Applying Automatic Program Verification Techniques to Spreadsheets

A Dissertation Presented in Fulfillment of the Requirements for the Degree of Masters of Research

Sarah Heimlich

B.Eng (Hons), Macquarie University, 2017

Department of Computing
Faculty of Science
Macquarie University, NSW 2109, Australia

Submitted July 2019

Declaration

This work has not previously been submitted for a degree or diploma in any university. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made in the thesis itself.

Signed: Sarah M. Heinlich.....

Date: 30/07/2019.....

Dedication

To Anna Marie Denny, beloved Grandmother and the most determined person I've ever known.

Acknowledgements

First and foremost I'd like to thank my family for never failing to support me through my studies. They know better than anyone that this thesis was a battle, and their constant support and encouragement kept me going.

To the entire village who supported me through the duration of this process, thank you. Thank you for holding my hand and keeping me from (literally and figuratively) falling down. In particular, thank you to Alison, Austin, and Kiatin for always having my back.

Last, but certainly not least, thank you to my supervisor Prof. Sloane for all the assistance.

Abstract

Errors in spreadsheets cost the global economy billions of dollars every year. Spreadsheets are a Turing complete functional end-user programming language. As such, it is not surprising that researchers have investigated how spreadsheet errors can be minimised and resolved using traditional software engineering practices. Despite general success in this field of research, spreadsheets exhibit many unique features that can make standard software engineering techniques difficult. In particular, as we will show, spreadsheets are a partially ordered, non-recursive class of programming languages that support native, automatic type conversion. We further spreadsheet research by proposing and creating a spreadsheet static analyser that automatically verifies whether a spreadsheet will execute without errors over a variety of inputs. The system statically analyses a program to locate spreadsheet specific errors then translates the spreadsheet into C so existing trace abstraction refinement verification tools can be used for common verification challenges. Using the tool, we analyze several spreadsheet corpora to determine the tool's efficacy. The tool was able to correctly determine the validity of all spreadsheets tested, find an undetected type system error, and determine lines of C code generated are a likely indicator of spreadsheet quality.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As computers have become more common in society, so has end-user programming. In particular, spreadsheets have emerged as a common tool for computer programming novices and experts alike to create code. In 1996 there were over 30 million users of Microsoft Excel [36], by 2015 that number was as high as 1.2 billion on desktop applications alone [51]. As computer literacy increases across the globe, it is predicted that end-user programming will expand as well [30].

It is estimated that 90% of industry analysts perform calculations using spreadsheets with 95% of American firms using spreadsheets for financial reporting [27]. Despite this, spreadsheets are far from perfect. Numerous studies by organizations including KPMG and Coopers and Lybrand have shown multiple bugs in 94% of spreadsheets examined [42]. As a result, it is unsurprising that spreadsheet errors cost the global economy billions of dollars annually [37].

Current research in end-user programming and spreadsheets is focused on helping end-user programmers find flaws in their algorithms or the root causes of run-time errors. Here, we consider how to find run-time errors before they occur through automatic program verification.

Our system performs basic static analysis before converting the spreadsheet to C using an Abstract Syntax Tree (AST) so more traditional static verification tools can be used. In particular, the system created utilizes Skink, Macquarie University's verification tool, to perform static program verification. Using the system created, we successfully verified and found errors in real-world and generated spreadsheets.

The purpose of this project is to create a system to demonstrate the viability of automatically verifying spreadsheets through static program analysis. It is important to note this means we do not need to be able to verify every spreadsheet, instead we can select a subset of spreadsheet functionality to consider.

1.1 Report Structure

Herein we present background knowledge in Chapter 2 and design considerations in Chapter 3. With this foundation, we present the system created in Chapter 4 and the results of running the spreadsheet on real-world and generated test cases in Chapter 5. Finally, we describe how the project could be extended and our final conclusions in Chapter 6.

Chapter 2

Background

To understand how we can apply automatic program verification to spreadsheets, we must first understand spreadsheets and automatic program verification. In this chapter, we examine the current corpus of research in both of these subject areas. We also examine end-user programming and software engineering since spreadsheets are an example of this research area.

2.1 Spreadsheets

Spreadsheets have a variety of use-cases across many subject domains - from education to simulations and many other diverse fields of study [31, 47]. Perhaps the largest domain of spreadsheets is in business, where they are often used in decision making processes. When errors occur in these spreadsheets, they can cost organizations millions of dollars [37]. As a result, it is unsurprising that spreadsheet research often occurs at the cross section of business and engineering with leading research groups coming from both faculties [7, 24].

In this section, we seek to further understand spreadsheets by examining the literature available. As with the majority of the literature, we do not consider extensions to spreadsheets such as macros or visual basic.

2.1.1 Spreadsheet Terminology

There are a variety of terms that are specific to spreadsheet programming environments, and it is critical this vernacular is understood and how it differs from traditional programming languages.

In a spreadsheet we refer to the smallest executable program as a cell. Cells are arranged in a grid and can be referenced by their column and row. Rows are denoted by base 10 numbers while columns are referenced by base 26 represented by letters [5].

Cells can be defined as either a constant or formula. In their turn, formulas can contain function calls, constants, references to cells, and traditional mathematical operators ($+$, $-$, $/$, x, etc.).

When cells are copy-pasted or moved, the referenced cells are updated based upon the offset of the move [4]. For example moving $=A2+C1$ from cell A1 to B2 would update the formula to $=B3+D2$. However, references can be forced to not adopt the offset by the addition of a dollar sign [5]. This can be applied in multiple ways; $A\$1$ would maintain the row, $\$A1$ would maintain the column, and $\$A\$1$ would always refer to $A1$ even when moved. These copy-paste features are what enable spreadsheets to be Turing-complete, we can consider copy-pasting a cell to be the program's execution.

2.1.2 Spreadsheets as a Programming Language

While not often viewed as a programming language, spreadsheets have been shown to be Turing Complete [25]. In many applications (Microsoft Excel, Google Sheets, etc.), iteration is not allowed. Instead, the iterations occur through sequential rows in the sheet [25].

Spreadsheets can be further classified as a first-order functional [1], end-user [25], data-sequenced [10], programming language. As a data sequenced language,

spreadsheets are executed based on the flow of information instead of the order of statements. To create looping functions, we can exploit the copy-paste features of spreadsheets to manually create dynamic programs [25, 47].

Given the popularity of spreadsheets, it is not surprising that ensuring spreadsheets are correct and accurate has become a major area of research. Research groups from around the world are working on ensuring spreadsheet are correct from a variety of angles — considering code smells [28, 17], testing [39], validation [44, 9] and more.

2.1.3 Spreadsheet Smells

Just as with traditional programming, spreadsheets are prone to poor implementation that can impact their usability and quality [47]. Many patterns of poor implementation have been documented and are commonly referred to as "code smells" [53]. For example, smells in spreadsheats can occur in formulas and referencing other data-sets [27, 25]. As we will discuss more in further chapters, conditional complexity and long calculation chains have both been found to be spreadsheet formula smells [25].

Conditional complexity occurs when embedded IF statements occur in the same formula. Because it can be difficult to track the different branches in these formulas, the usability and spreadsheet quality may suffer. By examining an example, it quickly becomes clear how the branching in embedded IF statements is difficult to track. In particular, if a cell is defined by the formula $= IF(A1; B1; IF(A2; IF(A3; B3; C3); C2))$ it is hard to follow the different paths and therefore the results.

Long calculation chains occur when formulas reference other cells whose formulas depend on additional cell references [25]. For example, the spreadsheet in Fig. 2.1 computes the Fibonacci sequence. Here, cell $G1$ depends on $A1$ and $B1$ through $C1$, $D1$, $E1$, and $F1$. These long chains make it difficult to determine how a change will propagate through the system.

Table 2.1: Number of formulas resulting in errors in the Enron dataset as discovered by Hermans and Murphy-Hill [26].

Error Type	Formulas
#N/A	948,194
#NAME?	339,365
#REF!	183,014
#VALUE!	111,024
#DIV/0!	76,656
#NUM!	4,087

	A	B	C	D	E	F	G
1	1	1	SUM(A1:B1)	SUM(B1:C1)	SUM(C1:D1)	SUM(D1:E1)	SUM(E1:F1)

Figure 2.1: Spreadsheet showing the long calculation chain smell by calculating the Fibonacci sequence.

2.1.4 Understanding Spreadsheet Errors

To better understand how spreadsheets are used in the real world, we must have a corpus to consider. In 2005, the EUSES corpus was compiled and released with 4,498 spreadsheets making it the largest set of spreadsheets available at the time [19]. Since then, the Enron bankruptcy and resulting law suits have provided a larger corpus with over 15,700 spreadsheets attached to emails that were released [26]. While both of these corpora provide a glimpse into how spreadsheets are used, the Enron corpus is generally considered to be more realistic as it is comprised of spreadsheets used in the day-to-day operations of Enron. As a result, we will focus our considerations here to the Enron data set.

Errors in the Enron Corpus

An analysis of the Enron corpus found out of 20,277,835 formulas, there were 1,662,340 errors [26], meaning over 8% of formulas result in an error. The study also found that 14% of spreadsheets contained at least one error [26]. The frequency of these errors is outlined in Table 2.1.

The most common error was $\#N/A$ which occurs when a look up function finds no matches. In a similar manner the second most common error, $\#NAME?$, occurs when the name of a function is not recognized. Both of these errors often occur through typos such as misspelling. Together, $\#N/A$ and $\#NAME?$ account for over 77% of errors.

The $\#REF!$ error occurs when a reference is invalid. For example if a column is deleted and cells formerly in said column are referenced, a $\#REF!$ would be thrown. The $\#VALUE!$ error is the most ambiguous and can occur for multiple reasons; it is in many ways the catch-all error. Of particular note, the $\#VALUE!$ error can occur if text is referenced when a number is expected. Finally the $\#DIV/0!$ error which obviously refers to division by zero and the $\#NUM!$ error which occurs when the calculation cannot be completed, for example taking a square root of a negative number. It is interesting to note that the $\#DIV/0!$ error occurs nearly 19 times as frequently as other errors that render a formula uncalculatable as denoted by the $\#NUM!$ error.

Formulas in the Enron Corpus

The Enron corpus also found that despite the wide variety of functions available in spreadsheets, not many are used in practice. In particular, 62.8% of the spreadsheets use only the 8 most common functions: SUM, $+$, $-$, $/$, $*$, IF, NOW, and $AVERAGE$ [26]. The spreadsheets were also found to be of low quality, with 49.5% having at least one smell. Of particular interest to this project, 22.3% of the spreadsheets had long calculation chains and 5.5% had conditional complexity [26].

Spreadsheet Error Classification

Spreadsheet errors can be classified as shown in Fig. 2.2 [43]. Throughout the literature, there is a focus on errors caused by the user whether it be from misunderstanding the problem or inputting the wrong data [10, 40].

