Automated Model-Based Spreadsheet Debugging

Dissertation

zur Erlangung des Grades eines

D o k t o r s   d e r   N a t u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Thomas Schmitz

Dortmund

2017

Tag der mündlichen Prüfung: 24.08.2017

Dekan: Prof. Dr.-Ing. Gernot A. Fink

Gutachter:
Prof. Dr. Dietmar Jannach
Prof. Dr. Franz Wotawa

Abstract

Spreadsheets are interactive data organization and calculation programs that are developed in spreadsheet environments like Microsoft Excel or LibreOffice Calc. They are probably the most successful example of end-user developed software and are utilized in almost all branches and at all levels of companies. Although spreadsheets often support important decision making processes, they are, like all software, prone to error. In several cases, faults in spreadsheets have caused severe losses of money.

Spreadsheet developers are usually not educated in the practices of software development. As they are thus not familiar with quality control methods like systematic testing or debugging, they have to be supported by the spreadsheet environment itself to search for faults in their calculations in order to ensure the correctness and a better overall quality of the developed spreadsheets.

This thesis by publication introduces several approaches to locate faults in spreadsheets. The presented approaches are based on the principles of Model-Based Diagnosis (MBD), which is a technique to find the possible reasons why a system does not behave as expected. Several new algorithmic enhancements of the general MBD approach are combined in this thesis to allow spreadsheet users to debug their spreadsheets and to efficiently find the reason of the observed unexpected output values. In order to assure a seamless integration into the environment that is well-known to the spreadsheet developers, the presented approaches are implemented as an extension for Microsoft Excel.

The first part of the thesis outlines the different algorithmic approaches that are introduced in this thesis and summarizes the improvements that were achieved over the general MBD approach. In the second part, the appendix, a selection of the author's publications are presented. These publications comprise (a) a survey of the research in the area of spreadsheet quality assurance, (b) a work describing how to adapt the general MBD approach to spreadsheets, (c) two new algorithmic improvements of the general technique to speed up the calculation of the possible reasons of an observed fault, (d) a new concept and algorithm to efficiently determine questions that a user can be asked during debugging in order to reduce the number of possible reasons for the observed unexpected output values, and (e) a new method to find faults in a set of spreadsheets and a new corpus of real-world spreadsheets containing faults that can be used to evaluate the proposed debugging approaches.

Contents

1 Introduction

Spreadsheets are interactive data organization and calculation programs that are developed in spreadsheet environments like Microsoft Excel or LibreOffice Calc. They are widely used in business as well as for private calculation tasks and are therefore the most wide-spread type of end-user developed software [Sca+05]. The success of spreadsheets has several reasons. First, as spreadsheets are designed in a visual environment, they are easy to develop also for users without a background in software development and they are more flexible than traditional software [Hun+05]. Yet, they are powerful enough for many daily calculation tasks like budget planning or tax computations. In addition, spreadsheets can be useful even at the beginning of their development as they can start as a simple data storage and then evolve to a complex calculation tool. For example, a list of expenses can evolve to a complete budget calculation. Therefore, spreadsheets can cover a wider range of tasks over time as their development progresses.

Especially in the industry, spreadsheets are a common tool for calculations in daily business as well as in preparation for business decisions [Pan+12]. In most companies a wide range of spreadsheets is created and maintained. For example, in the Enron Corporation, formerly one of the biggest energy companies in the US, 16,189 unique spreadsheets were sent by email during a time frame of two years [Her+15].

1.1 Faults in Spreadsheets

Although the creation of spreadsheets is often not perceived as software development, a spreadsheet that contains formulas in fact is a software that calculates the values of the output cells given the input values. These spreadsheets, as all other software, are prone to error [Pan98].

When speaking about errors, several definitions for the words “fault”, “error”, and “failure” exist in the research literature [Jan+14a]. According to the IEEE Standard Classification for Software Anomalies [IEE10] an “error” is a misapprehension on side of the one developing a software caused by a mistake or misconception occurring in the human thought process. A "fault" is the manifestation of an "error" within a software which may be causing a "failure". A "failure" is the deviation of the observed behavior of the software from the expectations. However, in the research literature the terms "fault" and "error" are often used interchangeably. In order to comply with the IEEE standard, in this thesis the terms "fault" and "error" are used according to the given definitions.

Faults in spreadsheets have already caused severe financial losses in the past. The consulting company F1F9 lists twelve famous cases of faulty spreadsheets, many of which had severe impacts [F1F]. One well-known example is the economic study of Reinhart and Rogoff, which states a strong negative relation between the debt of a country and its economic growth [Rei+10]. Politicians used this study to argue against new debts and changed their strategies accordingly. Later, Herndon *et al.* showed that faults in a spreadsheet led to miscalculations in the study and that the discovered relation was much weaker than originally stated [Her+13]. As another example, in 2014 the Wall Street Journal informed about a fault in a spreadsheet that caused an overestimation of the equity value of the software company Tibco by $100 million [Tan14].

When analyzing a spreadsheet for such important faults, different approaches are required to locate the various types of faults that can be made when designing a spreadsheet. In the literature, several taxonomies were proposed to classify spreadsheet errors [Pan98; Pur+06; Pow+08; Pan+10]. In this thesis, a combined taxonomy is used to structure the possible errors in a systematic way. The error taxonomy is shown in Figure 1.1 and can be summarized as follows.

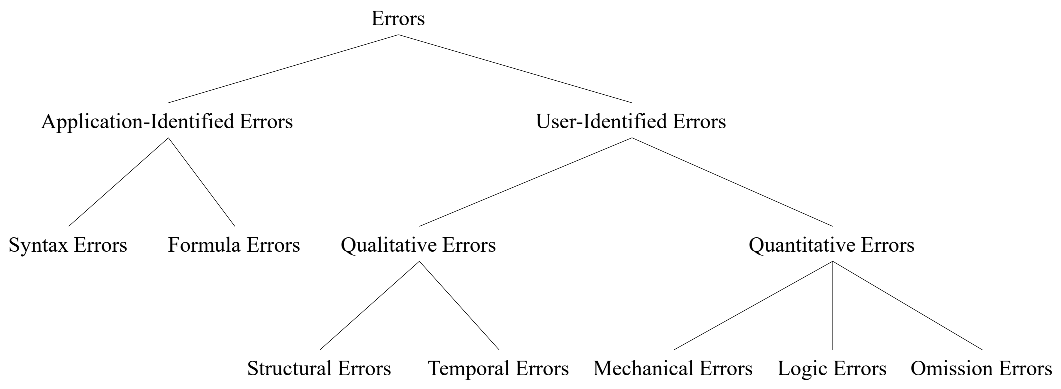


Figure 1.1: Taxonomy of spreadsheet errors, adapted from [Abe15].

Errors in a spreadsheet can be classified into two main categories. *Application-Identified Errors* can be automatically detected with certainty by the spreadsheet environment. Microsoft Excel, for example, automatically detects *Syntax Errors* and a user is not able to put a syntactically faulty formula in a cell as the spreadsheet environment will inform the user that the written formula is faulty. *Formula Errors* are detected by Excel and similar environments when they evaluate the value of a formula, for example, when dividing by zero.

In contrast to *Application-Identified Errors*, *User-Identified Errors* cannot be detected by the spreadsheet environment but have to be detected by the user or otherwise remain unknown. These errors can be split into two more sub-categories. *Qualitative Errors* do not result in a wrong calculation outcome in the current version of the spreadsheet but could result in a faulty value when the spreadsheet is changed later. They comprise *Structural Errors* and *Temporal Errors*. *Structural Errors* describe errors in the design of a spreadsheet, for example, hard-coded values in a formula that should be inputs. *Temporal Errors* summarize those values or formulas that are only correct for a specific time period and can be wrong at a later date, for example, a value that is only correct for a specific day of the year but is not labeled as such.

The group of errors which immediately result in faulty values in the current version of the spreadsheet is called *Quantitative Errors*. These errors can be split into *Mechanical Errors*, which describe errors by a user in the process of typing a formula, *Logic Errors*, that occur when a wrong function or algorithm is used, and *Omission Errors*, that occur if the user does not incorporate some aspect of the task he or she tries to solve. The main focus of this thesis lies on these *Quantitative Errors*, as these errors have a direct impact on the result of the spreadsheet and are therefore probably the most important ones to fix.

1.2 Spreadsheet Quality Assurance

To find possible faults when developing spreadsheets and to use the spreadsheets for important tasks without any risks, the quality of the spreadsheets has to be assured. This is potentially even more important for spreadsheets than for traditional software, as spreadsheet users who do not have a software development background might not be aware of the high risks. However, approaches for spreadsheet quality assurance (QA) have to be well integrated into the spreadsheet environment and easy to use even for users without any knowledge in software development. Since one important factor of the success of spreadsheets is their high flexibility compared to other software, this advantage should not be removed by the QA approaches.

Over the years, several techniques for spreadsheet quality assurance have been proposed in the research literature. In [Jan+14a], which is included in this thesis, a survey is presented that classifies the existing approaches for spreadsheet QA in two dimensions. The first dimension is used to distinguish between approaches that are made for locating faults in a spreadsheet and approaches that should help to avoid making errors in the first place. The second dimension was made to differentiate between the approaches based on how they fulfill their tasks. Table 1.1 shows for which tasks the different types of techniques can be used.

Table 1.1: Overview of main categories of automated spreadsheet QA [Jan+14a].

	Finding faults	Avoiding errors
Visualization-based approaches	✓	✓
Static code analysis & reports	✓	✓
Testing-based techniques	✓	
Automated fault localization & repair	✓	
Model-driven development approaches		✓
Design and maintenance support		✓

The different groups of techniques can be summarized as follows [Jan+14a].

Visualization-based approaches: Approaches of this group help the user by providing visualizations of the spreadsheet. Most of the proposed representations are utilized to explain the dependencies between the cells, groups of cells, or even the different worksheets of a spreadsheet. Such visualizations can help the user in the tasks of both categories finding faults as well as avoiding errors, because the user can detect anomalies in the existing dependencies or use them to improve the design of the spreadsheet to avoid making errors in the future.

Static code analysis & reports: Methods of this category perform static analyses of the formulas and data of a spreadsheet. They can be used to find irregularities and to point out problematic areas that are prone to be faulty or that can often lead to faults in subsequent versions of the spreadsheet. Therefore, these approaches can also be used to find faults or to avoid errors. They include techniques like “code smells”, detecting duplicates of data, or other approaches typically found in commercial tools that detect suspicious cells.

Testing-based techniques: Techniques in this category are based on the general approach of systematic testing. The approaches support the user in creating and organizing test cases that specify the input values of the spreadsheet and the expected output values of some formulas given the input values. As these techniques do not change the way a spreadsheet itself is built, they only support the user to find faults but not to avoid making errors. However, they can also be used to find faults during the construction of the spreadsheet and thus help to improve the quality of the built spreadsheet. The methods of this category include techniques like test case management, automated test case generation, or the analysis of the test coverage.

Automated fault localization & repair: The approaches presented in this thesis mostly fall into the category of automated fault localization & repair, which contains the techniques that computationally determine the possible reasons of a fault or an unexpected calculation outcome. To perform these calculations they typically require additional information provided by the user about unexpected output values. In addition to calculating the possibly faulty formulas, some approaches in this category provide suggestions of how these formulas could be "repaired".

Model-driven development approaches: In contrast to the previous categories, model-driven development approaches do not aim to find faults in an existing spreadsheet but propose a method to systematically develop a spreadsheet. This way these approaches try to support the user in developing spreadsheets that do not contain any faults. The main idea of these approaches is to use (object-oriented) conceptual models or model-driven software development techniques. These concepts have the advantage of adding an additional layer of abstraction and thus eliminate some types of possible faults like copy-and-paste errors or mechanical errors.

Design and maintenance support: Methods of this category help the spreadsheet developer when designing or maintaining a spreadsheet by automating common tasks or providing new methods to design spreadsheets in order to avoid common faults like range or reference errors. These techniques include, for example, refactoring tools, methods to avoid wrong cell references, and exception handling.

1.3 Overview of this Thesis

This *thesis by publication* combines several approaches to automatically locate faults in a spreadsheet. Most of these approaches are based upon and extend the approach of using Model-Based Diagnosis (MBD) for spreadsheets.

MBD is a systematic approach to find the possible reasons why a system under observation does not behave as expected. As it is shown in the structural overview of this thesis in Figure 1.2, Chapter 2 introduces the general idea of MBD in more detail and describes how it can be adapted to efficiently search for possibly faulty formulas in spreadsheets based on test cases that specify input values and corresponding expected output values for a spreadsheet [Jan+16a]. The general MBD approach, however, has two limitations depending on the structure and size of the analyzed spreadsheet. In the other chapters of this thesis by publication these limitations of the general MBD approach are addressed and improvements are introduced to mitigate them.

