-9.

[7] Scott Driza. *Word 2007 Document Automation With VBA And VSTO*. Wordware Publishing, 2007. ISBN 978-1598220476.

[8] Microsoft. How to: Expose code to vba in a visual c# project. `http://msdn.microsoft.com/en-us/library/bb608604.aspx`, . [Online; accessed 12-May-2014].

[9] Microsoft. Correct a #name? error. `http://office.microsoft.com/en-us/excel-help/correct-a-name-error-HA104049851.aspx`, . [Online; accessed 21-May-2014].

[10] Microsoft. Office support. `http://office.microsoft.com/en-gb/excel-help/`, . [Online; accessed 21-May-2014].

[11] Excel-Dna. Excel-dna packing tool. https://exceldna.codeplex.com/wikipage?title=Excel-DNA%20Packing%20Tool. [Online; accessed 23-May-2014].

[12] Excel-DNA. Excel-dna data type marshaling. https://exceldna.codeplex.com/wikipage?title=Reference. [Online; accessed 18-May-2014].

[13] Decision Models. Excel calculation secrets: User-defined functions. http://www.decisionmodels.com/calcsecretsj.htm. [Online; accessed 27-May-2014].

[14] Microsoft. Macrofun.exe file available on online services. http://support.microsoft.com/kb/128185, . [Online; accessed 17-March-2014].

[15] Excel-Dna.

[16] Microsoft. How to: Customize the office fluent ribbon by using an open xml formats file. http://msdn.microsoft.com/en-us/library/office/ff861787(v=office.15).aspx, . [Online; accessed 24-May-2014].