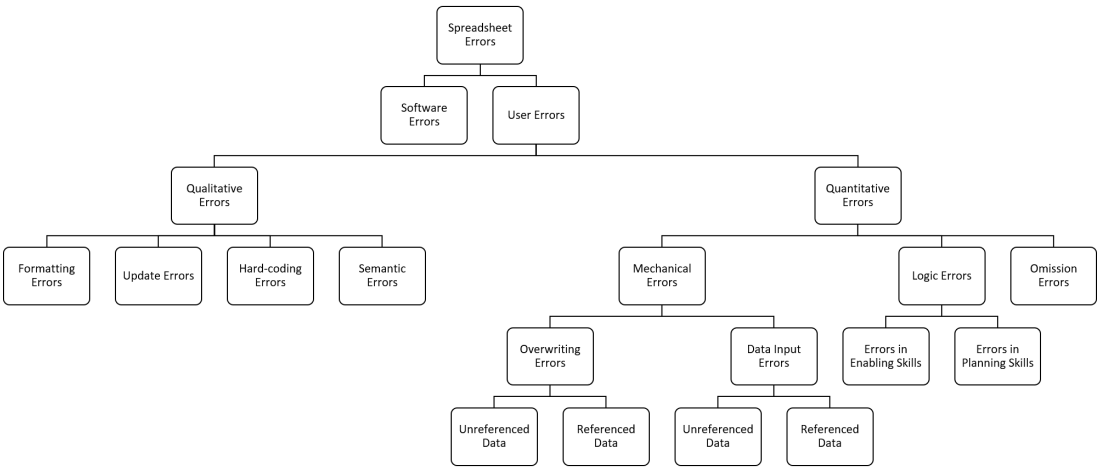

Figure 2.2: Possible errors as outlined by Rajalingha et. al. [43]

According to this school of thought, errors can all be classified as either software errors or user errors. Given a researcher's inability to fix errors in spreadsheet software, our focus is on user errors which can be further classified as either qualitative or quantitative. Qualitative errors are those that can result in an error message as discussed in section 2.1.4.

Given the goal of this project, we will be focusing on qualitative errors which are those caused by formatting, hard-coding, update, and semantic errors. While at first this may seem like a wide range of errors, it is important to recognize the common underlying cause of these issues.

Qualitative errors are caused when the program does not execute as expected due to an issue with the actual code. To put this in terms of the project, qualitative errors are addressable by automatic program verification and therefore will be the focus of this project. This differs from the majority of the research which focuses on quantitative errors [10, 40].

2.1.5 Parsing Spreadsheets

There are multiple grammars available for parsing spreadsheets [20, 5, 1]. The grammars vary in scope and ability to parse more complex formulas and spreadsheets. For example, Abraham and Erwig's grammar was designed to help determine types in spreadsheets [1].

The most in-depth comprehensive context free grammar available was produced in 2015 by Aivaloglou et. al as part of the Spreadsheet Lab at Delft University of Technology [4]. Since then, the grammar has been made open source and further improved [50]. As of 2017, the grammar was able to successfully parse 99.99% of formulas based on a corpus of 8 million unique formulas extracted from four datasets including the Enron corpus [5]. The full grammar can be viewed in Appendix A.

2.2 End-User Programming and Software Engineering

In 2012, there were less than 3 million professional programmers in the USA but over 55 million end-user programmers [32, 45]. As a result, it is understandable that end-user programming has become a large research topic in recent years [32, 10, 12, 18]. At its core, end-user programming quite simply occurs when the programmer is the user. Spreadsheets are an obvious case of end-user programming, and thus we consider this field of resarch in this section.

2.2.1 Understanding the User

End-User Programming is classified as situations where users create code for personal use [32]. It is important to note that personal use does not mean the code cannot be used in a professional setting. Instead, personal use refers to the fact that the user is both the creator and user of the system. For example, a professor deciding

grades with a spreadsheet is a personal use-case.

Because end-user software engineering is based on personal use-cases, the programmer is focused, more often than not, on functionality. As a direct result, the processes and systems that are at the forefront of most software engineering projects are an afterthought in end-user software engineering [32]. This is further compounded as most end-user programmers lack the formal training typical of professional programmers. This causes them to misjudge the need for formal testing and to then become over confident in their system [41, 35].

End-user programmers may not see themselves as programmers and therefore might be hesitant to see what they do as coding [25]. Instead, they see their task as something from their area of expertise (accounting, statistics, etc.) [11]. Put together, these factors make end-user programmers resistant to using tools, systems, and processes that are available. In some cases, the end-user programmer may not consider these options at all [11].

As a result, it is critical to provide easy to use systems that prompt the end-user programmer [11, 10].

2.2.2 Current Research Projects

The majority of current research projects on end-user programming and spreadsheets can fall into two major categories: testing to find errors and static analysis. For example, GoalDebug assists users in finding errors and then automatically suggests ways to correct the bugs [2]. Additional research projects in this area consider the ability to use static analysis to find code smells including faulty empty-cells [33, 52] and convert spreadsheets to use parallel processing [8]. Here we consider a project from both the testing and analysis bodies of research.

From the above sections, we know that it is critical that any systems designed to help end-user programmers be simple and easy to use. This led to the creation of the What You See Is What You Test (WYSIWYT) methodology [10]. From a survey

of the literature, it becomes apparent that WYSIWYT is one of the most successful spreadsheet testing research projects to date [25].

In the first implementation of this system, users could add assert statements in a visual method. This was shown to increase both the number of errors discovered and corrected in a laboratory setting [10]. This model was extended to automatically generate test cases and allow users to say if the results were correct or not [20]. However, this adds additional overhead for the end-user which goes against end-user software engineering principles [25]. WYSIWYT is built on the Surprise-Entice-Reward cognitive model [35]. This model drives users towards testing through curiosity by surprising the user with new information, enticing them through curiosity, and then rewarding by a better performing system [11].

As shown in section 2.1.4, we know that IF statements are one of the most commonly used spreadsheet functions. As a result, it is hardly surprising entire research projects focus on them. In particular, Zhang considers how IF statements can be simplified by eliminating impossible branches [54]. By using this approach, 98% of respondents said the automatic simplifications helped their understanding of the formula.

2.3 Automatic Program Verification

It is very easy to confuse program verification with validation. Program validation is a process across the system's lifetime, from requirements elicitation through to maintenance, that ensures the software does what the customer expects [3]. On the other hand, program verification tools seek to prove the program will behave for all possible inputs [48]. Program verification can examine behaviours ranging from program termination to divide-by-zero errors. Put simply, validation ensures the customer is satisfied while verification proves properties about the program.

For the purposes of this research project, we limit our literature examination

to static analysis for automatic program verification due to the nature of Skink, the automatic program verification tool we will be using. This limitation leaves plenty of research to consider as static analysis tools have been used since 1979 for applications ranging from homeland security to space flight [48].

2.3.1 Predicate Logic

The underlying basis of many automatic program verification techniques is predicate logic. Often, we use Hoare Triples of the form $(|\phi|)P(|\psi|)$ where ϕ is the precondition, P is the program, and ψ is the post condition. The goal of program verification is to prove ψ is implied by ϕ and P [29].

We prove ψ by statically analyzing a program which has been shown to be an NP Hard problem for sufficiently complex programs [34]. Static analysis can be deconstructed into many components such as Abstract Interpretation and Data Flow Analysis [34]. However, the lines between these components have become blurred over time [48].

2.3.2 Abstract Interpretation

In abstract interpretation, certain qualities of the program are abstracted and analyzed to prove the desired outcome [16]. A classic example of this is determining if the result of an equation will be positive or negative [15]. In this case, we are attempting to prove if the result will be positive or negative by abstracting the sign. For example, if we have the formula $positive * positive * negative$, we know from elementary mathematics that the result will be negative. In contrast, if we have $positive + positive + negative$ we know the answer could be positive or negative depending on the magnitude of the values. This abstraction of signs can be seen in Table 2.2.

The critical part of abstract interpretation is to automatically determine the correct abstraction [23]. Abstractions must be limited to the proper domain. Simply

Table 2.2: Table demonstrating the abstraction of signs over multiplication and addition.

Abstraction	Result
$pos \times pos$	pos
$pos \times neg$	neg
$neg \times pos$	neg
$neg \times neg$	pos
$pos + pos$	pos
$pos + neg$	?
$neg + pos$	?
$neg + neg$	neg

assuming an abstraction that works in one situation will work in another, in this case that the abstraction of sign over multiplication is the same as addition, would be incorrect. Abstract relationships such as these can be expressed as lattices to enable automatic program verification [14].

There is an intrinsic relationship between Abstract Interpretation and other automatic program verification techniques including Data Flow Analysis [46]. In data flow analysis, a program's execution order is abstracted into the form of a data-flow graph. This graph can then be annotated and used to verify attributes of the program [6].

2.3.3 Automata and Trace Abstraction Refinement

Another technique used in automatic program verification is based upon automata [23]. Using various forms of automata, we can check different properties of a program. For example, alternating automata can be used to verify concurrent programs and Floyd-Hoare automata to verify a correctness property. This technique works by creating automata where the accepting state should not occur [22].

We can create an automata where each node represents a line of code, and the transitions between nodes are the potential paths through the program. Each transition represents a different line of code. We can further define any assert statements that fail go to an accepting state of the automata. With this setup, by

proving the automata never ends in an accepting state, we have verified the program. More formally, we call the accepting paths through the automata abstract error traces. If we show there is a feasible abstract error trace, the program is incorrect.

Listing 2.1: Example assert to avoid divide by zero error.

```
1 assume(b > −5);
2 int myFunction(a, b) {
3         b = b + 5;
4         b = b / 2;
5         assert(b != 0);
6         return a / b;
7 }
```

Applying this to myFunction in Listing 2.1, we obtain the automata shown in Fig. 2.3. Because we know the value of b when entering myFunction cannot equal −5, it is possible to prove the accepting state never occurs. Since we cannot enter the accepting state, we know the program is correct.

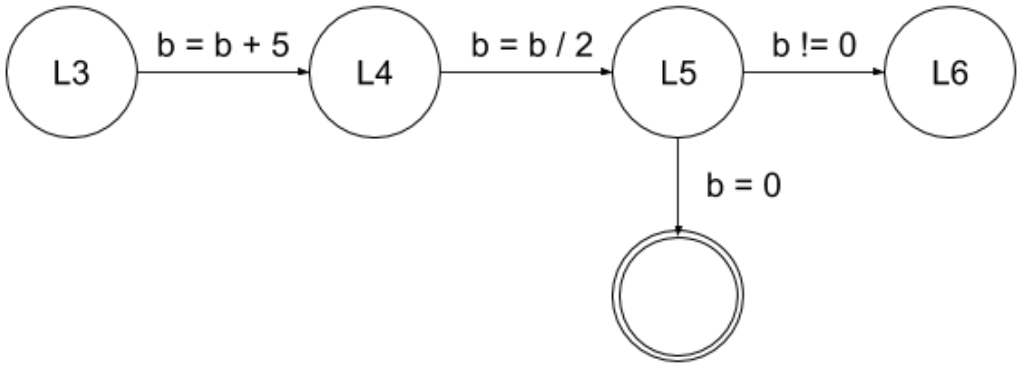

Figure 2.3: Automata which proves no divide by zero errors occur in the sample program.

By reducing the automata to simpler forms over multiple iterations, we refine the potential abstract error traces [22]. And thus, we have abstract error refinement.