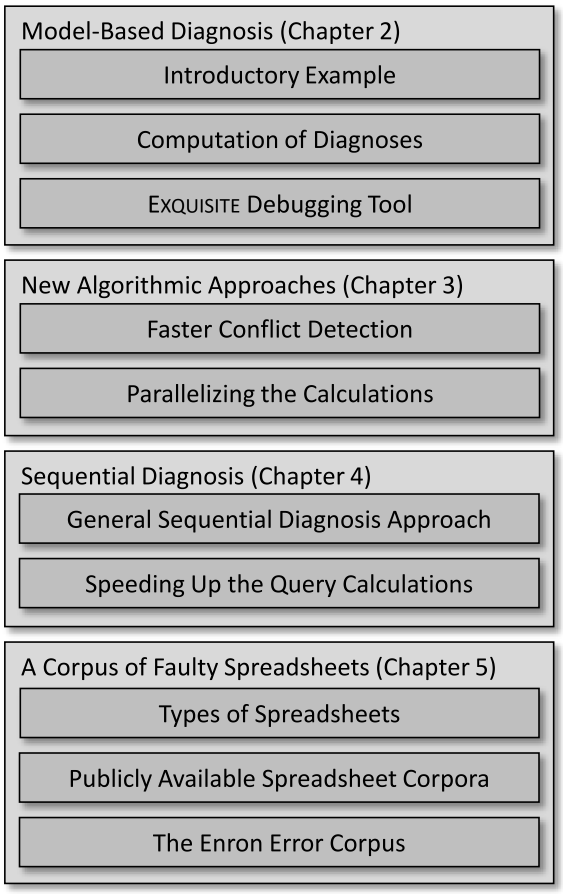

Model-Based Diagnosis (Chapter 2)
Introductory Example
Computation of Diagnoses
EXQUISITE Debugging Tool
New Algorithmic Approaches (Chapter 3)
Faster Conflict Detection
Parallelizing the Calculations
Sequential Diagnosis (Chapter 4)
General Sequential Diagnosis Approach
Speeding Up the Query Calculations
A Corpus of Faulty Spreadsheets (Chapter 5)
Types of Spreadsheets
Publicly Available Spreadsheet Corpora
The Enron Error Corpus

Figure 1.2: Structural overview of this thesis.

One limitation of the general approach is that for large or complex spreadsheets, the time required to calculate the possible reasons of a fault can exceed the time that is acceptable in an interactive setting. Therefore, two new algorithmic enhancements are proposed to speed up the computation (Chapter 3). First, in Section 3.1 a new approach is presented to efficiently search for so-called *conflicts*, which are sets of formulas in a spreadsheet that cannot all be correct at the same time [Shc+15b]. Second, the general MBD algorithm is parallelized to utilize the full computational capabilities of modern computer hardware (Section 3.2) [Jan+16b].

The other limitation of the general MBD approach is addressed in Chapter 4. Depending on the provided test cases too many possible reasons for a fault can be returned by the diagnosis algorithm so that a user cannot inspect all of them in reasonable time. Therefore, in [Shc+16b] a new algorithm is presented to efficiently determine questions that can be asked to the user interactively in order to reduce the number of possible reasons and to finally find the true reason of the observed unexpected output.

One open challenge that all research about spreadsheet QA faces is how to evaluate new approaches in a way that allows to draw conclusions about the effectiveness of the approach in real-world settings. Currently most approaches for spreadsheet debugging are evaluated on real-world spreadsheets which are altered by the researchers so that they contain faults. However, whether or not these artificially injected faults are representative for faults encountered in the real world remains unknown. Therefore, in Chapter 5 a work is presented in which the publicly available spreadsheets and emails of the Enron company are used to search for real faults and to build a corpus of these real-life faulty spreadsheets [Sch+16a].

1.4 Publications

This thesis by publication includes six of the author's publications. In this section, the individual contributions of the author are stated for each publication. The complete list of the author's publications can be found in the appendix.

1.4.1 Avoiding, Finding and Fixing Spreadsheet Errors – A Survey of Automated Approaches for Spreadsheet QA

Dietmar Jannach, Thomas Schmitz, Birgit Hofer, and Franz Wotawa. "Avoiding, Finding and Fixing Spreadsheet Errors - A Survey of Automated Approaches for Spreadsheet QA". in: *Journal of Systems and Software* 94 (2014), pp. 129–150

This survey was a joint effort with Dietmar Jannach, Birgit Hofer, and Franz Wotawa. The author of this thesis searched for most of the relevant works, categorized all of them, and wrote parts of the text.

1.4.2 Model-Based Diagnosis of Spreadsheet Programs

Dietmar Jannach and Thomas Schmitz. "Model-Based Diagnosis of Spreadsheet Programs: A Constraint-based Debugging Approach". In: *Automated Software Engineering* 23.1 (2016), pp. 105–144

This work was written together with Dietmar Jannach. The approach presented in this paper is based on a preliminary work by Dietmar Jannach, Arash Baharloo, and David Williamson [Jan+13]. The author of this thesis designed the parallelization techniques in collaboration with Dietmar Jannach, did the implementations that were required in addition to the previous work, designed and performed the evaluations as well as the user study, and wrote the corresponding parts of the text.

1.4.3 MergeXplain: Fast Computation of Multiple Conflicts for Diagnosis

Kostyantyn Shchekotykhin, Dietmar Jannach, and Thomas Schmitz. “MergeXplain: Fast Computation of Multiple Conflicts for Diagnosis”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2015)*. 2015, pp. 3221–3228

The research of this work was a joint effort with Kostyantyn Shchekotykhin and Dietmar Jannach. The proposed MergeXplain algorithm was designed in a collaboration between Kostyantyn Shchekotykhin and the author of this thesis, who also implemented and evaluated it.

1.4.4 Parallel Model-Based Diagnosis on Multi-Core Computers

Dietmar Jannach, Thomas Schmitz, and Kostyantyn Shchekotykhin. “Parallel Model-Based Diagnosis On Multi-Core Computers”. In: *Journal of Artificial Intelligence Research* 55 (2016), pp. 835–887

The paper is the result of a joint work with Dietmar Jannach and Kostyantyn Shchekotykhin. The author of this thesis designed the parallelization approaches together with Dietmar Jannach, implemented and evaluated them, and wrote parts of the text.

1.4.5 Efficient Sequential Model-Based Fault-Localization with Partial Diagnoses

Kostyantyn Shchekotykhin, Thomas Schmitz, and Dietmar Jannach. “Efficient Sequential Model-Based Fault-Localization with Partial Diagnoses”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2016)*. 2016, pp. 1251–1257

The work was a joint effort with Kostyantyn Shchekotykhin and Dietmar Jannach. Most parts of the text were written by the author of this thesis who also contributed to the design of the new approach, implemented, and evaluated it.

1.4.6 Finding Errors in the Enron Spreadsheet Corpus

Thomas Schmitz and Dietmar Jannach. “Finding Errors in the Enron Spreadsheet Corpus”. In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2016)*. 2016, pp. 157–161

This paper was written together with Dietmar Jannach. The author of this thesis designed the different approaches to search for faults in the spreadsheets in collaboration with Dietmar Jannach and wrote the text of the paper.

Model-Based Diagnosis for Spreadsheets

2

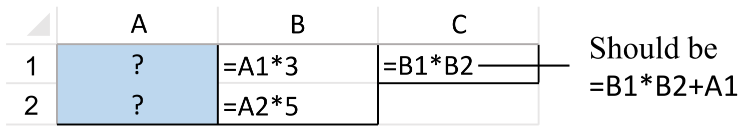
One of the possible ways presented in Section 1.2 to assure the quality of a spreadsheet is to do systematic testing. In order to systematically test a spreadsheet a user has to write so-called *test cases* by specifying the input values of the spreadsheet and expected values for some of its formula cells. If these expected values differ from what the spreadsheet environment actually computes given the input values, there has to be a fault somewhere in the formulas of the spreadsheet. In this case the task of spreadsheet debugging is to locate the fault, for example, by utilizing a debugging approach like Model-Based Diagnosis (MBD), which can be used to find the possible reasons of the unexpected calculation outcomes. How MBD can be applied to spreadsheets is described formally in [Jan+16a] and summarized in this chapter.

The principles of the general MBD technique were proposed in the 1980s [Kle+87; Rei87]. In these early works, MBD was used to search for faults in digital circuits. It can, however, be used to debug any kind of observable system for which the functionality can be simulated in a deterministic way. The system requires information about the expected behavior of the individual components of the system and how these components are connected. If there is a discrepancy between the simulated expected behavior of the system and an observation of its real behavior, the task of the MBD approach is to determine the sets of components that could possibly be the reason of this discrepancy. These candidates that, if assumed to behave in a faulty way, explain the faulty behavior of the system are called *diagnoses*. Formal definitions of diagnoses and other terms relevant in the MBD setting are given in [Jan+16a], which is included in this thesis by publication.

In the context of spreadsheets, the system is described as a set of formulas that represent the diagnosable components of the system. The observations are given as a test case that specifies the input values of the spreadsheet and some expected output values of the formulas. If there is a discrepancy between the specified test case and the calculated outcomes of the formulas given the same inputs, MBD can be used to find the sets of formulas that can be the reason for the observed discrepancy.

2.1 Introductory Example

In this thesis, a small example spreadsheet is used to explain how the MBD technique can help to determine the possibly faulty formulas in that spreadsheet. The formulas of the example spreadsheet are shown in Figure 2.1. Assume that the spreadsheet developer forgot to add the value of A1 in the formula of cell C1.

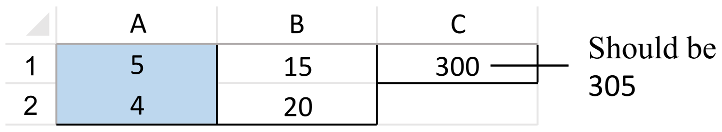


	A	B	C
1	?	=A1*3	=B1*B2
2	?	=A2*5	

Should be
=B1*B2+A1

Figure 2.1: A faulty spreadsheet.

If the user enters some values for the input cells in column A, as shown in Figure 2.2, he or she could realize that the result in cell C1 is wrong, because it should be 305 for the given input values. The values for the two input cells together with the expected output value therefore form a test case that describes a discrepancy between the expected and the observed behavior of the spreadsheet.



	A	B	C
1	5	15	300
2	4	20	

Should be
305

Figure 2.2: A test case for the faulty spreadsheet.

Once the user has detected the discrepancy, he or she can use the MBD approach to locate the possible reasons that can explain it. With the test case shown in Figure 2.2, the MBD approach would return two diagnoses as the possible reasons for the observed discrepancy: $\{C1\}$ and $\{B1, B2\}$. This means that either the formula in cell C1 is faulty or that the two formulas in the cells B1 and B2 both have to be faulty. In this example, $\{C1\}$ is the true diagnosis as the formula of cell C1 is in fact faulty. The diagnosis $\{B1, B2\}$ is therefore not true. In general, it is more unlikely that diagnoses containing multiple cells are true, because it would require the developer to have made multiple errors instead of just one.

The rationale behind the diagnoses is the following. The formula in C1 can be changed in a way that the result of the calculation would be 305, for example, by changing the formula to "=B1*B2+A1", "=B1*B2+5", or "=305". Therefore $\{C1\}$ is a diagnosis. $\{B1\}$, however, cannot be a diagnosis because changing the formula in B1 alone cannot result in the expected value in C1, assuming that only integer values are used as in the given test case. The same is true for cell B2. Both cells B1 and B2 have to be changed in order to achieve the expected result of 305 in C1 and therefore $\{B1, B2\}$ is another diagnosis.

2.2 Computation of the Diagnoses

In [Rei87] Reiter proposes an algorithm to build a *Hitting Set Tree* (HS-Tree) in order to determine the diagnoses of a faulty system under observation. The algorithm uses the concept of *conflicts*, which are sets of components of the system that cannot all be correct at the same time given the observations. In the example spreadsheet of Section 2.1 there are two of these conflicts, namely $\{\{B1, C1\}, \{C1, B2\}\}$. This means that the formulas of B1 and C1 cannot be both correct as well as the formulas of C1 and B2. The reason is that if, for example, both B1 and C1 would be assumed to be correct, the calculation could not result in the expected value. The same is true for the two formulas C1 and B2.

The idea of the HS-Tree algorithm is to systematically test different hypotheses about the health state of the components. As the algorithm progresses, it tests hypotheses involving more and more components that are assumed to be faulty. In the beginning it therefore assumes that everything is working correctly. If this assumption does not hold because the expected behavior conflicts with the observed behavior, the algorithm systematically tries to resolve all conflicts by assuming that at least one component of each conflict is faulty. To achieve this, the algorithm builds a tree in breadth-first manner to search for the hitting sets of the conflicts, i.e., sets that "hit" every conflict of the system. In his work Reiter showed that these hitting sets correspond to the diagnoses. To find the hitting sets efficiently, the algorithm utilizes a set of tree pruning rules to cut subtrees that cannot lead to further diagnoses. The resulting HS-Tree for the example spreadsheet is shown in Figure 2.3 and explained in the following.

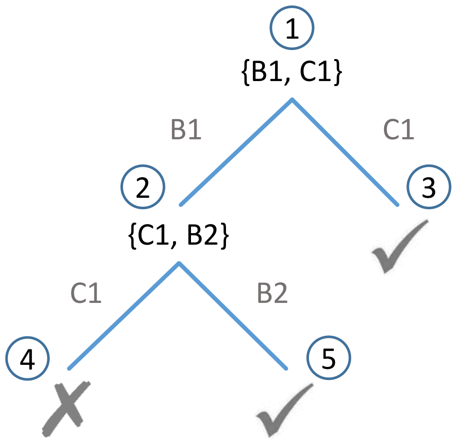


Figure 2.3: The resulting HS-Tree for the example spreadsheet.

At node ①, the algorithm searches for a conflict when all components (formulas) are assumed to be correct. To determine the conflicts, some kind of conflict detection technique is required that can calculate a conflict for the given system. For the example spreadsheet, assume that such a conflict detection technique would return one of the existing conflicts, for example, $\{B1, C1\}$. Node ① is then labeled with the found conflict and the algorithm will expand the search tree for each component inside this conflict.

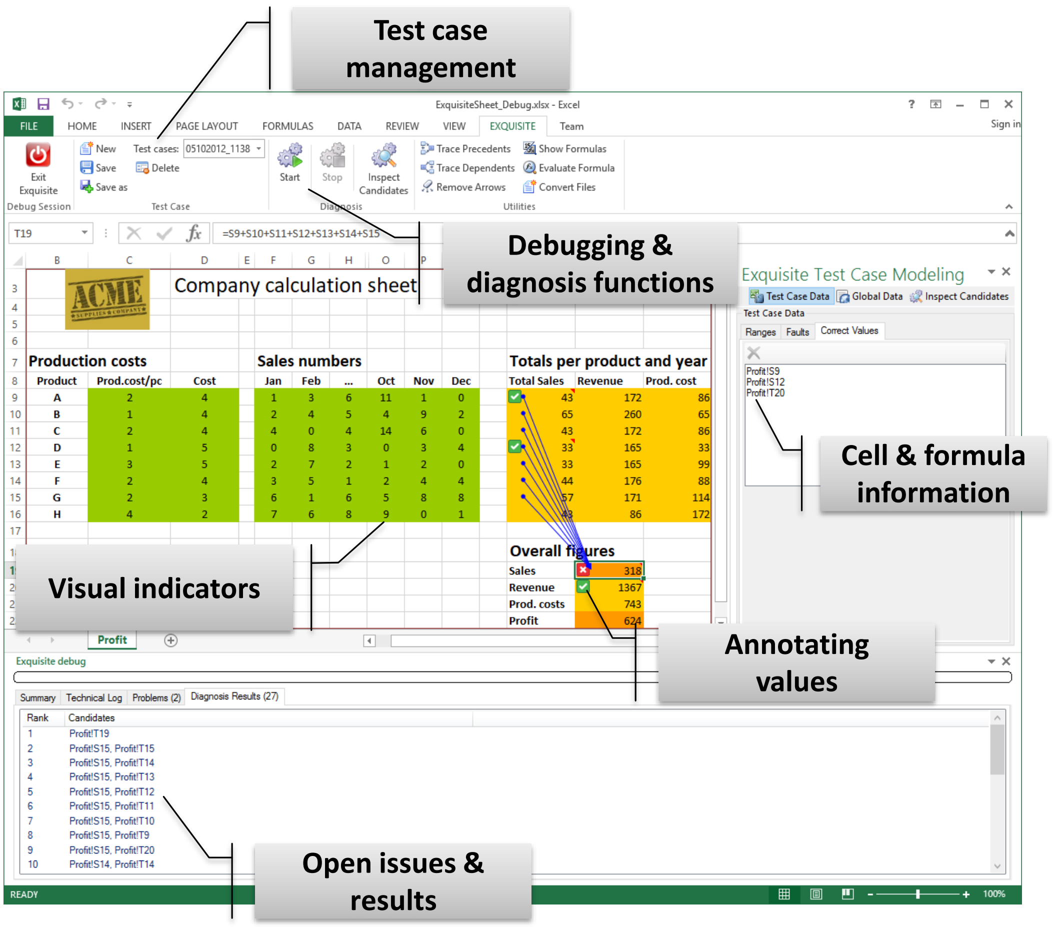
For node ②, the algorithm assumes the formula of B1 to be faulty and therefore checks if the spreadsheet still has a conflict when the formula of B1 is ignored. Since the spreadsheet has another conflict $\{B2, C1\}$, this conflict will be found this time and node ② will be labeled with the newly found conflict. At node ③, C1 is assumed to be faulty, as shown in Figure 2.3. Because no other conflict remains when the formula of C1 is ignored, the algorithm has found the diagnosis $\{C1\}$ and the node is labeled with a check mark.

On the next level, the HS-Tree algorithm expands node ② by creating two new nodes for the components of the conflict found for this node. Node ④, however, does not have to be further inspected and is closed, since for this node the resulting diagnosis $\{B1, C1\}$ would be a superset of the already found diagnosis $\{C1\}$ and is thus not relevant. Last, at node ⑤ the formulas of both cells B1 and B2 are considered to be faulty and the diagnosis $\{B1, B2\}$ is found, as no other conflict remains. Since all leaf nodes now either result in a diagnosis or are closed, the algorithm is finished and has found the two diagnoses $\{C1\}$ and $\{B1, B2\}$.

To compute the conflicts, different conflict detection techniques can be used. However, in order for the original HS-Tree algorithm of Reiter to work correctly, the minimality of the returned conflicts has to be ensured, because the algorithm was faulty regarding the use of non-minimal conflicts. In [Gre+89] Greiner *et al.* proposed an extension to the original algorithm to correct it in cases in which non-minimal conflicts are returned by the used conflict detection technique. In the implementations discussed in this thesis, however, QUICKXPLAIN [Jun04] and MERGEXPLAIN [Shc+15b] are used to compute the conflicts. Since both of these techniques are guaranteed to only return minimal conflicts, the correction by Greiner *et al.* is not required.

2.3 An Interactive Tool for Model-Based Spreadsheet Debugging

In order to test and evaluate the proposed approaches with users on real-world spreadsheets, the Model-Based Diagnosis approach for spreadsheets was implemented as an extension to Microsoft Excel, called EXQUISITE. An overview of the tool is shown in Figure 2.4.



Company calculation sheet

Production costs			Sales numbers						Totals per product and year		
Product	Prod.cost/pc	Cost	Jan	Feb	...	Oct	Nov	Dec	Total Sales	Revenue	Prod. cost
A	2	4	1	3	6	11	1	0	✓ 43	172	86
B	1	4	2	4	5	4	9	2	65	260	65
C	2	4	4	0	4	14	6	0	43	172	86
D	1	5	0	8	3	0	3	4	✓ 33	165	33
E	3	5	2	7	2	1	2	0	33	165	99
F	2	4	3	5	1	2	4	4	44	176	88
G	2	3	6	1	6	5	8	8	57	171	114
H	4	2	7	6	8	9	0	1	43	86	172

Overall figures	
Sales	✗ 318
Revenue	✓ 1367
Prod. costs	743
Profit	624

Rank	Candidates
1	Profit!T19
2	Profit!S15, Profit!T15
3	Profit!S15, Profit!T14
4	Profit!S15, Profit!T13
5	Profit!S15, Profit!T12
6	Profit!S15, Profit!T11
7	Profit!S15, Profit!T10
8	Profit!S15, Profit!T9
9	Profit!S15, Profit!T20
10	Profit!S14, Profit!T14

Figure 2.4: EXQUISITE, a Model-Based spreadsheet debugging tool [Jan+16a].

EXQUISITE: In the following an exemplary usage of EXQUISITE is described. When the debugging mode is started the tool automatically colors the cells according to their role in the spreadsheet. Input cells are colored in green, intermediate calculations in yellow, and output cells in orange. This colorization alone can help the user to spot some kinds of faults, for example, range errors or unused inputs, which are not highlighted. The user can then enter values for the input cells without overriding the values of the original spreadsheet and state expected values for the interim and output cells. The annotated values are shown as a list next to the spreadsheet and are also highlighted in the spreadsheet with a check mark for correct values and a cross for faulty values. The specified test cases can be saved and loaded at later times to support the test case specification over multiple sessions. Once a user detects a discrepancy between the expected and the observed behavior of the spreadsheet, he or she can start the debugging functionality. The system will then determine the diagnoses and present them as a list in the results section, which the user can inspect. By clicking on an item of the list, the cells containing the possibly faulty formulas are highlighted in the spreadsheet and arrows point to their precedents as well as dependents.

A preliminary version of the tool was already presented in [Jan+13]. In [Jan+16a] improvements to the tool as well as the algorithms behind it are presented and the performance of these new algorithmic approaches is evaluated (see Chapter 3).

User study: To evaluate if the MBD approach is advantageous for the users to debug a faulty spreadsheet, also a user study was performed. In this study, 24 participants had to locate a fault in a profit calculation spreadsheet. The participants were randomly split into two groups and were given a description about how the spreadsheet should work and an example with values that the spreadsheet should calculate. The first group had to locate the fault without using EXQUISITE, while the second group was introduced to the functionality of the add-in and used it to calculate a set of formulas that could be the reason of the fault. In both cases the users had to inspect the formulas which they thought to be faulty in order to find out what the real fault was.

The results of the study show that EXQUISITE can indeed help to locate faults in a spreadsheet. The participants using the tool found the injected fault faster on average than the participants not using it (less than 3 minutes compared to more than 9 minutes). In addition, of the participants not using the tool, 33% were not able to locate the fault at all in the given time frame of 30 minutes.

3 New Algorithmic Approaches for Faster Calculation of Diagnoses

The general MBD approach proved to be promising for spreadsheet debugging. However, for complex or large spreadsheets the time required to calculate the diagnoses can exceed the time that is acceptable in an interactive setting, in which a user expects a result almost instantly or at most after a few seconds. Therefore, in the next two sections two new algorithmic approaches to speed up the calculation of diagnoses are summarized. The full papers can be found in the appendix of this thesis by publication.

3.1 Faster Conflict Detection

As discussed in Section 2.2, the HS-Tree algorithm relies on some conflict detection technique that calculates the conflicts. The HS-Tree algorithm then uses these conflicts to determine the diagnoses. QUICKXPLAIN [Jun04] is an efficient divide-and-conquer technique proposed by Junker to determine such conflicts. For large or complex spreadsheets, however, many conflicts can exist. In these cases the HS-Tree algorithm will call QUICKXPLAIN each time a new conflict is required, i.e., when all known conflicts are already solved at the current node of the tree. Since QUICKXPLAIN only returns a single conflict for each call, the search for conflicts has to be “restarted” each time.

MERGEXPLAIN: To speed up the overall calculation of diagnoses and to solve the problem of the slow “restart” of the conflict search, in [Shc+15b] a new approach, called MERGEXPLAIN, is proposed that can calculate multiple conflicts in a single call. The rationale of this technique is that more time is spent to efficiently search for conflicts at the beginning of the calculation of diagnoses and in return the search for conflicts does not have to be restarted so often when the HS-Tree is built, because in most cases one of the previously found conflicts can be reused.

An example of how MERGEXPLAIN searches for conflicts is shown in Figure 3.1 and explained in the following. In the example the faulty system has 8 components or

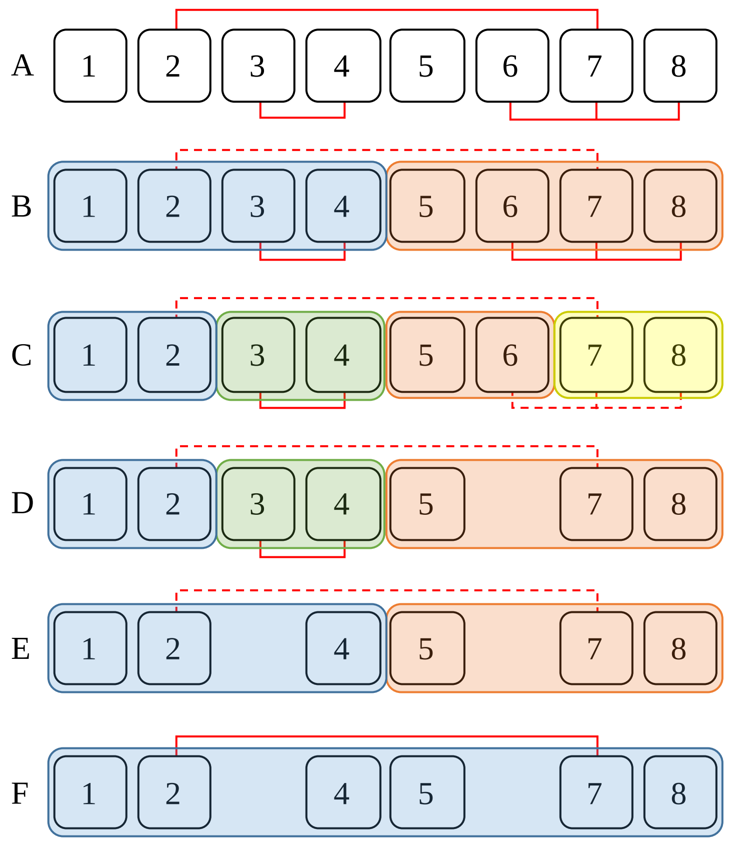


Figure 3.1: Example of MERGEXPLAIN searching for three conflicts shown as red lines between the components 1 to 8.

formulas. The system has 3 conflicts and the goal of MERGEXPLAIN is to find at least one of them or more, if possible. The conflicts, shown as red lines in Figure 3.1, are $\{2, 7\}$, $\{3, 4\}$, and $\{6, 7, 8\}$. First, MERGEXPLAIN recursively splits the sets of components into two separate sets, as long as the components of the set still contain at least one conflict. Because of this step, subsets of the components that do not contain any conflicts can be quickly excluded from further examinations.

Since the 8 components of the system contain a conflict (step A), they are split into two sets (step B). In step C, both sets are split again, because they both still contain a conflict. This time, however, on the right-hand side the last conflict, shown as a dotted line in Figure 3.1, was split and thus the two sets $\{5, 6\}$ and $\{6, 7\}$ both do not contain a conflict anymore. Therefore, the algorithm re-combines them and uses Junker's QUICKXPLAIN to locate a conflict in this set of components. As this set of components would not have been split if it did not contain any conflict, it is known to contain one conflict at least. In addition, because this set of components resulted from repetitive splitting of the original components and thus it is comparatively small, QUICKXPLAIN will find the conflict $\{6, 7, 8\}$ rather quickly. After this conflict has been found, one of the conflict's components, for example, component 6, is removed from further investigations to resolve the current conflict and QUICKXPLAIN is iteratively called again in order to find another conflict, if one exists (step D). In step E, the same was done for the left-hand side of the components and here the conflict $\{3, 4\}$ was found. Since no more conflicts remain in both halves of the components, the algorithm continues to merge the two sets of components again and searches for the last remaining conflict (step F).