2.3.4 Skink

For this project, we will be utilizing Skink, the software verification tool developed by Macquarie University [13]. Skink utilizes refinement of trace abstraction using automata as outlined above. As part of this process, Skink creates a control-flow graph and determines which paths the program can take. If a viable path to an error exists, Skink is able to return an abstract error trace, referred to as a "witness", demonstrating how the program could fail.

To tell Skink what to verify, we use assert statements. In Skink, these are of the form $\_\_VERIFIER\_error()$. Skink will return "FALSE" if one of these asserts can be called, otherwise Skink will return "TRUE." For FALSE results, Skink also provides an example of how the error could occur which we call a witness. An example witness can be found in Appendix B.

To use this in practice, we use if statements to determine whether the $\_\_VERIFIER\_error()$ should occur. This allows the system to capitalize on the ability for static analysis versifiers to determine if certain paths of execution are possible. For example, consider a divide by zero error. In this case, we need to ensure the denominator cannot be equal to zero, so we put the $\_\_VERIFIER\_error()$ inside an if statement that will trigger if the denominator is zero. This is shown in Listing 2.2. Putting this in terms of the Hoare Triples discussed in Section 2.3, the condition guarding the if statement ($A2 == 0$) is the post condition (ψ). The precondition (ϕ) and program (P) are defined prior to the code shown in Listing 2.2.

Listing 2.2: Example of using Skink to verify no divide by zero error is possible.

```
1 if(A2 == 0) {
2     __VERIFIER_error();
3 }
4 int B1 = A1/A2;
```

Chapter 3

Design Considerations

When designing and creating the system for this project, the first step was to decide which platform it should be based upon. Despite the prevalence of Microsoft Office, we chose to examine Open Office Calc. This choice was made due to the open source nature of the platform. This gave additional insight into the errors possible in a spreadsheet, how they are caused, and therefore how we can verify they cannot occur in a given spreadsheet. Unlike Excel which does not provide a complete list of error codes or their underlying causes, Open Office Calc has a fully published list of the possible errors and their causes [38]. Additionally, the .ods file format used is defined by the non-profit organization OASIS, a consortium of over 5,000 participants that creates standards for a multitude of computing platforms.

This decision lets us have a wider view of the problem. For example, as discussed in Section 2.1.4, other researchers have considered errors and the rate at which they occur in spreadsheets. However, the research only considered six error codes as these were the codes viewable in Excel. In comparison, Open Office Calc has 27 unique error codes, giving further insight into what is causing each error.

The choice to use Open Office Calc gives no loss in generality of our findings to other spreadsheets, like Excel. Instead, it provides greater transparency into the underlying system.

3.1 Narrowing Scope

As discussed in Section 2.1.4, by implementing eight of the built in spreadsheet functions (SUM, $+$, $-$, $/$, $*$, IF, NOW, and $AVERAGE$), we should be able to process over 60% of spreadsheets. Given the goal of this project is to demonstrate automatic program verification is viable, being able to work with 60% of spreadsheets is more than enough to accomplish our goal.

By considering these eight functions, it becomes apparent they represent a good cross-section of spreadsheet methods. It includes traditional math functions ($+$, $-$, $/$, and $*$), a function without an argument (NOW), a straightforward function with arguments (SUM), a function that could throw a divide-by-zero error ($AVERAGE$), and most importantly the branching function (IF).

It is critical that IF is included as a key component of any programming language is algorithms. As discussed in Section 2.1.2, we know that spreadsheets expressly forbid looping and self-referencing. As a result, IF statements are the main algorithmic device in spreadsheets. While IF statements have already been included as part of the 8 most common functions, it is important to note that IF statements need to be included in the scope to allow realistic algorithms.

With the scope narrowed to these eight functions, we consider which error types we want to verify will not occur. With / being the fourth most common function used in spreadsheets, the first error we chose to consider was $\#DIV/0!$. Given the complexity of the spreadsheet type system, which we will discuss in more detail in Section 3.2.1, the second error we verify is type errors which are a kind of $\#VALUE!$ error. While these two errors are not the most common, as shown in Table 2.1, it is critical to remember the goal of this project is to demonstrate the viability of applying automatic verification techniques to spreadsheets. These errors were selected to have one error that overlaps with traditional program verification challenges ($\#DIV/0!$) and one error that presents a unique challenge ($\#VALUE!$).

3.2 The Problem Space

As discussed in Section 2.1.2, spreadsheets are a Turing-complete programming language. However, because they are designed for end-user programming, they have many unique attributes, especially when compared to traditional programming languages. Here we consider some of these properties and how they will impact verification.

3.2.1 Type System

Spreadsheets have a weakly typed system that implicitly converts values where possible in a manner that mimics polymorphism as we demonstrate in this section. In particular, the type system of spreadsheets has three main types: string, number, and boolean. Each of these types behaves as a computer scientist would expect, a string contains a sequence of ASCII characters, a number can be an integer or float, and a boolean represents true/false. However, this only scratches the surface of the intricacies of each data type and does not account for null values.

It should be noted that while the data appears as many different types to the user, underneath most types are converted to a string, number, or boolean. For example, consider the NOW function. To the user, this appears to return a date and time. However, the date and time are stored as a floating-point number. As a result, for our purposes we can consider dates to be numbers, thus removing a layer of complexity and simplifying the process.

While booleans appears to the user as TRUE or FALSE, it quickly becomes obvious this is not true as we begin using them in formulas. For example, consider the formulas in Fig. 3.1. Here, we can see that booleans are actually stored as either 1 or 0, both through adding TRUE and FALSE as is done in $B1$ or concatenating the values as in $B2$. Thus, we can say that a boolean is a number. This makes our task easier as bool in C is usually represented as a 1 or 0.

	A	B	C
1	TRUE	1	A1+A2
2	FALSE	10	CONCATENATE(A1;A2)

Figure 3.1: Example of how booleans are actually 0 and 1. The formulas for the B column are shown in the C column.

We can take this argument further by considering IF statements. We know IF statements in spreadsheets are of the form $IF(BOOL; FOO; BAR)$ where if the $BOOL$ is true, we return FOO otherwise we return BAR. However, because booleans are simply numbers, it stands to reason a $BOOL$ can be any number. In fact, this is the case. The behavior is the same as for C; FOO will execute anytime $BOOL$ does not equal 0, and BAR will execute in all other cases.

Having considered booleans in great detail, next we consider numbers. As already stated, for our purposes we consider numbers to be both integers and floating point numbers. In Fig. 3.1 cell $B2$ shows that booleans, which are numbers, can be concatenated. As a result, we know that both booleans and numbers can be used as strings. This differs to C, which as a strongly typed language does not allow for substitution between most data types. In particular, numbers cannot be used in place of a string.

The remaining data type we consider is strings. Unlike booleans and numbers, strings can only be used as strings. When a number is required, a string cannot replace it. This intrinsically makes sense because while we know it is safe to replace the number 1 with the string "1" as needed, there is no logical numerical substitute for most strings.

Combining our knowledge of data types in spreadsheets, we can consider booleans to be a subclass of numbers, dates to be numbers, and numbers to be a subclass of strings. As a result, just as polymorphism allows for multiple classes to be interchangeable, numbers, dates, and booleans can be used as strings. The

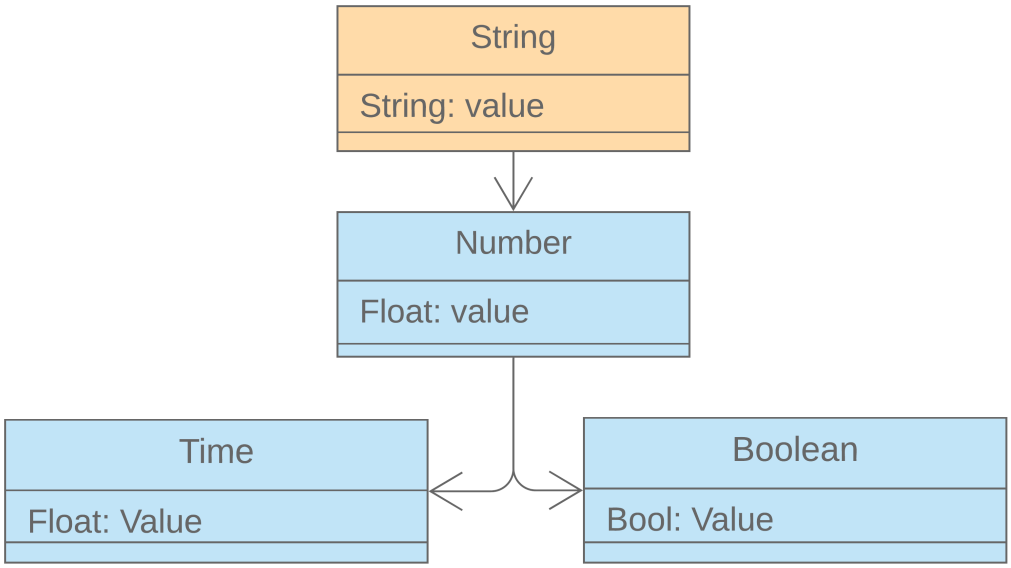

Figure 3.2: The hierarchy and polymorphic property of the spreadsheet type system, values stored as numbers are shown in blue.

type system's hierarchy is shown in Fig. 3.2 to better illustrate this polymophic relationship. In total, we can say that spreadsheets have a weak type system that applies native, automatic type conversion.

3.2.2 Null Cells

In most programming languages, using a variable with a null value causes errors. While this can happen in some cases in spreadsheets, it is the exception rather than the rule. In most cases the system will evaluate an empty cell to zero if it is expecting a number or an empty string if it is expecting a sting. This enables the spreadsheet to operate in a deterministic fashion without throwing errors in the majority of cases. However, because the spreadsheet continues to not throw errors,

it may lull users into a false sense of security. This has caused empty cells to be the target of other research projects as discussed in Section 2.2.2.

Unlike other projects, our concern with null cells is whether they can throw an error or not. Here, we consider the errors they are likely to cause to better inform our system design.

The obvious error a null cell could contribute is $\#DIV/0!$ as the numerical evaluation of a null cell is zero. In addition, other errors can occur because spreadsheets will ignore null cells in $COUNT$ functions. This goes beyond simply $COUNT$ and includes $AVERAGE$ which is computed as $SUM(cells)/COUNT(cells)$. As a result, if $AVERAGE$ is called on a set of empty cells, it will cause a divide by zero error.

3.2.3 Arguments

Because we are converting the functional spreadsheet language into procedural C, there are many interesting problems to solve. Notably, arguments passed into functions can be other function calls. For example, $=SUM(A2;SUM(A3;A4))$ is a valid spreadsheet cell. This case is fairly trivial as the two SUM statements can be evaluated recursively through an AST to give
$=SUM(A2;A3+A4)$ on the first iteration and $=A2+A3+A4$ on the second iteration. While this first case can be automatically handled through an AST, arguments can be much more difficult as show in Formula 3.1.

$$B1 = SUM(B2:B4;IF(C1;C2;C3)) \tag{3.1}$$

In this case, we have two interesting elements to consider. The first is the array $B2:B4$. Until now, we have only considered cases where every cell being referenced is explicitly stated in the formula. Once an array of length greater than 2 is considered, we say the cell is being implicitly referenced. This is because the only cells stated are $B2$ and $B4$, but the array means this formula also references $B3$. As

a result, we must account for implicit referencing in our system. Arrays can contain empty cells which we must correctly account for as discussed in section 3.2.2.