A detailed description of the MERGEXPLAIN algorithm can be found in [Shc+15b], which is included in this thesis by publication. In this paper, it is proven that MERGEXPLAIN will only return minimal conflicts because it internally uses QUICKXPLAIN, which also only returns minimal conflicts. MERGEXPLAIN is also proven to always return at least one conflict or more. However, because it is not guaranteed to return all existing conflicts of a diagnosable system, MERGEXPLAIN still has to be called multiple times when used to determine the conflicts that are required to calculate the diagnoses.

Evaluation: To evaluate the proposed approach it was compared to QUICKXPLAIN when used by the HS-Tree algorithm to calculate a small subset of the diagnoses [Shc+15b]. The different tested systems contained digital circuits, Constraint Satisfaction Problems (CSPs), spreadsheets, as well as artificial systems to simulate different problem characteristics. The average reductions of the calculation times are summarized in Table 3.1.

Table 3.1: Average reductions of computation times when using MERGEXPLAIN compared to QUICKXPLAIN to search for five diagnoses with the HS-Tree algorithm [Shc+15b].

System type	Avg. reduction
Digital circuits	27%
Constraint Satisfactions Problems	22%
Spreadsheets	15%
Simulation experiments	42%

The efficiency of the approach very much depends on the structure of the problem and thus the improvements vary for the individual problems. Although for some problem instances no speedups could be achieved, for others the time required to calculate the diagnoses could be reduced by up to 54%. Therefore, additional simulation experiments were performed, in which artificial problems with different characteristics were tested. The goal of this evaluation was to find out which problem characteristics lead to the highest performance improvements. Among others, one result is that depending on the characteristics of the conflicts MERGEXPLAIN can achieve improvements of up to 76% over QUICKXPLAIN while for other characteristics it results in the same performance. Details of the evaluation can be found in [Shc+15b].

3.2 Parallelizing the Calculation of Diagnoses

In addition to improving the calculation of conflicts, the overall search for diagnoses can be enhanced as well. The HS-Tree algorithm only expands one node of the search tree at a time and only a single thread is used for the calculation. As modern computers, laptops, and even smartphones have multiple computation cores, the tree construction process can be parallelized by expanding multiple nodes of the search tree at the same time. Thereby, the full potential available in today's hardware architectures is utilized. In [Jan+16a; Jan+16b] different approaches to parallelize the HS-Tree algorithm were proposed. In this section two of these approaches are presented: *Level-Wise Parallelization* and *Full Parallelization*.

Level-Wise Parallelization: The original HS-Tree algorithm proposed in [Rei87] uses several tree pruning rules to reduce the search space (see Section 2.2 for an example). As these pruning rules require that the nodes of the search tree are expanded in the correct order, the parallelization of the HS-Tree algorithm is not trivial.

Therefore, the main idea of the first parallelization approach presented in this thesis, called Level-Wise Parallelization (LWP), is to mostly keep the order in which the nodes are expanded intact. To achieve this goal, all nodes on the same level are expanded in parallel and the algorithm continues with the next level once all nodes of the previous level are finished. An example of how LWP works is shown in Figure 3.2 and explained in the following.

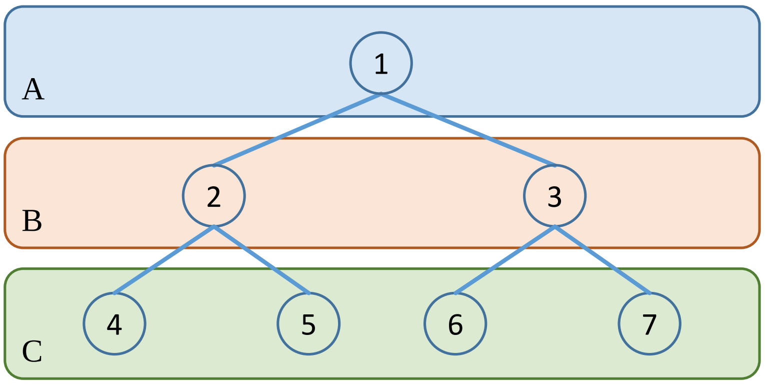


Figure 3.2: Exemplary schedule of the Level-Wise Parallelization technique with three scheduling steps A to C.

In the first step (A), only node ① can be processed, as no other nodes exist yet. When the first node is created, nodes ② and ③ are expanded in parallel (B) and the algorithm waits until the expansions of both nodes are finished. After both nodes are created, the algorithm continues with the third level (C) and processes all nodes of this level in parallel. As all nodes of the previous level were finished before the expansion of the new level began, all pruning rules of Reiter's HS-Tree algorithm [Rei87] can be applied. In addition, synchronization between threads is only required to ensure that no thread explores a path that is already being explored by another thread. The soundness and completeness of LWP is proven in [Jan+16b], which can be found in the appendix of this thesis.

The main advantage of the LWP approach is that it provides a way to parallelize the construction of nodes in the search tree while requiring only little synchronization to ensure the correctness of the tree pruning rules. However, if some node of a level needs more time to expand than the other nodes of the same level, it can happen that the algorithm has to wait for this single node before the expansion of the next level can start.

Full Parallelization: The main idea of the Full Parallelization (FP) approach is not to wait at the end of a level but to continue with the expansion of the nodes of the next level, even though the previous level has not been finished. An example of the parallel expansion progress is shown in Figure 3.3.

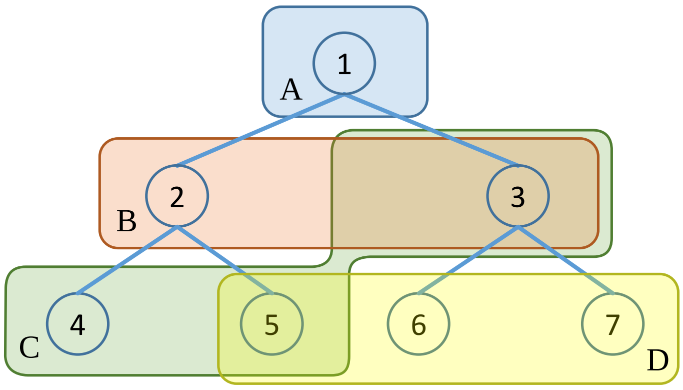


Figure 3.3: Exemplary schedule of the Full Parallelization technique with four scheduling steps A to D.

The FP algorithm always schedules all available nodes for parallel expansion and thus does not use discrete scheduling steps anymore that correspond to the levels of the search tree. In the example the algorithm expands nodes ② and ③ in parallel (B), after node ① is finished, as LWP does. After one of these nodes is finished, for example node ②, the algorithm immediately continues to expand the child nodes ④ and ⑤ of the finished parent (C) and does not wait at the end of the level like LWP. After node ③ is finished the algorithm can queue nodes ⑥ and ⑦ for expansion in addition to the nodes that are still being expanded (D).

It can happen that nodes of a previous level are still expanding when nodes on the next level are already finished. In some of these cases an already expanded node should be pruned according to the tree pruning rules. Therefore, after the expansion of every node the algorithm has to check if some of the other already created nodes should be removed again because of the newly obtained information. In [Jan+16b] the details of FP as well as a proof of its correctness are given.

In comparison to LWP, FP has the advantage that it does not have to wait for single nodes at the end of a level. However, FP has to perform some additional checks and an additional synchronization between the threads to ensure the correctness of the approach. In cases in which the last nodes of each level finish at the same time, LWP could therefore be faster than FP, because it has less overhead.

Evaluation: In [Jan+16b] LWP and FP were evaluated on different system types in comparison to the sequential HS-Tree algorithm. Table 3.2 summarizes the average reductions of the computation times that could be achieved when 4 threads were used for the parallelized algorithms.

Table 3.2: Average reductions of the computation times of LWP and FP using 4 threads compared to the sequential HS-Tree algorithm [Jan+16b].

System type	LWP	FP
Digital circuits	45%	65%
Constraint Satisfactions Problems	39%	40%
Spreadsheets	48%	50%
Ontologies	38%	36%
Simulation experiments	69%	70%

For the tested spreadsheets the required calculation times could be reduced by about 48% for LWP and 50% for FP on average. This means that the required calculation time was halved using the proposed parallelization techniques. Although these reductions are below the theoretical optimum of 75% when using 4 threads on a computer with 4 computation cores, the results are still encouraging as good speedups could be achieved by the proposed approaches, which utilize the full potential of the available hardware.

Sequential Diagnosis 4

Model-Based Diagnosis approaches determine all possible reasons of a discrepancy between the expected and the observed calculation outcomes of a spreadsheet. For large or complex spreadsheets and depending on the provided test cases, however, it can happen that too many diagnoses are returned by these techniques so that a user cannot inspect all of them manually.

To find the true reason of the discrepancy, called *preferred diagnosis*, one possible approach is to reduce the number of diagnoses by iteratively asking the user for new information. This technique is called *sequential diagnosis* and is depicted in Figure 4.1. The new information obtained through the queries can include new observations about correct or faulty values or state the correctness of some formulas. The statements are then added to the knowledge about the spreadsheet and with their information new diagnoses can be determined that are more precise than the previous ones.

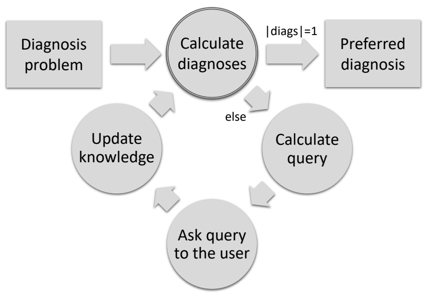


Figure 4.1: The sequential diagnosis approach [Shc+16c].

4.1 The General Sequential Diagnosis Approach

The general idea of using additional measurements to reduce the number of diagnoses was already proposed in the early works of MBD [Rei87; Kle+87]. De Kleer *et al.* additionally presented a method to determine the next best query to ask to the user [Kle+87]. In several later works including [Fel+10; Shc+12; Shc+16b] this method was used and improved. In this thesis, the sequential diagnosis approach is summarized based on the description in [Shc+16b], which can be found in the appendix. Although in this paper sequential diagnosis is not used for the spreadsheet setting, it can be easily applied to spreadsheets as shown in this section.

The goal of most sequential diagnosis approaches is to find the true reason of an observed fault with as few queries as possible. Since the system cannot predict how the user will answer a query, it tries to choose a query that will eliminate as many diagnoses as possible regardless of the user's answer. To do so, first, the system calculates a set of diagnoses with the currently available knowledge. Next, it splits the set of diagnoses into two sets that have the same value based on some criteria. The value of a set of diagnoses can, for example, be determined by using the number of formulas contained in these diagnoses or by using the probabilities of the individual formulas being faulty, if this information is available. Once such a partition is found, the system tries to find a query to discriminate between these two sets, i.e., a query for which one set of the diagnoses remains if the user answers "yes" and the other set remains if the user answers "no". If no such query can be found, the system tries the next best possible partition and continues until a partition is found for which a query exists.

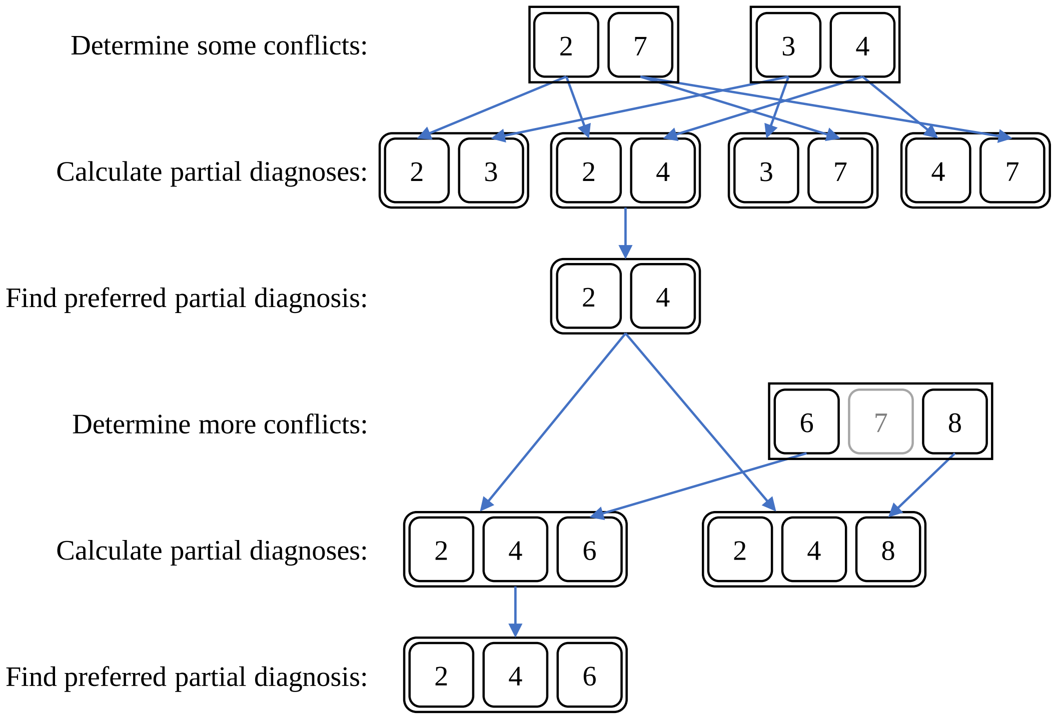
The calculated query is then presented to the user who has to evaluate and answer it. The information gained from the user's answer is added to the knowledge about the spreadsheet and the process is repeated until only a single diagnosis remains that is then known to be the true reason of the observed fault.