More importantly, Formula 3.1 gives us a glimpse into IF statement usage with $IF(C1;C2;C3)$ used as an argument. This generates many additional problems as will be discussed in section 3.2.5.

3.2.4 Statement Ordering

Because spreadsheets are a data-sequenced language, the order of the statements is dependent on the order in which each cell is referenced. Cells that reference other cells must be processed later. In particular, if we have the cell $A1$ defined by the formula $=A2$ then we know that $A1$ must be defined after $A2$. We express this as $A1 < A2$. It is important to note that there does not need to be a unique order in which cells must be processed. For example, if a spreadsheet does not contain any references, any ordering of the cells is considered legitimate. As a result, it is apparent that the ordering of cells exhibits the required characteristics to be a partially ordered set.

Knowing the order of statement processing is a partially ordered set lets us prove that self-referencing is impossible. To prove this, let us consider a case where cell $A1$ is defined as $=A2$ and cell $A2$ is defined as $=A1$, then according to our partial order we would have $A1 < A2$ and $A2 < A1$. Because both of these statements cannot be true, we know recursion is not allowed in spreadsheets. This example can easily be generalised and proven to be true for any cells that self-reference. Additional examples of referencing can be seen in Fig. 3.3 which has two examples of self-referencing, both of which cause errors as they make the order of statement execution impossible.

	A	B	C
1	A2	FALSE	
2	A1	IF(B1;B3;C3)	
3		B2	0

Figure 3.3: Static Functional Data Sequencing implies recursion is forbidden as shown by the cell pairs A1/A2 and B2/B3.

3.2.5 If Statements

Because spreadsheets do not allow any looping or self-referencing, IF statements are one of the major algorithmic devices available. However, their implementation causes many unique verification challenges.

The majority of spreadsheet functions have a set return type: SUM will return a number and $CONCAT$ will return a string. However, IF statements are an exception to this rule, as they do not have a specific return type. To make this even more challenging, each branch of the IF statement can have a different return type. For example $= IF(A1; 27; "RUSH")$ would return a number if $A1$ is true and a string if $A1$ is false. This means when an IF statement occurs, we must change not only the result, but also the type of the result.

The simple answer to process IF statements is allow an AST to handle them in a recursive method as discussed with SUM in section 3.2.3. This could be achieved through conditional expressions. For example, if we consider formula 3.1 again, we could convert this to C code via a conditional statements as shown in Listing 3.1. However, as discussed in section 3.2.1, IF statements can return a boolean, number, or string which we must account for in the verification process. In our example here, that means we must track the result of the IF and then ensure the result can be used as a number. As a result, the system cannot simply use conditional statements to implement IF statements, instead a more vigorous approach must be applied as

will be discussed in Section 4.2.

Listing 3.1: Example of converting an embedded *IF* to a conditional that does not allow for type checking.

```
1 int B1 = B2 + B3 + B4 + (C1 == 0 ? C2 : C3);
```

3.3 Converting to Verifiable C Code

The main hurdle this project must address is how to translate spreadsheet properties we wish to verify into problems that can be solved by static analysis verification tools in general and Skink in particular. Static analysis has its limitations, but it is particularly good at determining if certain paths of execution are possible. As a result, we must translate the spreadsheet properties we wish to verify into branching problems which Skink can solve. In this section, we consider how this can be accomplished.

3.3.1 Abstract Syntax Tree Representation

To convert the spreadsheet to C, we build an AST, manipulate the tree, and then convert it to C. As part of this process, we must ensure that all the cells referenced in a formula, including those implicitly referenced through arrays, are in the final C code. To accomplish this, either all the references must be included in the AST or we must account for implicit arguments when converting the AST to C.

The simplest solution is to have all the implicit references converted to explicit references in the AST. This allows the system to directly convert each node in the AST to C code. For example, consider the trees shown in Fig. 3.4. Here, the tree on the left has the arguments as an array and a reference while the tree on the right has all the arguments explicitly referenced. The right hand tree is simpler to implement and convert to C as the *SUM* function only has to handle a single type, namely references.

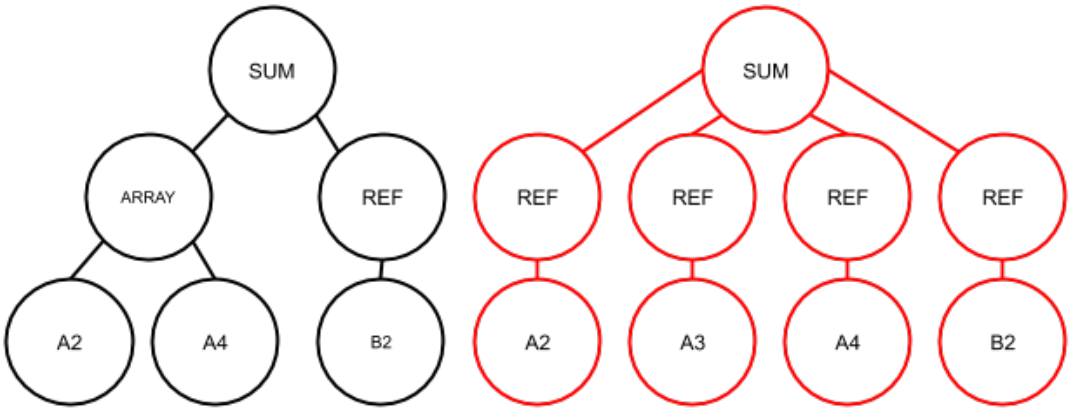

Figure 3.4: Example of converting arrays to references to avoid implicit references. Both trees represent $SUM(A2:A4,B2)$, the tree on the left (in black) maintains the array, while the tree on the right (in red) translates the array to references.

3.3.2 C Code Compilation

It is of upmost importance that all the C programs generated must compile. While this sounds obvious and trivial, because of the complex spreadsheet type system, having programs compile is not a given. For example, consider the spreadsheet shown in Fig. 3.5 which attempts to add together a number (971) and a string (Spartans). The simple and intuitive way of converting this to C is shown in Listing 3.2. However, this code would not compile as you cannot add an integer and char array in C.

	A	B
1	971	
2	Spartans	A1+A2

Figure 3.5: Example of how the type system can cause even simple spreadsheets to not compile.

Listing 3.2: Example of how the spreadsheet type system can cause a simple translation to C to not compile.

```
1 int A1 = 971;
2 char A2[] = "Spartans";
3 int B2 = A1 + A2;
```

The case shown in Fig. 3.5 is simple enough that cell $B2$ would throw an error in the spreadhseet itself, thereby alerting the user to the problem. However, other cases are not as straight forward. For example, let us consider a case that involves an IF statement as shown in Fig. 3.6. This IF statement can mask the error. In this case, no error code would be thrown by the spreadsheet as cell $A2$ would evaluate to a number (971). However, to generate C code, it is not obvious if $A2$ should be defined as a number or a string.

	A	B
1	971	TRUE
2	IF(B1; 971; "Spartans")	A1+A2

Figure 3.6: Example of how more complex spreadsheets can hide compile errors.

The system must be able to handle these cases and ensure that all code generated compiles.

3.3.3 Asserts

While verifying a $\#DIV/0!$ is simple to verify, other cases are more difficult. In particular, verifying the types are correct presents many unique challenges because C and Skink have a strictly typed system, unlike the weak polymorphic typing in spreadsheets. As a result, our system must track the type of a variable and verify only correct data types are used.

3.3.4 Defining the inputs

We know the goal of verification is to show a post condition (ψ) will hold for program (P) for the precondition (ϕ). More informally, we say that verification proves certain properties hold over a range of inputs. To accomplish this, the code requires a means to tell the verification tool that a certain value could vary. For example, in Skink we can denote a non-deterministic integer by $\_\_VERIFIER\_nondet\_int()$.

The challenge with spreadsheets is to know which values should be variable, and which ones should be constant. For example, consider the spreadsheet in Fig. 3.7 which has cell $A2$ defined by the formula $=A1/B1$. If we consider the $A1$ and $B1$ cells to be constant, to verify the system does not throw a $\#DIV/0!$ error, we would execute the code in Listing 3.3. This would return TRUE as Skink would recognize $B1$ equals 32 which is not 0, and thus the error cannot be thrown. This creates a situation in which we are not properly verifying the spreadsheet as we will only discover errors that are already present. Instead, we want to show that for a variety of values the spreadsheet will still execute properly.

	A	B
1	31	32
2	A1/B1	

Figure 3.7: Example of divide.

Listing 3.3: Example of having too many constants creating uninteresting verification.

```
1 int A1 = 31;
2 int B1 = 32;
3 if(B1 == 0) {
4     __VERIFIER_error();
5 }
6 int A2 = A1/B1
```

With no additional knowledge about the spreadsheet, it is difficult to know which cells should be verified as variables and which should be constants. With everything as a constant, there will be spreadsheets falsely verified to be correct,

but with everything set as a constant, there will be spreadsheets falsely verified to be incorrect. Given the goal of the project is to demonstrate that automatic program verification is viable, we would prefer too many errors to be detected within the scope of this project. This is addressed in more detail in Chapter .

As a result, we consider all cells that are not formulas or null to be inputs to the system. This allows us to obtain the code shown in Listing 3.4 which would return FALSE. This is because Skink would recognised that when $B1$ is zero, a $\#DIV/0!$ error would occur.

Listing 3.4: Example of inserting variable integers to create interesting verification.

```
1 int A1 = __VERIFIER_nondet_int();
2 int B1 = __VERIFIER_nondet_int();
3 if(B1 == 0) {
4     __VERIFIER_error();
5 }
6 int A2 = A1/B1
```

Unlike cells with an explicit value, we have chosen to assign null cells a value of zero as this is what they evaluate to as discussed in Section 3.2.2. This choice was made as unlike cells with values, these cells do not represent anything, and thus do not have a fluctuating value.

Chapter 4

The System

In the previous chapters we have given the background of research for this project, limited the scope of inquiry, and examined the unique elements of spreadsheets that will impact verification. In this chapter, we bring this knowledge together to describe the system created.

At a high level, the system will verify spreadsheets by compiling them into C code with numerous assert statements to ensure errors cannot occur across a range of inputs. The assert-filled C code is verified by Skink, Macquarie University's software verification tool. To easily integrate with Skink, the system was created in Scala, the same language Skink is written in, so the systems can seamlessly work together.

In total, the system works in four steps: parsing, transformation, code creation, and verification through Skink. Each stage of the process handles some of the challenges discussed in previous sections as outlined in Fig. 4.1. In this chapter we walk through each step in detail and explain its role in the overall system.

4.1 Step One - Parsing

The goal of the parsing step is to transform the spreadsheet from the .ods file created by Open Office Calc into a sequence of properly ordered formulas that we call an expression and from that expression into the AST.

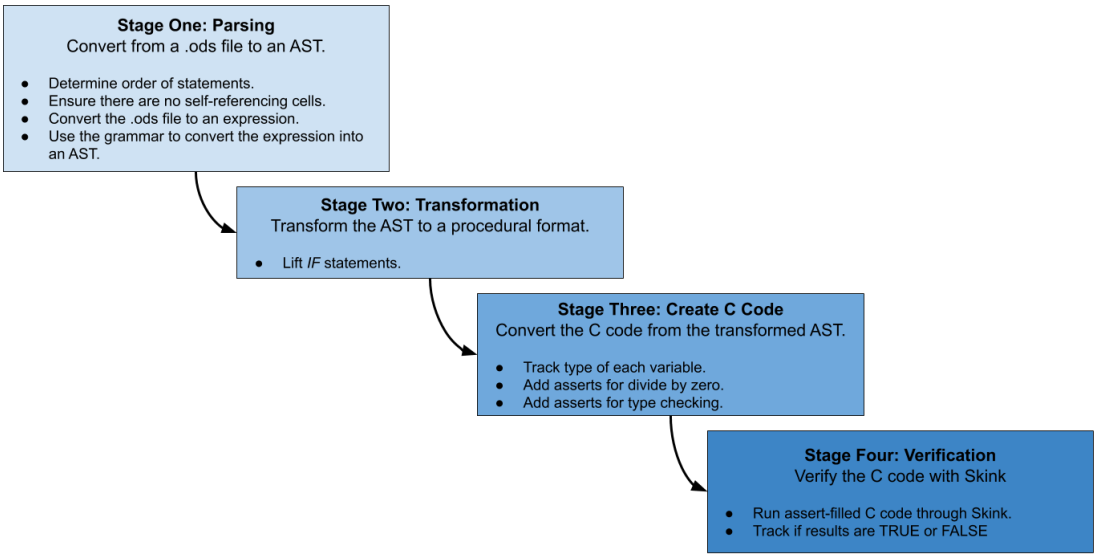

Figure 4.1: The stages of the system and their responsibilities.

4.1.1 Statement Ordering and Excluding Self-Referencing

Two key challenges in creating the system are proper ordering of cells into C statements and removing self-referencing of cells in the process. To order the statements and create a simple data structure to pass to the grammar, we turn the .ods file into what we call an expression. The expression is the definitions of the cells in a proper order separated by commas. For example, the spreadsheet shown in Fig. 3.7 would give the expression $A1 = 31, B1 = 32, A2 = A1/B1$, This step not only determines the order of statements, but also ensures there are no self-referencing cycles, removes arrays as arguments, and creates a simpler format for the grammar to parse. This step also allows us to change between different spreadsheet file formats (.ods, .exls, etc.) by switching out a single module instead of needing to recreate the entire system.

When considering how to order the statements, it quickly becomes apparent that this problem is similar in nature to excluding self-referencing. To solve both problems, we need to determine the order in which the statements should be processed, if a cell references itself either directly or transitively, we will discover

this as there will be no correct ordering of the statements.

We discussed in Section 3.2.4, we know that the order in which cells are processed can be considered a partially ordered set. Consequently, we can create a Hasse diagram of the ordering in the form of a directed graph. By performing a topological sort on the graph, we can determine a valid order to process the cells. This also allows us to discover self-referencing cells as they will create a cycle in the Hasse diagram.

To determine the order in which the cells should be processed, we create a directed graph that models the referencing. For example, let us consider the spreadsheet and resulting directed graph shown in Fig. 4.2. Here, we can see how the referenced cells point to their parent cell, such as in the the A column where $A1$ is defined as $= A2 + A3$, which creates the edges from $A2$ to $A1$ and $A3$ to $A1$. Fig. 4.2 also shows an example of self-referencing through the cycle between $B1$ which is defined as $= B2$ and $B2$ which is defined as $= B1$. To ensure we have a connected graph, we create a null cell that references all constant values. This is shown via cell $C1$. With a connected graph, we can easily perform a topological sort which gives a valid ordering of the statements.

	A	B	C
1	A2+A3	B2	8
2	1	B1	
3	2		

Figure 4.2: A spreadsheet that contains a set of self-referencing cells.

When creating the directed graph, we also include implicitly referenced cells from arrays as discussed in section 3.2.3. In this process we transform the array into a list of arguments to better facilitate creating the AST as we will discuss in more detail below. For example, if we had a cell defined by the formula $= SUM(A2 : B4)$, this would be transformed into $= SUM(A2; A3; A4; B2; B3; B4)$.

4.1.2 The Grammar and Abstract Syntax Tree Creation

Rather than create a grammar from scratch, we based the grammar on the Excel formula parsing grammar presented by Aivaloglou, Hoepelman, and Hermans [4]. While multiple grammars are available for parsing spreadsheets, the Aivaloglou grammar was selected as it successfully parsed 99.99% of the eight million formulas with which it was tested including those in the Enron corpus [4].

For our purposes we have simplified the grammar to only use the eight most common functions as discussed in Section 3.1 along with $=$, $>$, and % as required for some of our test cases as we will discuss in Chapter 5. While our scope of inquiry would enable the usage of other grammars, by using the Aivaloglou grammar we know the system can be easily extended for future work.

The Aivaloglou grammar was created to parse an Excel formula instead of an Open Office Calc spreadsheet. As a result, the grammar had to be slightly modified to allow for multiple Calc formulas to be parsed. To implement the grammar, we use SBT-RATS [49] which utilizes Parsing Expression Grammars (PEGs) to define the language [21]. The exact SBT-RATS parsable grammar used can be viewed in Fig. 4.3.

The majority of the changes made to the Aivaloglou grammar were superficial. For example, Calc uses semi-colons instead of commas to separate arguments passed into a function. Additionally, in the .ods file format used by Calc, cell references are enclosed in square brackets and preceded by a full stop, meaning the reference of $A1$ is $[.A1]$. This means we have to parse $SUM([.A1];[.A2])$ instead of $SUM(A1,A2)$.

In addition to these superficial changes, changes were also made to facilitate multiple formulas to be parsed. The Aivaloglou grammar only allows a single formula to be parsed at a time, but we want to be able to parse the entire sheet. As a result, expressions and statements were added to the grammar. Statements are of the form $cell = value$, and when combined create expressions.

Figure 4.3: The grammar used to parse spreadsheets represented as expressions.

```
Exp =
    Assign Exp
    |Assign
    |AssignIf Exp.

Assign = Cell "=" Formula ",".

AssignIf = ifRef + "=" nIf.

Formula =
        nIf
        |Formula "+" Formula
        |"+" Formula
        |Formula "-" Formula
        |"-" Formula
        |Formula "*" Formula
        |Formula "/" Formula
        |Formula "=" Formula
        |Formula ">" Formula
        |Formula "%"
        |"SUM(" +Args+ ")"
        |"AVERAGE(" +Args+ ")"
        |"NOW()"
        |Array
        |CellRef
        |ifRef
        |NumConstant
        |Str
        |"(" Formula ")"
        |"null".

nIf = "IF(" +Formula+ ";"+
    Formula+ ";" +Formula+ ")".

Args =
     Formula ";"Args
    |Formula.

NumConstant =
              Decimal
             |Number
             |Bool.

Array = "[." Cell ":." Cell "]".

CellRef = "[." Col Row "]".

Col = "$"? [A-Z]+.

Row = "$"?[ 1-9][0-9]*.

ifRef = row+ "if".

Number = [0-9]+.

Decimal = Number "." Number.

Str = "" " middleOfString.

middleOfString =
                 " " "
                |_middleOfString.

Bool =
     "false"
    |"true".
```

The "*null*" formula was added to the grammar to allow empty cells to be parsed and represented in the AST. This is important as null cells have many unique properties we want to verify as discussed sections 3.2.2 and 3.3. We also added the "assignIf" formula to simplify the process of removing IF statements from arguments as we will discuss in more detail in Section 4.2.

4.2 Step Two - Transformation

To create the C code, we need the AST to be in a procedural format. Because formulas can be embedded as arguments, the AST cannot be directly translated into C code. For example, considerwhat would happen if we have an IF inside a SUM. While this is a perfectly valid snippet of spreadsheet code, as discussed in section 3.2.5, we cannot simply embed the IF statement inside a formula in C as we must allow for type verification. As a result, we must transform the AST to a more procedural format.

To transform the AST from a functional to procedural format, we lift the IF out of the SUM. This process can be repeated recursively as needed to handle embedded IF statements. While we have focused our discussion here on lifting IF statements, the same process can be applied to any function that cannot exist as an argument within a procedural language.

4.3 Step Three - Code Creation and Assert Insertion

Once the AST is in a procedural format, creating the C code is a detailed but simple process. We process the AST in a left-recursive order, transforming each node to the appropriate C code.

The important part of the code creation is ensuring that the code will compile and find $\#DIV/0!$ and type $\#VALUE!$ errors as discussed in Section 3.1. To accomplish this, we must properly implement IF statements, add the appropriate assert

statements, and resolve type system compilation errors.

4.3.1 Handling IF Statements

As discussed in Section 4.2, we know IF statements are lifted into their own assign nodes in the AST. However, they are still defined in the functional manner of assigning a variable name to result of the statement. To remedy this, we assign the value inside the if and else statements. In line with good programming practice, we assign the variable a temporary value of zero before the if-else statement to ensure the variable has a value.

To create the if-else statement in C, we must remember that booleans are actually numbers in spreadsheet languages where zero represents false and one represents true as discussed in Section 3.2.1. Taking this a step further, any non-zero value evaluates to true. As a result, instead of having $if(val)$ we convert the IF statement to $if(val != 0)$. While C already evaluates IF statements this way, we explicitly state it to be clear in our intent and simplify potential future work with strings. Putting this together we can transform the statement $IF(B1;B1;B2)$ into the C code shown in Listing 4.1.

Listing 4.1: C Version of $IF(B1;B1;B2)$.

```
1 int if1 = 0;
2 if(B1 != 0) {
3     if1 = B1;
4 } else {
5     if1 = B2;
6 }
```

4.3.2 Type Checking

Once we begin to have multiple types, ensuring the generated code compiles becomes more challenging. For example, if the user tries to add a string and number together, the C code will not compile. As a result, we model everything as a number. This

decision can be justified as all eight functions within our scope are based on numbers. Additionally, for the purposes of verification, the concern is not about what the function outputs, but rather whether the function is capable of causing errors. By modeling everything as an integer, code compilation becomes much simpler.

Based on Section 3.2.1, we know all the data types we expect to encounter are either numbers or strings. As a result, the system simply needs to differentiate between these types. To accomplish this, for every cell we add an additional integer that tracks whether the cell contains a number or a string. For example if $A1 = 3132$, we would have $A1Num = 1$ while if $A2 =$ "TDU" we would have $A2Num = 0$. To verify the correct type is used, we can check the tracking integer has the expected value. For example, if we had cell $A1$ defined by the formula $= A2 + A3$ we could verify both $A2$ and $A3$ are numbers using the code shown in Listing 4.2.