4.2 Speeding Up the Query Calculation

For large systems determining the next query can take too long in the interactive sequential diagnosis process. The reason is that a set of diagnoses is required to determine the next query. Although it was shown that a set of 9 diagnoses is sufficient to determine a good query [Shc+12], for larger systems calculating these 9 diagnoses can already exceed acceptable times.

Algorithmic approach: In [Shc+16b] a new algorithmic approach was presented to speed up the calculation of the diagnoses required to determine the next query. The approach builds upon the new concept of so-called *partial diagnoses*. These partial diagnoses are, as the name suggests, subsets of real diagnoses. The idea of using partial diagnoses is to search for conflicts only once during the HS-Tree construction, for example using MergeXplain (see Section 3.1), and to use the found conflicts to determine partial diagnoses without checking if they fully explain the observed fault. Since the found conflicts are a subset of all conflicts of the system, the partial diagnoses determined because of these conflicts will also be subsets of the (complete) diagnoses of the system. Therefore, queries that help to discriminate between the calculated partial diagnoses will also help to reduce the number of (complete) diagnoses.

If we would have, for example, a system with components 1 to 8 and conflicts $\{\{2,7\},\{3,4\},\{6,7,8\}\}$, as used in the example of Section 3.1, the (complete) diagnoses for this system would be $\{\{3,7\},\{4,7\},\{2,3,6\},\{2,3,8\},\{2,4,6\},\{2,4,8\}\}$. If we now assume that we only computed 2 of these 3 conflicts, for example, $\{\{2,7\},$ $\{3,4\}\}$, we could determine the partial diagnoses $\{\{2,3\},$ $\{2,4\},$ $\{3,7\},$ $\{4,7\}\}$. These partial diagnoses are all subsets of complete diagnoses. In fact, 2 of these partial diagnoses are even complete although only 2 of the 3 conflicts of the system were used to calculate them.



Determine some conflicts:
2
7
3
4
Calculate partial diagnoses:
2
3
2
4
3
7
4
7
Find preferred partial diagnosis:
2
4
Determine more conflicts:
6
7
8
Calculate partial diagnoses:
2
4
6
2
4
8
Find preferred partial diagnosis:
2
4
6

Figure 4.2: Example of the sequential diagnosis process using partial diagnoses.

The concept of partial diagnoses can be utilized in the sequential diagnosis process using the following technique [Shc+16b]. An example of the process is shown in Figure 4.2. First, the algorithm searches for a set of conflicts in the given faulty system using MERGEXPLAIN or some other conflict detection technique that is in the best case able to efficiently determine multiple conflicts and will find, for example, the conflicts $\{2,7\}$ and $\{3,4\}$. The found conflicts are then used to determine a limited number, for example, 9, of partial diagnoses. In the example of Figure 4.2, however, only 4 partial diagnoses can be calculated because of the found conflicts. The system uses these partial diagnoses to determine queries to ask to the user in the same way as the general sequential diagnosis approach does (see Section 4.1). The process of calculating the partial diagnoses, determining a query, and asking it to the user is repeated until only a single partial diagnosis can be found, for example, $\{2, 4\}$. This partial diagnosis is then called the *preferred partial diagnosis* and is known to be a subset of the true reason of the observed fault. The algorithm then continues to search for an additional set of conflicts with MERGEXPLAIN and repeats the process for these new conflicts. In the example, the new conflict $\{6, 7, 8\}$ is found. The component 7, however, was already excluded because of the previously asked questions and is thus ignored. Therefore, only 2 partial diagnoses can be calculated with the new conflict and the system asks another query to find the preferred partial diagnosis among them. Since no more conflicts can be found in the next step, the preferred partial diagnosis determined this way is known to be a complete diagnosis and the true reason of the fault. In [Shc+16b], which is included in this thesis, the details of this technique are described and its correctness is proven.

Evaluation: To evaluate the new approach it was compared to another technique that calculates diagnoses directly without using the concept of conflicts and was shown to be efficient in [Shc+12]. The average reductions in computation time, number of queries, and number of queried statements, which were asked in the queries, are shown in Table 4.1 for the two tested types of systems.

Table 4.1: Average reductions of the computation time, number of queries, and number of queried statements of the new approach presented in [Shc+16b] compared to the technique presented in [Shc+12]. Values in parentheses show the reductions for systems that require more than a second to compute.

System type	Time	#Queries	#Statements
Digital circuits	61% (81%)	30%	1%
Ontologies	83% (88%)	4%	5%

The results show that using partial diagnoses significantly reduces the time required to calculate the queries. This reduction in time is even bigger for those systems that require more than a second to compute (shown in parentheses in Table 4.1). For the most complex digital circuit, the technique proposed in [Shc+12] was not able to find the true reason of the fault after 24 hours while the new approach needed about 40 minutes. Regarding the number of required queries and queried statements in order to find the true reason of the fault, using partial diagnoses resulted in about the same numbers as the compared approach except for the number of queries for the digital circuits. For these systems the new approach was able to reduce the number of required queries by 30%. This means that using partial diagnoses does not lead to an increased amount of effort required by the user.

Creating a Corpus of Faulty Spreadsheets 5

Most of the approaches for fault detection in spreadsheets are evaluated on real-world spreadsheets in which the researchers inserted faults manually or based on randomly mutating the formulas [Jan+14a]. Although these evaluations are a good indicator to show that the tested approaches could theoretically help to locate faults in the spreadsheets, whether these approaches would work for spreadsheets with real faults cannot be evaluated with certainty based on these artificial faults.

To assess the quality of new approaches for fault detection in practice, spreadsheets are required that contain formula faults made by real users. An additional challenge is that although many real-world spreadsheets probably contain faults, it has to be known where these faults are in order to evaluate if the techniques for spreadsheet debugging are able to detect them. Therefore, we need to know which formulas are faulty and how they should be corrected.

5.1 Types of Spreadsheets Used in Research

In the research literature about fault detection in spreadsheets, three different types of spreadsheets with fault information are used to evaluate the efficiency or effectiveness of the approaches. Examples of these evaluations are given in [Jan+14a]. The different types of spreadsheets used in existing evaluations can be summarized as follows:

- **Artificial spreadsheets with artificial faults:** These spreadsheets were designed by the researchers in order to evaluate their new approach. Often, such spreadsheets are inspired by real-world spreadsheets, but are much simpler and did not evolve over time. In addition, as the faults were artificially inserted by the researchers, evaluations solely based on these spreadsheets can only serve as a first indicator for the quality of the approach.

- **Artificial spreadsheets with real faults:** Spreadsheets of this category are created in spreadsheet development experiments, see [Pan00] for examples.

In these experiments the participants have to develop a spreadsheet to fulfill a given task. After the experiment, the experimenters can then check the created spreadsheets for faults as the expected behavior of the spreadsheets is well defined. Although the faults found this way are real, the spreadsheets themselves are artificial because they were only created for the experiment and it is not known how well the specified task fits to the tasks encountered in practice.

- **Real-world spreadsheets with artificial faults:** Most of the approaches for fault detection in spreadsheets are evaluated on spreadsheets of this category. These spreadsheets were used in the industry to solve real tasks and are thus a good example of what kind of spreadsheets can be found in the real world. Although many of these spreadsheets probably contain faults, no information about the contained faults is available, as the semantics of a spreadsheet cannot be reconstructed with certainty. Therefore, researchers insert artificial faults in these spreadsheets in order to use them for their evaluations.

As all of these spreadsheet types are not sufficient to fully evaluate the functionality of new approaches in the real world, spreadsheets of the fourth possible type are desirable.

- **Real-world spreadsheets with real faults:** The ideal spreadsheets to be used in an evaluation of a new fault detection approach are real-world spreadsheets for which the information about the contained real faults is available, i.e., the spreadsheets have faults made by real users and it is known which formulas are faulty and what the correct formulas should be. Since the spreadsheets of this category have been used to solve real tasks and their faults were made by real users, they represent good examples of faults that should be detected by all testing and fault localization techniques.

5.2 Publicly Available Spreadsheet Corpora

Because companies usually do not publish their internal spreadsheets as they possibly contain confidential information, researchers have to use corpora of spreadsheets that are publicly available in order to evaluate new approaches. In this section, a list of publicly available spreadsheet corpora is given.

EUSES corpus: The most widely used corpus in fault detection research for spreadsheets is the EUSES corpus [Fis+05]. It was created to assist researchers in evaluating new spreadsheet QA approaches and contains 4,498 spreadsheets obtained by a

Google web search with different search terms related to business and education. The spreadsheets can be considered to be authentic although some of them might have been created for showcase purposes. The drawback of this corpus is that no information about the contained faults is available so that in order to use it for evaluations of fault detection techniques artificial faults have to be inserted.

Fuse corpus: Similar to the EUSES corpus, the Fuse corpus contains spreadsheets found through a web search. In their work [Bar+15], Barik *et al.* give an exact description of how the corpus can be obtained to ensure reproducibility and extensibility. The extensive web search led to a corpus of 249,376 spreadsheets.

Info1 corpus: The Info1 corpus was created during a spreadsheet development exercise and contains 119 faulty versions of 2 different spreadsheets. Since the intended semantics of the spreadsheets are known, the faults made by the participants could be identified and the information about the contained faults is included in the corpus. However, the spreadsheets of this corpus cannot be considered to reflect spreadsheets from the industry, because they were developed in an exercise. The corpus is described in [Get15] and can be obtained from [Inf].

Payroll/Gradebook corpus: This corpus originally consisted of spreadsheets developed in the academic Forms/3 spreadsheet environment. These artificial spreadsheets with injected faults were used in a user study in which 20 participants had to debug and test two different spreadsheets [Rut+06]. In addition to the information about the faults, the (possibly faulty) test cases created by the users are available. An MS Excel version of this corpus can be obtained from [Pgc].

Enron corpus: The Enron Corporation was one of the biggest companies in the US and one of the world's major electricity and gas companies. When it went bankrupt in 2001, a big accounting fraud was revealed, which is known as the Enron scandal. In the process of the investigations, all emails sent from or to Enron between 2000 and 2002 were published in 2003. In [Her+15] Hermans and Murphy-Hill extracted 15,770 spreadsheets contained in these emails and published them as the Enron corpus. Since all of these spreadsheets were sent in emails related to the business of Enron, they can be considered real-world spreadsheets. Again, no information about the contained faults is available.

Of all publicly available spreadsheet corpora, none contains both real-world spreadsheets and information about real faults.

5.3 Building a Real-World Spreadsheet Corpus with Fault Information

Although multiple spreadsheet corpora are available to evaluate new approaches in spreadsheet QA, there is still a need for a corpus that consists of real-world spreadsheets combined with information about the real faults that are contained in these spreadsheets.

In this thesis, a new method is presented to build such a corpus based on the available spreadsheets and emails of Enron. The spreadsheets of the Enron corpus were used in practice and as spreadsheets are error-prone, at least some of them will contain faults made by the users. Because the spreadsheets were sent as email attachments, the information of the spreadsheets can be combined with the information given in the emails. The following aspects can be used to detect real faults in the spreadsheets of the Enron corpus:

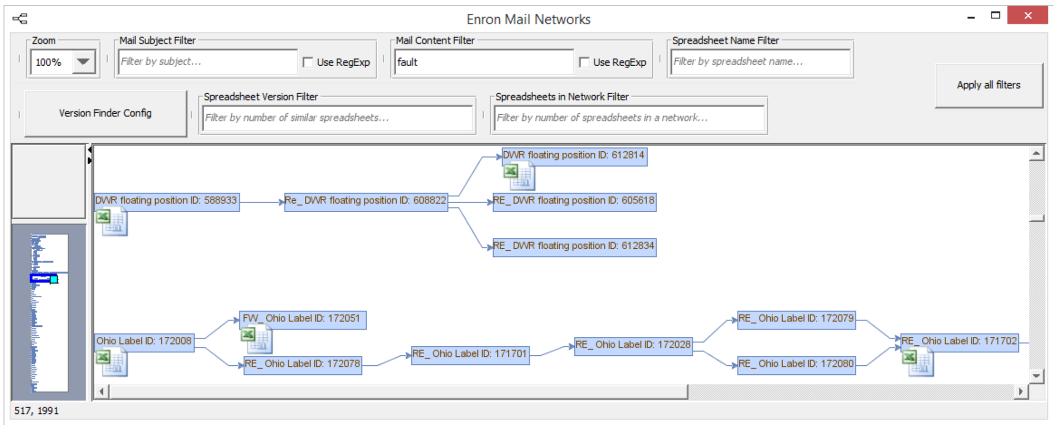
- In the emails to which the spreadsheets are attached, the text message can include some descriptions about the spreadsheets. In these descriptions faults in the spreadsheets can be mentioned that, for example, were detected or fixed.

- In many cases, multiple versions of the same spreadsheet were sent over time that only differ in a few cells. If from one version to another only a single or a few formulas have been changed and the rest of the spreadsheet was kept unchanged, these changes could be the result of a fault correction by the user.

5.3.1 Fault Detection Methods

In [Sch+16a], two techniques are presented to help a researcher detect faults in the spreadsheets of the Enron corpus. The techniques were designed to combine the information given in the emails and in the spreadsheets themselves. However, the approach is not limited to the emails of the Enron corpus and can be applied to any corpus of emails containing spreadsheets, because no domain-specific knowledge is required.

Reconstruction of email conversations: A description of a fault that is found in a spreadsheet could possibly be included in the answer to the email that the spreadsheet was attached to. To utilize this information, a tool was developed that automatically reconstructs the email conversations, as shown in Figure 5.1.


Enron Mail Networks
Mail Subject Filter
Filter by subject...
Use RegExp
Use RegExp
Spreadsheet Name Filter
Filter by spreadsheet name...
Apply all filters
Version Finder Config
Spreadsheet Version Filter
Filter by number of similar spreadsheets...
Spreadsheets in Network Filter
Filter by number of spreadsheets in a network...
DWR floating position ID: 612814
DWR floating position ID: 588933
Re_ DWR floating position ID: 608822
RE_ DWR floating position ID: 605618
RE_ DWR floating position ID: 612834
FW_ Ohio Label ID: 172051
Ohio Label ID: 172008
RE_ Ohio Label ID: 172078
RE_ Ohio Label ID: 171701
RE_ Ohio Label ID: 172028
RE_ Ohio Label ID: 172079
RE_ Ohio Label ID: 172080
RE_ Ohio Label ID: 171702
517, 1991

Zoom 100%
Mail Content Filter fault

Figure 5.1: Example of reconstructed email conversations [Sch+16a]. The spreadsheet icons denote that spreadsheets are attached to the emails.

The conversations can be searched for keywords related to errors. The researcher can then read these conversations in the order in which they were sent. If the message text in an email mentions a corrected or a found fault in some spreadsheet, the researcher can explicitly search for this fault in the attached spreadsheet. To inspect a suspicious spreadsheet attached to an email he or she can click on the spreadsheet icon to open it. The visualization of the conversations helps the researcher to quickly get an overview of the different conversations and to understand the relationships between the emails.

To reconstruct the email conversations, for each email of the corpus the previous and following messages of the same conversation have to be found. However, the Enron corpus does not contain any explicit information for the emails that allows a precise reconstruction of these conversations. Therefore, the system uses a set of heuristics based on the subject, sender, recipients, time stamp, and the message text of the emails to do an approximate reconstruction. A detailed description of the used heuristics can be found in [Sch+16a], which is included in this thesis by publication.

Analyzing the differences in spreadsheets: If only a single or a few formula cells in a spreadsheet were changed, these changes could possibly represent a correction of a fault. Whether such a difference really represents a correction of a fault or a change of the modeled business logic can only be decided by a spreadsheet expert who manually inspects the changes. A tool can, however, support the expert in his or her task by listing a set of candidate spreadsheets of which only a few formulas were changed and by visualizing these changes.

In [Sch+16a], a systematic approach is presented to detect the changes made from one spreadsheet version to another. Searching for the differences between two spreadsheets in a meaningful way is not trivial. The system has to detect inserted or deleted rows and columns because otherwise every single cell after such a row or column would be perceived as a difference. It also has to report the same change to multiple equivalent formulas as only a single difference because otherwise such a change would result in multiple differences. This has to be avoided since spreadsheet versions that contain too many differences are not considered to contain a correction of a fault. Details of this approach are given in [Sch+16a].

5.3.2 The Enron Error Corpus

With the help of the presented approaches a first initial inspection of the email conversations and the fault correction candidates was done, which led to a corpus of 30 spreadsheets containing 36 real faults. For most of the faults the corpus contains a faulty and a corrected version of the spreadsheet. This can be useful to evaluate approaches that propose to make suggestions how faulty formulas should be repaired. The Enron Error Corpus can be found at [Sch+16c].

Table 5.1: Overview of the Enron Error Corpus [Sch+16c].

Error type	Nb of errors
Qualitative	8
Quantitative	28
Mechanical	14
Logic	9
Omission	5
Total	36

An overview of the detected faults is given in Table 5.1. The corpus contains 8 qualitative faults, which did not result in a faulty value of the current spreadsheet but could do so in a later version of the spreadsheet. Since the main goal was to search for faults that result in wrong output values the majority of the found faults are quantitative.

The corpus was published in order to support researchers in evaluating their approaches on real-world spreadsheets with real faults and we plan to use it for our future evaluations as well. In addition, the tool was published to allow other researchers to search for faults in the Enron spreadsheets.

6 Conclusion

Spreadsheets are widely used in the industry for day-to-day business activities and to support strategic business decisions. Since spreadsheets, like any other software, can be faulty and since these faults often remain undetected, they have led to severe consequences, for example, loosing money or worse. Therefore, better tool support is required to support the users in detecting and correcting these faults.

In this thesis by publication an overview of the different domains in automated spreadsheet quality assurance was given and different new algorithmic approaches were presented that help users to detect faults in spreadsheet formulas. In the evaluations it was shown that all of these approaches are beneficial in comparison to previous state-of-the-art techniques. Since the presented approaches can be used in combination with each other, they can be utilized to efficiently find the true reason of a detected miscalculation.

In addition to the various algorithmic enhancements mentioned in the appended papers, one open question in the research field is whether extensive tool support for spreadsheet quality assurance will be accepted by the spreadsheet users in the industry. As some of the biggest benefits of the spreadsheets are their flexibility and the fast development times, it is important that approaches for spreadsheet QA do not reduce these benefits.

Therefore, one important future topic of investigation should be to check if real users in the industry accept the proposed quality assurance techniques. To evaluate this aspect, field studies with real users who test the different approaches are required. The studies should show (a) if users are willing to use the tools in their daily business and (b) if the approaches can help to enhance the quality of the spreadsheets.

Another mostly open question is how the awareness of spreadsheet users for the risks caused by faulty spreadsheets can be raised. Since a common mistake in the industry is to underestimate these high risks [Pan98; Pan+12], raising the risk awareness would help to motivate users to test their spreadsheets and the detected faults could then, for example, be located by the approaches presented in this thesis.

Bibliography

[Abe15] Stephan Abel. “Automatische Erkennung von Spreadsheetversionen”. Bachelor’s thesis. TU Dortmund, 2015 (cit. on p. 2).

[Bar+15] Titus Barik, Kevin Lubick, Justin Smith, John Slankas, and Emerson Murphy-Hill. “Fuse: A Reproducible, Extendable, Internet-Scale Corpus of Spreadsheets”. In: *Proceedings of the IEEE/ACM 12th Working Conference on Mining Software Repositories*. 2015, pp. 486–489 (cit. on p. 29).

[Fel+10] Alexander Feldman, Gregory Provan, and Arjan Van Gemund. “A Model-Based Active Testing Approach to Sequential Diagnosis”. In: *Journal of Artificial Intelligence Research* 39 (2010), p. 301 (cit. on p. 23).

[Fis+05] Marc Fisher and Gregg Rothermel. “The EUSES Spreadsheet Corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms”. In: *SIGSOFT Software Engineering Notes* 30.4 (2005), pp. 1–5 (cit. on p. 28).

[Get15] Elisabeth Getzner. “Improvements for Spectrum-based Fault Localization in Spreadsheets”. Master’s thesis. Graz University of Technology, May 2015 (cit. on p. 29).

[Gre+89] Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. “A Correction to the Algorithm in Reiter’s Theory of Diagnosis”. In: *Artificial Intelligence* 41.1 (1989), pp. 79–88 (cit. on p. 14).

[Her+13] Thomas Herndon, Michael Ash, and Robert Pollin. *Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff*. Working Paper 322, Political Economy Research Institute, University of Massachusetts, Amherst. 2013 (cit. on p. 2).

[Her+15] Felienne Hermans and Emerson Murphy-Hill. “Enron’s Spreadsheets and Related Emails: A Dataset and Analysis”. In: *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*. 2015, pp. 7–16 (cit. on pp. 1, 29).

[Hof+14] Birgit Hofer, Dietmar Jannach, Thomas Schmitz, Kostyantyn Shchekotykhin, and Franz Wotawa. “Tool-supported fault localization in spreadsheets: Limitations of current research practice”. In: *Proceedings of the 1st International Workshop on Software Engineering Methods in Spreadsheets (SEMS 2014)*. 2014 (cit. on p. 44).

[Hun+05] Christopher D. Hundhausen and Jonathan Lee Brown. “What you see is what you code: a radically dynamic algorithm visualization development model for novice learners”. In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2005)*. 2005, pp. 163–170 (cit. on p. 1).

[Jan+13] Dietmar Jannach, Arash Baharloo, and David Williamson. “Toward an integrated framework for declarative and interactive spreadsheet debugging”. In: *Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2013)*. 2013, pp. 117–124 (cit. on pp. 7, 16).

[Jan+14a] Dietmar Jannach, Thomas Schmitz, Birgit Hofer, and Franz Wotawa. “Avoiding, Finding and Fixing Spreadsheet Errors - A Survey of Automated Approaches for Spreadsheet QA”. In: *Journal of Systems and Software* 94 (2014), pp. 129–150 (cit. on pp. 1, 3, 4, 7, 27, 43).

[Jan+14b] Dietmar Jannach, Thomas Schmitz, and Kostyantyn Shchekotykhin. “Parallelized Hitting Set Computation for Model-Based Diagnosis”. In: *Proceedings of the 25th Workshop on Principles of Diagnosis (DX 2014)*. 2014 (cit. on p. 44).

[Jan+14c] Dietmar Jannach, Thomas Schmitz, and Kostyantyn Shchekotykhin. “Toward Interactive Spreadsheet Debugging”. In: *Proceedings of the 1st International Workshop on Software Engineering methods in Spreadsheets (SEMS 2014)*. 2014 (cit. on p. 44).

[Jan+15a] Dietmar Jannach, Thomas Schmitz, and Kostyantyn Shchekotykhin. “Parallelized Hitting Set Computation for Model-Based Diagnosis”. In: *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI 2015)*. 2015, pp. 1503–1510 (cit. on p. 44).

[Jan+15b] Dietmar Jannach and Thomas Schmitz. “Using Calculation Fragments for Spreadsheet Testing and Debugging”. In: *Proceedings of the 2nd International Workshop on Software Engineering Methods in Spreadsheets at ICSE 2015 (SEMS 2015)*. 2015 (cit. on p. 44).

[Jan+16a] Dietmar Jannach and Thomas Schmitz. “Model-Based Diagnosis of Spreadsheet Programs: A Constraint-based Debugging Approach”. In: *Automated Software Engineering* 23.1 (2016), pp. 105–144 (cit. on pp. 5, 7, 11, 15, 16, 20, 43).

[Jan+16b] Dietmar Jannach, Thomas Schmitz, and Kostyantyn Shchekotykhin. “Parallel Model-Based Diagnosis On Multi-Core Computers”. In: *Journal of Artificial Intelligence Research* 55 (2016), pp. 835–887 (cit. on pp. 6, 8, 20–22, 43).

[Jun04] Ulrich Junker. “QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems”. In: *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI 2004)*. 2004, pp. 167–172 (cit. on pp. 14, 17).

[Kle+87] Johan de Kleer and Brian C. Williams. “Diagnosing Multiple Faults”. In: *Artificial Intelligence* 32.1 (1987), pp. 97–130 (cit. on pp. 11, 23).

[Pan+10] Raymond R. Panko and Salvatore Aurigemma. “Revising the Panko-Halverson taxonomy of spreadsheet errors”. In: *Decision Support Systems* 49.2 (2010), pp. 235–244 (cit. on p. 2).

[Pan+12] Raymond R. Panko and Daniel N. Port. "End User Computing: The Dark Matter (and Dark Energy) of Corporate IT". In: *Proceedings of the 45th Hawaii International Conference on System Sciences (HICSS 2012)*. 2012, pp. 4603–4612 (cit. on pp. 1, 33).

[Pan00] Raymond R. Panko. "Spreadsheet Errors: What We Know. What We Think We Can Do." In: *Proceedings of the European Spreadsheet Risks Interest Group 1st Annual Conference (EuSpRIG 2000)*. 2000 (cit. on p. 27).

[Pan98] Raymond R. Panko. "What We Know About Spreadsheet Errors". In: *Journal of End User Computing* 10.2 (1998), pp. 15–21 (cit. on pp. 1, 2, 33).

[Pow+08] Stephen G. Powell, Kenneth R. Baker, and Barry Lawson. "A critical review of the literature on spreadsheet errors". In: *Decision Support Systems* 46.1 (2008), pp. 128–138 (cit. on p. 2).

[Pur+06] Michael Purser and David Chadwick. "Does an awareness of differing types of spreadsheet errors aid end-users in identifying spreadsheets errors?" In: *Proceedings of the European Spreadsheet Risks Interest Group 7th Annual Conference (EuSpRIG 2006)*. 2006, pp. 185–204 (cit. on p. 2).

[Rei+10] Carmen M. Reinhart and Kenneth S. Rogoff. "Growth in a Time of Debt". In: *American Economic Review* 100.2 (2010), pp. 573–578 (cit. on p. 2).

[Rei87] Raymond Reiter. "A Theory of Diagnosis from First Principles". In: *Artificial Intelligence* 32.1 (1987), pp. 57–95 (cit. on pp. 11, 13, 20, 21, 23).

[Rut+06] Joseph R. Ruthruff, Margaret Burnett, and Gregg Rothermel. "Interactive Fault Localization Techniques in a Spreadsheet Environment". In: *IEEE Transactions on Software Engineering* 32.4 (2006), pp. 213–239 (cit. on p. 29).