Listing 4.2: Asserts to check $A2$ and $A3$ are numbers.

```
1 if (A2Num == 0){
2     __VERIFIER_error();
3 }
4 if (A3Num == 0){
5     __VERIFIER_error();
6 }
```

While this is simple when a cell simply contains a value, it becomes more complex when formulas are considered. In the simplest case, we can take the return type of the function called, and assume the cell takes this value. For example, where $A1 = A2 + A3$, because we know the + function returns a number, we would assume $A1$ is a number. In cases where a cell references another cell, we assume they have the same type. In particular, we say that if $A1 = A2$ then $A1Num = A2Num$. This enables us to handle seven of our initial eight functions in scope, with the exception being IF statements.

Unsurprisingly, IF statements complicate the situation. As discussed in Section 3.3, IF statements could return any type, and may return a different type depending

on the input. For example, consider Fig. 4.4. In this example, if $C1$ is true, $A2$ will be a number. But if $C1$ is false, $A2$ will be a string. As a result, we use the same methodology as for IF statements. We set the variable tracking the type to 0, and then update it as needed inside the if-else statements. This is shown in Listing 4.4 which coverts Fig. 4.4 into verifiable C code.

	A	B	C
1	3132	TDU	TRUE
2	IF(C1;A1;B1)	A1+A2	

Figure 4.4: Spreadsheet which could throw a error.

4.3.3 Divide by Zero

The simplest assert statements that we consider are those to ensure there are no $\#DIV/0!$ errors in the spreadsheet. As discussed in Section 3.3, we simply assert that the denominator of the function cannot equal zero. For example, if we had $A1 = A2/A3$ our code would become that in Listing 4.3.

Listing 4.3: Verification no divide by zero error occurs.

```
1 if(A3 == 0)
2     __VERIFIER_error();
3 int A1 = A2 / A3;
```

4.3.4 Final Code

With the compilation difficulties and assert statements well understood, the only remaining item is adding some simple code at the start and end to ensure the file compiles. Putting all of this together, we can transform the spreadsheet shown in Fig. 4.4 into the code in Listing 4.4.

Listing 4.4: The fully generated C code for verifying the spreadsheet in Fig. 4.4.

```
1  extern void __VERIFIER_error() __attribute__ (
2              (__noreturn__));
3  unsigned int __VERIFIER_nondet_uint();
4  int main() {
5      int C1Num = 1;
6      int C1 = __VERIFIER_nondet_int();
7      int B1Num = 0;
8      int B1 = __VERIFIER_nondet_int();
9      int A1Num = 1;
10     int A1 = __VERIFIER_nondet_int();
11     int if1Num = 0;
12     // Compute the IF statement in cell A2.
13     int if1 = 0;
14     if(C1!= 0) {
15         if1Num = A1Num;
16         if1 = A1;
17     } else {
18         if1Num = B1Num;
19         if1=B1;
20     }
21     int A2Num = if1Num;
22     int A2 = if1;
23     // Ensure the cells added in B2 are both numbers.
24     if(A1Num == 0)
25           __VERIFIER_error();
26     if(A2Num == 0)
27           __VERIFIER_error();
28     int B2Num = 1;
29     int B2 = A1 + A2;
30 }
```

4.4 Step Four - Verification

The final step in our verification system is handing the assert filled C code to Skink. Skink then analyses the code to determine if the assert statements could occur given all possible values of the variables. Skink then outputs the results, along with a witness if an error could occur. An example witness is shown in Appendix B.

Chapter 5

Results

To determine the success of the code and the project, we ran the code on three sets of input data: a subset of the Enron corpus, a grading sheet, and generated test cases. In this process, we added three simple functions to the grammar, namely equal to, greater than, and percentage, to enable some of these data-sets to be parsed. Each of these data-sets let us examine the system in a different manner.

After running a spreadsheet through the system, a result of TRUE meant the spreadsheet was verified to be correct, while FALSE meant an error had been detected. When Skink returned FALSE, the witness error trace was examined to determine what type of error had been discovered.

5.1 Enron

As previously discussed, the Enron spreadsheet corpus is considered the gold standard for testing spreadsheets. Because it was obtained through subpoenas as part of the bankruptcy process, the spreadsheets give a unique glimpse into how spreadsheets are used in a corporate setting.

To test the system created, we decided to analyze the spreadsheets created by a single user to act as a use case. After analyzing spreadsheets generated by different employees, we found a large percentage of those generated by Barry Tycholiz could be parsed by the system. As a result, we ran all nine spreadsheets he had created that only contained parsable formulas through the system. By manually checking

these spreadsheets, we believed three should return FALSE while six should return TRUE. In running the system on the sheets, all had the expected return type. All of the FALSE errors discovered were caused by division. In particular, the employee was dividing by a variable that could equal zero. A summary of the results can be seen in Table 5.1.

Two spreadsheets generated stack overflow errors on initial testing. The first one when run through the system generated and the second one through Skink. By increasing the size of the stack, we were able to successfully run both programs.

It is believed the stack overflow in Skink was caused due to the large programs generated by the system. In particular, the file that caused the error was 1,135 lines long. This caused us to consider the size of the C files generated. At a minimum, every cell with a value or referenced will generate two lines of C code: one for defining the variable and one for tracking the variable's type. However, if the cell is defined by a formula with references, this will add an additional three lines per referenced cell to perform the type checking. If an IF statement is used, an additional nine lines will be added.

This basic understanding was confirmed by examining the number of lines of code generated compared to the number of cells in each spreadsheet as shown in Table 5.1. As expected, each cell generated at least two lines of C code. On average, 3.52 lines of C code were generated for each cell in the spreadsheet. By graphing the function as shown in Fig. 5.1, we see there is a generally linear trend between the number of spreadsheet cells and lines of C code. However, we can also see that the spreadsheets that are not verified to be correct appear to be above the average trend line. Investigating further, we found spreadsheets that could cause errors have an average of 4.49 lines of C code generated per spreadsheet cell. This is both above the average for all spreadsheets, as well as those that were verified to be correct which had an average of 3 lines of code per cell.

While this finding is not statistically significant, it is in line with other research

Enron Corpus ID	Result	Stack Overflow	Cells	Lines	Lines per Cell
barry_tycholiz_000_1_1.pst.65	TRUE	No	39	123	3.15
barry_tycholiz_000_1_1.pst.67	TRUE	No	45	115	2.56
barry_tycholiz_000_1_1.pst.56	TRUE	No	62	229	3.69
barry_tycholiz_000_1_1.pst.10	TRUE	Yes	140	361	2.58
barry_tycholiz_000_1_1.pst.31	FALSE Divide by zero	No	141	503	3.57
barry_tycholiz_000_1_1.pst.30	TRUE	No	195	675	3.46
barry_tycholiz_000_1_1.pst.13	FALSE Divide by zero	No	216	1301	6.02
barry_tycholiz_000_1_1.pst.75	TRUE	No	237	657	2.77
barry_tycholiz_000_1_1.pst.32	FALSE Divide by zero	Yes, in Skink	292	1135	3.89
Average			**151.89**	**566.56**	**3.52**

Table 5.1: Table outlining the results of running the system on nine sheets from the Enron corpus.

which has shown more complicated spreadsheets are more likely to have bugs and throw errors. We know the additional lines of code are generated by having additional references to cells or IF statements, thereby making lines of generated code per cell a measure related to spreadsheet complexity. In particular, we know a spreadsheet full of smells such as conditional complexity and long chain calculations as discussed in Section 2.1.3 will lead to longer C files due to the increased number of IF statements and references required for these smells to exist.

5.2 Grades

The next spreadsheet considered was one used by a professor to determine grades for students. The spreadsheet mainly consisted of the same line which added up a student's marks to determine their final numerical score, and then convert that score into a letter grade. The line was repeated for every student in the class.

Through manual examination, the spreadsheet appeared to be valid, but upon running it through the system, Skink said an error could occur. By examining the witness, a cell containing the text "0" was found being called by a SUM statement. Because the value was "0", the value was ignored by the SUM function and thus the

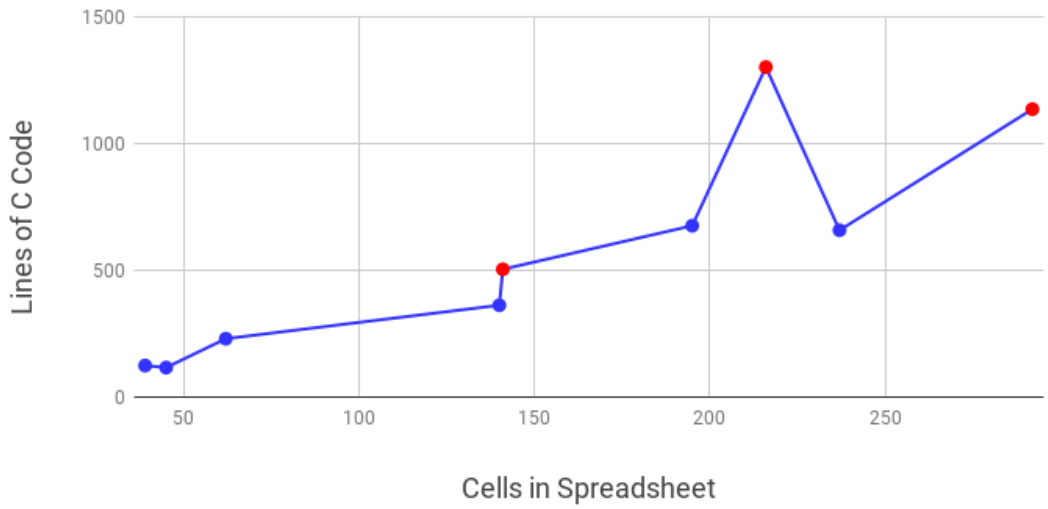

Figure 5.1: Graph showing the number of cells in a spreadsheet compared to the lines of generated C code. Files that were found to have potential errors are denoted in red.

cell still had the correct result. As a result, this error had gone undetected both by the original user and the manual inspection. Had the value been any other number, the error would have produced an incorrect calculation result. After changing the cell from the string "0" to the number zero, the spreadsheet was run through the system again and was verified to be correct.

5.3 Generated Test Cases

In addition to the real world spreadsheets used, we also created some generated files to highlight some key attributes of the system.

After running the system on some Enron sheets, it was concerning that every division was causing the spreadsheet to not be validated. As a result, our first goal was to create a spreadsheet that would not throw a $\#DIV/0!$ error even when dividing by a variable. To accomplish this, we created the spreadsheet shown in Fig. 5.2. This spreadsheet has a divide by a value which cannot equal zero. This occurs because the denominator will be evaluated as $0+1$ since cell $C1$ is null and

therefore we consider it to always evaluate to 0. As a result, despite the appearance of a potential $\#DIV/0!$ error, our verification system shows the spreadsheet is valid.

	A	B	C
1	B1/(1+C1)	1	

Figure 5.2: Spreadsheet with division which was validated using the system.

Additionally, although through the grade spreadsheet we found a type error, we wanted to ensure these errors could be found in more complex situations, such as when the type could change based on an IF statement. To achieve this, we ran the code on the sheet shown in Fig. 4.4. In this situation, Skink found a type error could occur as we expected.

5.4 Summary of Findings

In running the system on sheets from the Enron corpus, a grading example, and a generated test case, we found it worked on all the cases it was presented. While the files were larger than anticipated, by increasing the size of the stack we were able to successfully validate the spreadsheets. The system was able to find type errors that had gone undetected by the original user as well as a manual inspector prior to testing. We also successfully showed that while the system does not validate the majority of spreadsheets that divide by a variable, there are cases for which this will not occur. Perhaps our most interesting finding is that the number of lines of C code generated per spreadsheet cell appears to be an indicator of spreadsheet quality. It is important to note this claim centers around the number of lines of C code generated per spreadsheet cell rather than the number of lines of code generated from the spreadsheet as a whole. This is most likely caused by the numerous spreadsheet smells, namely conditional complexity and chain calculations) that cause additional lines of code to be generated per cell.

Chapter 6

Conclusions and Future Research

6.1 Future Work

The goal of this project was to create a system to demonstrate the viability of automatically verifying spreadsheets through static program analysis. While we have accomplished this goal, it has opened many potential directions for future projects. Here we discuss the next steps in automatic program verification applied to spreadsheets.

Due to the time frame of this project, our scope was limited to the eight most commonly used functions in spreadsheets. An obvious next step is to expand this to allow more spreadsheets to be verified using the system. This will allow for a wider corpus to be verified.

In verifying $AVERAGE$, we have not considered if all the arguments passed to $AVERAGE$ are null. As discussed, this would cause a $\#DIV/0!$ error to occur.

When we generate the Hasse structure to determine the order in which cells are processed, we assume all branches can occur. However, we know from other research that this is not always the case in spreadsheets. By identifying and removing these dead branches from the system, we would stop identifying self-referencing cycles that cannot exist. For example, consider the spreadsheet in Fig. 6.1. The current system would find a self referencing cycle between cells $A1$ and $A2$, however, upon further inspection, this sheet cannot throw an error as the self referencing can never occur. $A1$ will equal $A2$ only if $B1$ is true. However, $A2$ will equal $A1$ if $B1$ is

false. Since $B1$ cannot be both true and false, the self referencing can never occur.

	A	B
1	IF(B1;A2;B2)	TRUE
2	IF(NOT(B1);A1;B2)	1678

Figure 6.1: Example of a potential self-referencing cycle that cannot occur due to IF statements.

As we found when running the system, the majority of the time a user divides by a referenced cell, the system will not validate the spreadsheet. This is because we currently assume cells that are assigned a single value could be any number. Future projects could allow the user to specify cells as constants or a range of values, thereby enabling more spreadsheets to be verified. This can be easily acheived through Skink with the $\_\_VERIFIER\_assume(p)$ function.

Because spreadsheets can quickly have many cells and formulas, the size of the AST initially caused stack overflow errors. Future projects should consider how to minimise the size of the AST to reduce the computing power required to verify spreadsheets.

The current user interface is not intuitive and requires some basic knowledge of technical tools such as command line and the ability to understand the trace abstraction witness. By creating a more intuitive display, we would allow more end-users to access the system.

Once the system is further developed as discussed above, the obvious next step is to test it with real end-users. This would allow us to know if the system helps users find errors before they occur and identify what is going wrong with errors that already exist.

6.2 Conclusion

It is widely known that bugs in spreadsheets cost the global economy billions of dollars. As a result, there is significant interest in improving spreadsheet accuracy and quality. The goal of this project was to create a system to demonstrate the viability of automatically verifying spreadsheets through static program analysis.

In this document, we have outlined current research projects on spreadsheets, end-user programming, and automatic program verification. Based on this knowledge, we limited the scope of our system to the eight most common spreadsheet functions enabling us to process over 60% of spreadsheets.

The background research done allowed us to consider the unique properties of spreadsheets and how this changes the verification process. In particular, it enabled us to notice the partially ordered execution order of cells makes self-referencing cells impossible. It also informed our understanding of the weak polymorphic type system employed in spreadsheets that enables native, automatic type conversion.

With this knowledge, we created a four-step system that parses spreadsheets, transforms the AST from functional to procedural, creates C code with assert insertion, and finally verifies the code with Skink a trace abstraction refinement verification tool.

We were able to successfully use the system to verify spreadsheets from the Enron corpus, a grading spreadsheet, and generated test cases. The system returned the proper results for all the spreadsheets and found a bug in the grading spreadsheet that had gone undetected. Interestingly, lines of C code generated per spreadsheet cell appears to be an indicator of spreadsheet quality.

The goal of this project was to determine if spreadsheets could be verified through static program analysis. Given the spreadsheets verified and errors found, we can definitively say yes. Static program verification can be applied to spreadsheets.

Appendix A

The Aivaloglou Spreadsheet Grammar

The Aivaloglou Grammar as presented in [5]

⟨Start⟩::= ⟨Formula⟩

| ’=’ ⟨Formula⟩

| ‘=’ ⟨Formula⟩‘’

⟨Formula⟩::= ⟨Constant⟩

| ⟨Reference⟩

| ⟨FunctionCall⟩

| ‘(’ ⟨Formula⟩‘)’

| ⟨ConstantArray⟩

| RESERVED-NAME

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

⟨FunctionCall⟩::= EXCEL-FUNCTION⟨Arguments⟩‘)’

| ⟨UnOpPrefix⟩⟨Formula⟩

| ⟨Formula⟩‘%’

| ⟨Formula⟩⟨BinOp⟩⟨Formula⟩

⟨UnOpPrefix⟩ ::= ’+’ | ’-’

⟨BinOp⟩::= ‘+’ | ’-’ | ’*’ | ’/’ | ”̂ | ’&’ | ‘<’ | ‘>’ | ‘=’ | ‘<=’ | ‘>=’ | ‘<>’

⟨Arguments⟩::= ⟨Argument⟩ ’,’

⟨Argument⟩ | ϵ

⟨Argument⟩::= ⟨Formula⟩ | ϵ

⟨Reference⟩::= ⟨ReferenceItem⟩

| ⟨RefFunctionCall⟩

| '(' ⟨Reference⟩')'

| ⟨Prefix⟩⟨ReferenceItem⟩

| FILE '!' DDECALL

⟨RefFunctionCall⟩::= ⟨Union⟩

| ⟨RefFunctionName⟩⟨Arguments⟩')'

| ⟨Reference⟩':' ⟨Reference⟩

| ⟨Reference⟩'␣' ⟨Reference⟩

⟨ReferenceItem⟩::= CELL

| ⟨NamedRange⟩

| VERTICAL-RANGE

| HORIZONTAL-RANGE

| UDF⟨Arguments⟩')'

| ERROR-REF

| ⟨StructuredReference⟩

⟨Prefix⟩::= SHEET

| FILE SHEET

| FILE '!'

| MULTIPLE-SHEETS

| FILE MULTIPLE-SHEETS

| '" SHEET-QUOTED

| '" FILE SHEET-QUOTED

| '" MULTIPLE-SHEETS-QUOTED

| '" FILE MULTIPLE-SHEETS-QUOTED

⟨RefFunctionName⟩::= REF-FUNCTION

| REF-FUNCTION-COND

⟨NamedRange⟩::= NR | NR-COMBINATION

⟨Union⟩::= ‘(’ ⟨Reference⟩‘,’ ⟨Reference⟩ ‘)’

⟨StructuredReference⟩::= ⟨SRElement⟩

| ‘[’ ⟨SRExpression⟩‘]’

| NR⟨SRElement⟩

| NR ‘[’ ‘]’

| NR ‘[’ ⟨SRExpression⟩‘]’

⟨SRExpression⟩::= ⟨SRElement⟩

| ⟨SRElement⟩(‘:’ |‘,’) ⟨SRElement⟩

| ⟨SRElement⟩‘,’ ⟨SRElement⟩(‘:’ | ‘,’) ⟨SRElement⟩

| ⟨SRElement⟩‘,’ ⟨SRElement⟩‘,’

⟨SRElement⟩‘:’ ⟨SRElement⟩

⟨SRElement⟩::= ‘[’ (NR | SR-COLUMN) ‘]’

| FILE

⟨ConstantArray⟩::= ‘’ ⟨ArrayColumns⟩‘’

⟨ArrayColumns⟩::= ⟨ArrayRows⟩ ‘;’ ⟨ArrayRows⟩

⟨ArrayRows⟩::= ⟨ArrayConst⟩ ‘,’ ⟨ArrayConst⟩

⟨ArrayConst⟩::= ⟨Constant⟩ | ⟨UnOpPrefix⟩NUMBER

| ERROR-REF

Appendix B

False Witness from Skink

The false witness generated by running the code in Listing 4.4 through Skink.

Listing B.1: The false witness produced by Listing 4.4.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <graphml xmlns:xsi=
3                   "http://www.w3.org/2001/XMLSchema-instance"
4                   xmlns=
5                   "http://graphml.graphdrawing.org/xmlns">
6
7  <key id="entry"           for="node"   attr.name="entry"
   attr.type="boolean"><default>false</default></key>
8  <key id="block"           for="node"   attr.name="block"
   attr.type="int"    />
9  <key id="node.src"        for="node"   attr.name="node.src"
   attr.type="string"/>
10 <key id="edge.src"        for="edge"   attr.name="edge.src"
   attr.type="string"/>
11 <key id="startline"       for="edge"   attr.name="startline"
   attr.type="int"    />
12 <key id="endline"         for="edge"   attr.name="endline"
   attr.type="int"    />
13 <key id="violation"       for="node"   attr.name="violation"
   attr.type="boolean"><default>false</default></key>
14 <key id="witness-type"    for="graph"
15 attr.name="witness-type"      attr.type="string"/>
16 <key id="sourcecodelang" for="graph"
17 attr.name="sourcecodelang"    attr.type="string"/>
18 <key id="producer"        for="graph" attr.name="producer"
   attr.type="string"/>
19 <key id="specification"   for="graph"
20 attr.name="specification"     attr.type="string"/>
```

```
<key id="programfile"     for="graph" attr.name="programfile"
attr.type="string"/>
<key id="programhash"     for="graph" attr.name="programhash"
attr.type="string"/>
<key id="memorymodel"     for="graph" attr.name="memorymodel"
attr.type="string"/>
<key id="architecture"    for="graph"
attr.name="architecture"      attr.type="string"/>
<key id="assumption"    for="edge" attr.name="assumption"
attr.type="string"/>
<key id="assumption.note"    for="edge"
        attr.name="assumption.note"   attr.type="string"/>
<key id="assumption.scope"    for="edge"
        attr.name="assumption.scope" attr.type="string"/>
<key id="assumption.resultfunction" for="edge"
        attr.name="assumption.resultfunction"
attr.type="string"/>

<graph edgedefault="directed">
  <data key="witness-type">violation_witness</data>
  <data key="sourcecodelang">C</data>
   <data key="producer">skink</data>
  <data key="specification">
    CHECK( init(main()), LTL(G ! call(__VERIFIER_error())))
  </data>
  <data key="programfile">false-type.c</data>
  <data key="programhash">
    a161f9f5d39596d218c726cd33bca3aef5ff3b8f
  </data>
  <data key="memorymodel">simple</data>
  <data key="architecture">32bit</data>

<node id="N0">
  <data key="entry">true</data>
</node>

<edge id="E0" source="N0" target="N1">
  <data key="assumption">\result == 0;</data>
  <data key="assumption.note">hex: 0</data>
  <data key="assumption.scope">main</data>
  <data key="assumption.resultfunction">
```