[Sca+05] Christopher Scaffidi, Mary Shaw, and Brad Myers. "Estimating the Numbers of End Users and End User Programmers". In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2005)*. 2005, pp. 207–214 (cit. on p. 1).

[Sch+16a] Thomas Schmitz and Dietmar Jannach. "Finding Errors in the Enron Spreadsheet Corpus". In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2016)*. 2016, pp. 157–161 (cit. on pp. 7, 9, 30–32, 44).

[Sch+16b] Thomas Schmitz, Birgit Hofer, Dietmar Jannach, and Franz Wotawa. "Fragment-Based Diagnosis of Spreadsheets". In: *Proceedings of the 3rd International Workshop on Software Engineering Methods in Spreadsheets (SEMS 2016)*. 2016 (cit. on p. 44).

[Shc+12] Kostyantyn Shchekotykhin, Gerhard Friedrich, Philipp Fleiss, and Patrick Rodler. "Interactive ontology debugging: Two query strategies for efficient fault localization". In: *Journal of Web Semantics* 12-13 (2012), pp. 788–103 (cit. on pp. 23, 24, 26).

[Shc+15a] Kostyantyn Shchekotykhin, Dietmar Jannach, and Thomas Schmitz. "A Divide-And-Conquer Method for Computing Multiple Conflicts for Diagnosis". In: *Proceedings of the 26th Workshop on Principles of Diagnosis (DX 2015)*. 2015 (cit. on p. 44).

[Shc+15b] Kostyantyn Shchekotykhin, Dietmar Jannach, and Thomas Schmitz. “MergeXplain: Fast Computation of Multiple Conflicts for Diagnosis”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2015)*. 2015, pp. 3221–3228 (cit. on pp. 6, 8, 14, 17, 19, 43).

[Shc+16a] Kostyantyn Shchekotykhin, Thomas Schmitz, and Dietmar Jannach. “Efficient Determination of Measurement Points for Sequential Diagnosis”. In: *Proceedings of the Joint German/Austrian Conference on Artificial Intelligence (KI 2016)*. 2016 (cit. on p. 44).

[Shc+16b] Kostyantyn Shchekotykhin, Thomas Schmitz, and Dietmar Jannach. “Efficient Sequential Model-Based Fault-Localization with Partial Diagnoses”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2016)*. 2016, pp. 1251–1257 (cit. on pp. 6, 8, 23–26, 43).

[Shc+16c] Kostyantyn Shchekotykhin, Thomas Schmitz, and Dietmar Jannach. “Using Partial Diagnoses for Sequential Model-Based Fault Localization”. In: *Proceedings of the 27th International Workshop on Principles of Diagnosis (DX 2016)*. 2016 (cit. on pp. 23, 45).

[IEE10] IEEE Computer Society. “IEEE Standard Classification for Software Anomalies”. In: *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)* (2010), pp. 1–23 (cit. on p. 1).

Web pages

[F1F] F1F9. *The Dirty Dozen*. URL: `http://blogs.mazars.com/the-model-auditor/files/2014/01/12-Modelling-Horror-Stories-and-Spreadsheet-Disasters-Mazars-UK.pdf` (visited on Apr. 3, 2017) (cit. on p. 2).

[Inf] *Info1 corpus*. May 2015. URL: `http://spreadsheets.ist.tugraz.at/index.php/corpora-for-benchmarking/info1/` (visited on Apr. 3, 2017) (cit. on p. 29).

[Pgc] *Payroll/Gradebook corpus*. 2006. URL: `http://spreadsheets.ist.tugraz.at/index.php/corpora-for-benchmarking/payrollgradebook-2/` (visited on Apr. 3, 2017) (cit. on p. 29).

[Sch+16c] Thomas Schmitz and Dietmar Jannach. *The Enron Error Corpus*. 2016. URL: `http://ls13-www.cs.tu-dortmund.de/homepage/spreadsheets/enron-errors.htm` (visited on Apr. 3, 2017) (cit. on p. 32).

[Tan14] Gillian Tan. *Spreadsheet Mistake Costs Tibco Shareholders $100 Million*. 2014. URL: `http://on.wsj.com/1vjYdWE` (visited on Apr. 3, 2017) (cit. on p. 2).

List of Figures

List of Tables

Publications

In this thesis by publication the following six works of the author are included. These publications are closely related to Model-Based Debugging of spreadsheets. The full texts of these works can be found after this list.

- Dietmar Jannach, Thomas Schmitz, Birgit Hofer, and Franz Wotawa. “Avoiding, Finding and Fixing Spreadsheet Errors - A Survey of Automated Approaches for Spreadsheet QA”. in: *Journal of Systems and Software* 94 (2014), pp. 129–150

- Dietmar Jannach and Thomas Schmitz. “Model-Based Diagnosis of Spreadsheet Programs: A Constraint-based Debugging Approach”. In: *Automated Software Engineering* 23.1 (2016), pp. 105–144

- Kostyantyn Shchekotykhin, Dietmar Jannach, and Thomas Schmitz. “MergeXplain: Fast Computation of Multiple Conflicts for Diagnosis”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2015)*. 2015, pp. 3221–3228

- Dietmar Jannach, Thomas Schmitz, and Kostyantyn Shchekotykhin. “Parallel Model-Based Diagnosis On Multi-Core Computers”. In: *Journal of Artificial Intelligence Research* 55 (2016), pp. 835–887

- Kostyantyn Shchekotykhin, Thomas Schmitz, and Dietmar Jannach. “Efficient Sequential Model-Based Fault-Localization with Partial Diagnoses”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2016)*. 2016, pp. 1251–1257

- Thomas Schmitz and Dietmar Jannach. “Finding Errors in the Enron Spreadsheet Corpus”. In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2016)*. 2016, pp. 157–161

In addition to these six main publications, the author of this thesis worked on the following other publications related to spreadsheet debugging that are not part of this thesis.

- Dietmar Jannach, Thomas Schmitz, and Kostyantyn Shchekotykhin. “Toward Interactive Spreadsheet Debugging”. In: *Proceedings of the 1st International Workshop on Software Engineering methods in Spreadsheets (SEMS 2014)*. 2014

- Birgit Hofer, Dietmar Jannach, Thomas Schmitz, Kostyantyn Shchekotykhin, and Franz Wotawa. “Tool-supported fault localization in spreadsheets: Limitations of current research practice”. In: *Proceedings of the 1st International Workshop on Software Engineering Methods in Spreadsheets (SEMS 2014)*. 2014

- Dietmar Jannach, Thomas Schmitz, and Kostyantyn Shchekotykhin. “Parallelized Hitting Set Computation for Model-Based Diagnosis”. In: *Proceedings of the 25th Workshop on Principles of Diagnosis (DX 2014)*. 2014

- Dietmar Jannach, Thomas Schmitz, and Kostyantyn Shchekotykhin. “Parallelized Hitting Set Computation for Model-Based Diagnosis”. In: *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI 2015)*. 2015, pp. 1503–1510

- Dietmar Jannach and Thomas Schmitz. “Using Calculation Fragments for Spreadsheet Testing and Debugging”. In: *Proceedings of the 2nd International Workshop on Software Engineering Methods in Spreadsheets at ICSE 2015 (SEMS 2015)*. 2015

- Kostyantyn Shchekotykhin, Dietmar Jannach, and Thomas Schmitz. “A Divide-And-Conquer Method for Computing Multiple Conflicts for Diagnosis”. In: *Proceedings of the 26th Workshop on Principles of Diagnosis (DX 2015)*. 2015

- Thomas Schmitz, Birgit Hofer, Dietmar Jannach, and Franz Wotawa. “Fragment-Based Diagnosis of Spreadsheets”. In: *Proceedings of the 3rd International Workshop on Software Engineering Methods in Spreadsheets (SEMS 2016)*. 2016

- Kostyantyn Shchekotykhin, Thomas Schmitz, and Dietmar Jannach. “Efficient Determination of Measurement Points for Sequential Diagnosis”. In: *Proceed-*

ings of the Joint German/Austrian Conference on Artificial Intelligence (KI 2016). 2016

- Kostyantyn Shchekotykhin, Thomas Schmitz, and Dietmar Jannach. “Using Partial Diagnoses for Sequential Model-Based Fault Localization”. In: *Proceedings of the 27th International Workshop on Principles of Diagnosis (DX 2016)*. 2016

Avoiding, Finding and Fixing Spreadsheet Errors - A Survey of Automated Approaches for Spreadsheet QA

Dietmar Jannach[1a], Thomas Schmitz[a], Birgit Hofer[b], Franz Wotawa[b]

[a] *TU Dortmund, Germany*
[b] *TU Graz, Austria*

Abstract

Spreadsheet programs can be found everywhere in organizations and they are used for a variety of purposes, including financial calculations, planning, data aggregation and decision making tasks. A number of research surveys have however shown that such programs are particularly prone to errors. Some reasons for the error-proneness of spreadsheets are that spreadsheets are developed by end users and that standard software quality assurance processes are mostly not applied. Correspondingly, during the last two decades, researchers have proposed a number of techniques and automated tools aimed at supporting the end user in the development of error-free spreadsheets. In this paper, we provide a review of the research literature and develop a classification of automated spreadsheet quality assurance (QA) approaches, which range from spreadsheet visualization, static analysis and quality reports, over testing and support to model-based spreadsheet development. Based on this review, we outline possible opportunities for future work in the area of automated spreadsheet QA.

Keywords: Spreadsheet, Quality Assurance, Testing, Debugging

1. Introduction

Spreadsheet applications, based, e.g., on the widespread Microsoft Excel software tool, can nowadays be found almost everywhere and at all levels of

[1]Corresponding author: D. Jannach (dietmar.jannach@tu-dortmund.de), Postal address: TU Dortmund, 44221 Dortmund, Germany, T: +49 231 755 7272

organizations [1]. These interactive computer applications are often developed by non-programmers – that is, domain or subject matter experts – for a number of different purposes including financial calculations, planning and forecasting, or various other data aggregation and decision making tasks.

Spreadsheet systems became popular during the 1980s and represent the most successful example of the End-User Programming paradigm. Their main advantage can be seen in the fact that they allow domain experts to build their own supporting software tools, which directly encode their domain expertise. Such tools are usually faster available than other business applications, which have to be developed or obtained via corporate IT departments and are subject to a company's standard quality assurance (QA) processes.

Very soon, however, it became obvious that spreadsheets – like any other type of software – are prone to errors, see, e.g., the early paper by Creeth [2] or the report by Ditlea [3], which were published in 1985 and 1987, respectively. More recent surveys on error rates report that in many studies on spreadsheet errors at least one fault was found in every single spreadsheet that was analyzed [4]. Since in reality even high-impact business decisions are made, which are at least partially based on faulty spreadsheets, such errors can represent a considerable risk to an organization[2].

Overall, empowering end users to build their own tools has some advantages, e.g., with respect to flexibility, but also introduces additional risks, which is why Panko and Port call them both "dark matter (and energy) of corporate IT" [1]. In order to minimize these risks, researchers in different disciplines have proposed a number of approaches to avoid, detect or fix errors in spreadsheet applications. In principle, several approaches are possible to achieve this goal, beginning with better education and training of the users, over organizational and process-related measures such as mandatory reviews and audits, to better tool support for the user during the spreadsheet development process. In this paper, we focus on this last type of approaches, in which the spreadsheet developer is provided with additional software tools and mechanisms during the development process. Such tools can for example help the developer locate potential faults more effectively, organize the

[2]See `http://www.eusprig.org/horror-stories.htm` for a list of real-world stories or the recent article by Herndorn et al. [5] who found critical spreadsheet formula errors in the often-cited economic analysis of Reinhart and Rogoff [6].

test process in a better structured way, or guide the developer to better spreadsheet designs in order to avoid faults in the first place. The goals and contributions of this work are (A) an in-depth review of existing works and the state-of-the-art in the field, (B) a classification framework for approaches to what we term "automated spreadsheet QA", and (C) a corresponding discussion of the limitations of existing works and an outline of perspectives for future work in this area.

This paper is organized as follows. In Section 2, we will define the scope of our research, introduce the relevant terminology and discuss the specifics of typical spreadsheet development processes. Section 3 contains our classification scheme for approaches to automated spreadsheet QA. In the Sections 4 to 9, we will discuss the main ideas of typical works in each category and we will report how the individual proposals were evaluated. Section 10 reviews the current research practices with respect to evaluation aspects. In Section 11, we point out perspectives for future works and Section 12 summarizes this paper.

2. Preliminaries

Before discussing the proposed classification scheme in detail, we will first define the scope of our analysis and sketch our research method. In addition, we will briefly discuss differences and challenges of spreadsheet QA approaches in comparison with tool-supported QA approaches for traditional imperative programs.

2.1. Scope of the analysis, research method, terminology

Spreadsheets are a subject of research in different disciplines including the fields of Information Systems (IS) and Computer Science (CS) but also fields such as Management Accounting or Risk Management, e.g., [2] or [7].

Scope. In our work, we adopt a Computer Science and Software Engineering perspective, focus on tool support for the spreadsheet development process and develop a classification of automated spreadsheet QA approaches. Examples for such tools could be those that help the user locate faults, e.g., based on visualization techniques or by directly pointing them to faulty cells, or tools that help the user avoid making faults in the first place, e.g., by supporting complex refactoring work. Spreadsheet error reduction techniques from the IS field, see, e.g., [8], and approaches that are mainly based on "manual" tasks like auditing or code inspection will thus not be in the focus of our work.