```
58      __VERIFIER_nondet_int
59    </data>
60    <data key="edge.src">
61      int C1=__VERIFIER_nondet_int();
62    </data>
63    <data key="startline">5</data>
64  </edge>
65
66  <node id="N1">
67  </node>
68
69  <edge id="E1" source="N1" target="N2">
70    <data key="assumption">\result == 0;</data>
71    <data key="assumption.note">unknown</data>
72    <data key="assumption.scope">main</data>
73    <data key="assumption.resultfunction">
74      __VERIFIER_nondet_int
75    </data>
76    <data key="edge.src">
77      int A1=__VERIFIER_nondet_int();
78    </data>
79    <data key="startline">7</data>
80  </edge>
81
82  <node id="N2">
83  </node>
84
85  <edge id="E2" source="N2" target="N3">
86    <data key="assumption">\result == 0;</data>
87    <data key="assumption.note">unknown</data>
88    <data key="assumption.scope">main</data>
89    <data key="assumption.resultfunction">
90      __VERIFIER_nondet_int
91    </data>
92    <data key="edge.src">
93      int B1=__VERIFIER_nondet_int();
94    </data>
95    <data key="startline">9</data>
96  </edge>
97
98  <node id="N3">
```

```
 99 </node>
100
101 <edge id="E3" source="N3" target="N4">
102   <data key="edge.src">
103     __VERIFIER_error();
104   </data>
105   <data key="startline">24</data>
106 </edge>
107
108 <node id="N4">
109   <data key="violation">true</data>
110 </node>
111
112 </graph>
113
114 </graphml>
```

Bibliography

[1] Robin Abraham and Martin Erwig. Type inference for spreadsheets. In *Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming - PPDP '06*, page 73, 2006.

[2] Robin Abraham and Martin Erwig. GoalDebug: A spreadsheet debugger for end users. *Proceedings - International Conference on Software Engineering*, pages 251–260, 2007.

[3] W Richards Adrion, Martha A Branstad, and John Cherniavsky. Validation, verification, and testing of computer software. *ACM Computing Surveys*, 14(2):159–192, 1982.

[4] Efthimia Aivaloglou, David Hoepelman, and Felienne Hermans. A grammar for spreadsheet formulas evaluated on two large datasets. *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation, SCAM 2015 - Proceedings*, pages 121–130, 2015.

[5] Efthimia Aivaloglou, David Hoepelman, and Felienne Hermans. Parsing Excel formulas : A grammar and its application on four large datasets. 2017.

[6] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, 1976.

[7] Kenneth Baker, Stephen Powell, Barry Lawson, Lynn Foster-Johnson, and Robert Burnham. Spreadsheet Engineering Research Project. http://faculty.tuck.dartmouth.edu/serp/. [Accessed: May 19, 2019].

[8] Florian Biermann, Wensheng Dou, and Peter Sestoft. Rewriting high-level spreadsheet structures into higher-order functional programs. In *International Symposium on Practical Aspects of Declarative Languages*, pages 20–35. Springer, 2018.

[9] Brian Bishop and Kevin McDaid. An Empirical Study of End-User Behaviour in Spreadsheet Error Detection & Correction. *arXiv preprint arXiv:0802.3479*, pages 165–176, 2008.

[10] M. Burnett, C. Cook, O. Pendse, G. Rothermel, J. Summet, and C. Wallace. End-user software engineering with assertions in the spreadsheet paradigm. *25th International Conference on Software Engineering, 2003. Proceedings.*, 2003(May):93–103, 2003.

[11] Margaret Burnett. What is end-user software engineering and why does it matter? *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5435 LNCS:15–28, 2009.

[12] Margaret Burnett, Curtis Cook, and Gregg Rothermel. End-user software engineering. *Communications of the ACM*, 47(9):53, 2004.

[13] Franck Cassez, Anthony M. Sloane, Matthew Roberts, Matthew Pigram, Pongsak Suvanpong, and Pablo Gonzalez de Aledo. Skink: Static analysis of programs in LLVM intermediate representation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10206 LNCS:380–384, 2017.

[14] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified latice model. *POPL '77 Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[15] Patrick Cousot and Radhia Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2-3):103–180, 1992.

[16] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[17] Jácome Cunha, João P. Fernandes, Hugo Ribeiro, and João Saraiva. Towards a catalog of spreadsheet smells. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7336 LNCS(PART 4):202–216, 2012.

[18] William J Doll and Gholamreza Torkzadeh. End-User Computing. 12(2):259–274, 2018.

[19] Marc Fisher and Gregg Rothermel. The EUSES Spreadsheet Corpus: A Shared Resource for Supporting Experimentation with Spreadsheet Dependability Mechanisms. Technical report.

[20] Marc Fisher, Gregg Rothermel, Darren Brown, Mingming Cao, Curtis Cook, and Margaret Burnett. Integrating automated test generation into the WYSIWYT spreadsheet testing methodology. *ACM Transactions on Software Engineering and Methodology*, 15(2):150–194, 2006.

[21] Bryan Ford. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation.

[22] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Refinement of trace abstraction. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5673 LNCS(Sas):69–85, 2009.

[23] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software Model Checking for People Who Love Automata. (8044):36–52, 2013.

[24] Felienne Hermans. Felienne Hermans | TU Delft Online. https://online-learning.tudelft.nl/instructors/felienne-hermans/. [Accessed: May 19, 2019].

[25] Felienne Hermans, Bas Jansen, Sohon Roy, Efthimia Aivaloglou, Alaaeddin Swidan, and David Hoepelman. Spreadsheets are Code: An Overview of Software Engineering Approaches Applied to Spreadsheets. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 56–65, 2016.

[26] Felienne Hermans and Emerson Murphy-Hill. Enron's Spreadsheets and Related Emails: A Dataset and Analysis. *Proceedings - International Conference on Software Engineering*, 2:7–16, 2015.

[27] Felienne Hermans, Martin Pinzger, and Arie Van Deursen. Detecting and visualizing inter-worksheet smells in spreadsheets. *Proceedings - International Conference on Software Engineering*, pages 441–451, 2012.

[28] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Detecting and refactoring code smells in spreadsheet formulas. *Empirical Software Engineering*, 2015.

[29] Michael Huth and Mark Ryan. Hoare Triples. In *Logic in Computer Science*, chapter 4, pages 262–265. Cambridge University Press, 2 edition, 2004.

[30] Capers Jones. Endluser programming. *Computer*, 28(9):68–70, 1995.

[31] Bennett Kankuzi, Bassey Isong, and Lucia Letlonkane. Using the Spreadsheet Paradigm to Introduce Fundamental Concepts of Programming to Novices. (July):39–45, 2017.

[32] Andrew J. Ko, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, Susan Wiedenbeck, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret

Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, and Henry Lieberman. The state of the art in end-user software engineering. *ACM Computing Surveys*, 43(3):1–44, 2011.

[33] Patrick W Koch, Birgit Hofer, and Franz Wotawa. Static Spreadsheet Analysis. *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, (1):167–174, 2016.

[34] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, 1992.

[35] David Lizcano, Javier Soriano, Genoveva López, and Javier J. Gutiérrez. Automatic verification and validation wizard in web-centred end-user software engineering. *Journal of Systems and Software*, 125:47–67, 2017.

[36] Microsoft. More Than 30 Million Users Make Microsoft Excel The World's Most Popular Spreadsheet Program | Stories. https://news.microsoft.com/1996/05/20/more-than-30-million-users-make-microsoft-excel-the-worlds-most-popular-spreadsheet-program/, 1996. [Accessed: May 19, 2019].

[37] Patrick O'Beirne, Felienne Hermans, Tie Cheng, and Mary Pat Campbell. EuSpRIG Horror Stories. http://www.eusprig.org/horror-stories.htm. [Accessed: May 19, 2019].

[38] OpenOffice. General error codes - Apache OpenOffice Wiki. https://wiki.openoffice.org/wiki/Documentation/OOo3_User_Guides/Calc_Guide/General_error_codes, 2018. [Accessed: May 19, 2019].

[39] Raymond R. Panko. Applying Code Inspection to Spreadsheet Testing. *Journal of Management Information Systems*, 16(2):159–176, 1999.

[40] R.R. Panko. Spreadsheet Errors: What We Know. What We Think We Can Do. *Proc. European Spreadsheet Risks Int. Grp. (EuSpRIG)*, page 9, 2000.

[41] Pak-Lok Poon, Fei-Ching Kuo, Huai Liu, and Tsong Yueh Chen. How can non-technical end users effectively test their spreadsheets? *Information Technology & People*, 27(4):440–462, 2014.

[42] Stephen G. Powell, Kenneth R. Baker, and Barry Lawson. A critical review of the literature on spreadsheet errors. *Decision Support Systems*, 46(1):128–138, 2008.

[43] K. Rajalingham, D. Chadwick, B. Knight, and D. Edwards. Quality control in spreadsheets: a software engineering-based approach to spreadsheet development. *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, 00(c):1–9, 2000.

[44] Nick Randolph, John Morris, and Gareth Lee. A Generalised Spreadsheet Verification Methodology. *Proceeding ACSC '02 Proceedings of the twenty-fifth Australasian conference on Computer science*, 4(February 2002):215–222, 2002.

[45] Christopher Scaffidi, Mary Shaw, and Brad Myers. Estimating the numbers of end users and end user programmers. *Proceedings - 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2005:207–214, 2005.

[46] David A. Schmidt. Data flow analysis is model checking of abstract interpretations. *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '98*, pages 38–48, 1998.

[47] Andrew F Seila. Spreadsheet Simulation. In *Winter Simulation Conference*, pages 11–18, 2006.

[48] Vijay D Silva, Daniel Kroening, and Georg Weissenbacher. A Survey of Automated Techniques for Formal Software Verificatio. *IEEE TRANSACTIONS*

ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, 27(7):1165–1178, 2008.

[49] Anthony M Sloane, Franck Cassez, and Scott Buckley. The sbt-rats parser generator plugin for scala (tool paper). In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, pages 110–113. ACM, 2016.

[50] SpreadsheetLab. XLParser. https://github.com/spreadsheetlab/XLParser. [Accessed: May 19, 2019].

[51] Liam Tung. 'Microsoft by the numbers' 2015: 700k Windows Store Apps, 1.2bn Office users | ZDNet. https://www.zdnet.com/article/microsoft-by-the-numbers-2015-700k-windows-store-apps-1-2bn-office-users/, 2015. [Accessed: May 19, 2019].

[52] Liang Xu, Shuo Wang, Wensheng Dou, Bo Yang, Chushu Gao, Jun Wei, and Tao Huang. Detecting faulty empty cells in spreadsheets. *25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings*, 2018-March(2):423–433, 2018.

[53] Aiko Yamashita and Leon Moonen. To what extent can maintenance problems be predicted by code smell detection?–an empirical study. *Information and Software Technology*, 55(12):2223–2242, 2013.

[54] Jie Zhang, Shi Han, Dan Hao, Lu Zhang, and Dongmei Zhang. Automated Refactoring of Nested-IF Formulae in Spreadsheets. pages 1–11, 2017.