Research on spreadsheets for example in the field of Information Systems often covers additional, more user-related, or fundamental aspects such as error types, error rates and human error research in general, the user interface, cognitive effort and acceptance issues of tools, user over-confidence, as well as methodological questions regarding the empirical evaluation of systems, see, e.g., [9, 10, 11, 12, 13, 4]. Obviously, these aspects and considerations should be the basis when designing an automated spreadsheet QA tool that should be usable and acceptable by end users. In our work and classification, we however concentrate more on the provided functionality and the algorithmic approaches behind the various tools. We will therefore discuss the underlying assumptions for each approach, e.g., with respect to user acceptance or evaluation, only as they are reported in the original papers. Still, in order to assess the overall level of research rigor in the field, we will report for each class of approaches how the individual proposals were evaluated or validated.

These insights will be summarized and reviewed in Section 10. In this section, we will also look at the difficulties of empirically evaluating the true value of spreadsheet error reduction techniques according to the literature from the IS field.

Regarding tool support in commercial spreadsheet environments, we will briefly discuss the existing functionality of MS Excel and comparable systems in the different sections. Specialized commercial auditing add-ons to MS Excel usually include a number of QA tools. As our work focuses more on advanced algorithmic approaches to spreadsheet QA, we see the detailed analysis of current commercial tools to be beyond the scope of this paper. Finally, we will also not cover fault localization or avoidance techniques for the imperative programming extensions that are typically part of modern spreadsheet environments.

Research method. For creating our survey, we conducted an extensive literature research. Papers about spreadsheets are published in a variety of journals and conference proceedings. However, there exists no publication outlet which is only concerned with spreadsheets, except maybe for the application-oriented EuSpRIG conference series[3]. In our research, we therefore followed an approach which consists both of a manual inspection of relevant journals

[3]`http://www.eusprig.org`

and conference proceedings as well as searches in the digital libraries of ACM and IEEE. Typical outlets for papers on spreadsheets which were inspected manually included both broad Software Engineering conferences and journals such as ICSE, ACM TOSEM, or IEEE TSE. At the same time, we reviewed publications at more focused events such as ICSM or the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). In addition, major IS journals and events such as Information Systems Research, ACM TOIS or ICIS were considered in our research.

When searching the digital libraries, we started by looking for papers containing the term "spreadsheet" in the title or abstract. From the 400 to 500 results returned by the search engines of the libraries, we manually inspected the abstracts. Provided their scope was relevant for our research, we categorized them according to the categorization framework described in Section 3, and followed the relevant references mentioned in the articles.

Terminology. Regarding the terminology used in the paper, we will use the terms "spreadsheet", "spreadsheet application", or "spreadsheet program" more or less interchangeably as often done in the literature. When we refer to the underlying software system to create spreadsheets (e.g., Microsoft Excel), we will use the term "spreadsheet environment" or "spreadsheet tool". In some papers, the term "form-based visual languages" is used [14] to describe the more general family of such systems. In our work, we will however rely on the more widespread term "spreadsheet".

There are a number of definitions of the terms "error", "fault", and "failure" in the literature. According to IEEE standards for Software Engineering an "error" is a misapprehension on side of the one developing a spreadsheet caused by a mistake or misconception occurring in the human thought process. A "fault" is the manifestation of an "error" within a spreadsheet which may be causing a "failure". A "failure" is the deviation of the observed behavior of the spreadsheet from the expectations. In the literature on spreadsheets, in particular the terms "fault" and "error" are often used in an interchangeable manner. Surveys and taxonomies of spreadsheet problems like [15], [16], or [17], for example, more or less only use the term "error". In our review, we will – in order to be in line with general Software Engineering research – use the term "fault" instead of "error" whenever appropriate.

2.2. Specifics of spreadsheets and their QA tools

The requirements for automated approaches for spreadsheet QA can be quite different from those of tools that are used with typical imperative languages. In [14], Rothermel et al. illustrated some of the major differences in the context of spreadsheet testing. Many of the aspects mentioned in their work however do not only hold for the testing domain, but should in general be taken into account when developing tools supporting the spreadsheet developer.

First, the way in which users interact with a spreadsheet environment is largely different from how programs in imperative languages are developed. In spreadsheets, for example, the user is often constructing a spreadsheet in an unstructured incremental process using some test data. For the given test data, the user continuously receives visual and immediate feedback. To increase the chances of being accepted by developers, any supporting tool should therefore be designed in a way that it supports such an incremental development process. In that context, trying to de-couple the actual implementation tasks from other tasks like testing or design could be problematic. At the same time, being able to provide immediate feedback in the incremental process appears to be crucial.

Second, the computation paradigm of spreadsheets is quite different from imperative programs. The basic nature of spreadsheets is that their computations – the “evaluation” of the program – are driven by data dependencies between cells and explicit control flow statements are only contained in formulas in the cells. When designing supporting QA mechanisms and tools, this aspect should be kept in mind. For example, when adapting existing QA approaches from imperative programs to spreadsheet development, there might be different characteristics and quality measures that have to be considered. Data dependencies can, for example, be more relevant than the control flow. At the same time, the conceptual model of the users might be rather based on the data and formula dependencies than on execution orders.

Third, spreadsheet programs are not only based on a simpler computational model than imperative programs, their “physical” layout – i.e., the spatial arrangement of the labels and formulas – is typically strongly determined by the intended computation semantics. This spatial information can be used by automatic QA tools, e.g., to detect inconsistencies between neighboring cells and to assess the probability of a formula being semantically correct, to automatically infer label information, or to rank change suggestions in goal-directed debugging approaches [18].

Finally, developers of spreadsheets are mostly non-professional programmers. Developers of imperative programs often have a formal training or education in software development and are generally aware of the importance of systematic QA processes. People developing spreadsheets are mostly non-programmers and may have limited interest and awareness when it comes to investing additional efforts in QA activities like testing or refactoring. Therefore, any QA methodology and the corresponding tool support should make it easy for a non-programmer to understand the value of investing the additional efforts. To cope with this, approaches for spreadsheet QA should not require special training or an understanding of the theory behind the approach. The used language should avoid special terminology from the underlying theory or technique. When discussing the different approaches in the next sections, we will therefore briefly discuss the approaches from the perspective of usability and what is expected from the end user.

3. A classification of automated approaches to spreadsheet QA

Generally, we classify the various spreadsheet QA approaches into two main categories depending on their role and use in the development lifecycle.

- "Finding and fixing errors" is about techniques and tools that are mainly designed to help the user detect errors and understand the reasons for the errors. These tools are typically used by the developer or another person, e.g., an auditor or reviewer, during or after the construction of the spreadsheet.

- "Avoiding errors" is about techniques and tools that should help the developer create spreadsheets that do not have errors in the first place. These approaches support the creation process of spreadsheets.

In our work, we however aim to develop a finer-grained categorization scheme to classify the existing approaches to automated spreadsheet QA. The main categories of our proposed scheme are shown in Table 1.

The categories (1) and (2) can serve both the purpose of finding and avoiding errors. A good visualization, for example, of cell dependencies, helps the user to spot a problem. At the same time, a visualization can be used to highlight cells or areas for which there is a high probability that an error will be made in the future, for example, when there are repetitive structures in the spreadsheet. Static analyses can both identify already existing problems such

	Finding errors	Avoiding errors
(1) Visualization-based approaches	x	x
(2) Static code analysis & reports	x	x
(3) Testing approaches	x	
(4) Automated fault localization & repair	x	
(5) Model-driven development approaches		x
(6) Design and maintenance support		x

Table 1: Overview of main categories of automated spreadsheet QA.

as references to empty cells and serve as indicators for potential problems, e.g., by listing formulas which are too complex. The techniques falling into the categories (3) and (4) mainly contribute to the problem of "Finding and fixing errors" as they either help the user to identify the existence of a problem or to localize the error causes. The methods in the categories (5) and (6) often provide means to avoid errors, e.g., by supporting the refactoring process or adding an additional layer of abstraction. In general, the schema shown in Table 1 serves as a rough guideline for the categorization. There might be individual techniques within certain subcategories, which can serve both the purposes of finding and avoiding errors.

We summarize the main idea of the individual families of approaches as follows.

- Visualization-based approaches: These approaches provide the user with a visually enhanced representation of some aspects of the spreadsheet to help him or her understand the interrelationships and dependencies between cells or larger blocks of the spreadsheet. These visualizations help the user to quickly detect anomalies and irregularities in the spreadsheet.

- Static analysis & reports: These approaches are based on static code analysis and aim to point the developer to potentially problematic areas of the spreadsheet. Examples of techniques include "code smells" or the detection of data clones but also the typical family of techniques found in commercial tools capable of detecting circular dependencies or reporting summaries about unreferenced cells.

- Testing-based techniques: The methods in this category aim to stimulate and support the developer to systematically test the spreadsheet application during or after construction. The supporting tools for example include mechanisms for test case management, the automated generation of test cases or analysis of the test coverage.

- Automated fault localization & repair: The approaches in this category rely on a computational analysis of possible causes of an error or unexpected behavior (algorithmic debugging). They rely on additional input by the developer such as test cases or statements about the correctness of individual cells. Some approaches are also capable of providing "repair" suggestions.

- Model-driven development approaches: Methods in this category mainly adopt the idea of using (object-oriented) conceptual models as well as model-driven software development techniques, which are nowadays quite common in the software industry. The typical advantages of such approaches include the introduction of additional layers of abstraction or the use of code-generation mechanisms.

- Design and maintenance support: The approaches in this category either help the spreadsheet developer to end up with better error-free designs or support him or her during spreadsheet construction. The mechanisms proposed in that context for example include automated refactoring tools, methods to avoid wrong cell references, and exception handling.

Table 2 outlines the structure of the main sections of the paper.

4. Visualization-based approaches

Visualization-based approaches are helpful in different ways. They can, for example, help a developer or reviewer understand a given spreadsheet and its formulas, so that he or she can check it more easily for potential errors or bad design. In addition, visualizations are a good starting point for a reviewer other than the original author of the spreadsheet to understand

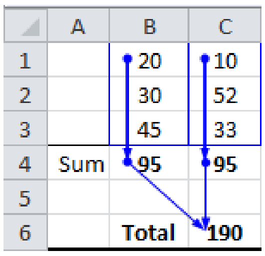
4. Visualization-based approaches	4.1. Dataflow and dependency visualization 4.2. Visualization of related areas 4.3. Semantic-based visualizations 4.4. Information Visualization approaches
5. Static code analysis & reports	5.1. Unit and type inference 5.2. Spreadsheet smells 5.3. Static analysis in commercial tools
6. Testing approaches	6.1. Test adequacy and test case management 6.2. Automated test case generation 6.3. Assertion-based testing 6.4. Test-driven spreadsheet development
7. Automated fault localization & repair	7.1. Trace-based candidate ranking 7.2. Constraint-based fault localization 7.3. Repair approaches
8. Model-driven development approaches	8.1. Declarative spreadsheet models 8.2. Spreadsheet templates 8.3. Object-oriented visual models 8.4. Relational spreadsheet models
9. Design and maintenance support	9.1. Reference management 9.2. Exception handling 9.3. Changes and spreadsheet evolution 9.4. Refactoring 9.5. Reuse

Table 2: Outline of the main parts of the paper.

its basic structure and the dependencies between the formulas. A typical application scenario is thus the use of visualizations in the auditing process, see, e.g., [19] or [20]. We categorize the different approaches for spreadsheet visualization from the literature as follows.

4.1. Dataflow and dependency visualization

A number of approaches visualize the dataflow in the spreadsheet and the corresponding dependencies of the formula cells, see, e.g., [21, 22, 23, 24, 25, 26, 27, 28] and [29]. In many cases, arrows are used to represent the usage of a cell in a formula, which is a standard feature of commercial spreadsheet environments like MS Excel as shown in Figure 1.



	A	B	C
1		20	10
2		30	52
3		45	33
4	Sum	95	95
5			
6		Total	190

Figure 1: Simple dependency visualization in MS Excel.

One of the earlier works going beyond simple dependency visualizations was presented by Davis in [19]. In addition to the use of arrows within the spreadsheet to visualize dependencies between cells, spreadsheets are visualized as graphs. The graph visualization is based on a spreadsheet description language proposed by Ronen et al. earlier in [30] to model the functionality of a spreadsheet. In these graphs, the cells correspond to the nodes and edges represent dependencies between cells. Two experiments with users were performed in which the new techniques – arrows and graphs – were compared with the existing features of MS Excel 3.0. At that time, MS Excel could only provide a listing of cell dependencies but had no graphical visualization. In the first experiment, 27 students had to find all dependent cells of a given cell. In the second experiment involving 22 students, the task was to correct an observed fault in a given cell. Overall, both new approaches were found to be more helpful for the given tasks than the standard functionality of MS Excel. Interestingly, the simple arrow-based approach was outperforming the more complex dependency graph approach.

Another more advanced method for dependency visualization was presented by Igarashi et al. in [21]. In their work, the authors rely on an animated and interactive visualization approach and “fluid interfaces” to give the user a better understanding of the data flow in the spreadsheet. Advanced animations are used to visualize which cells are input to other calculations. These animations made it possible to represent comparably complex dependencies in a visual form. In addition, users could interact with the visualizations and thereby manipulate the references through drag and drop operations, e.g., to move references, scale referenced arrays or interactively fill areas with formulas. To evaluate their approach, a prototype system was built, tested with comparably small spreadsheets and informally discussed in their paper. A study with real users and with large spreadsheets was however not done.

A different approach to visualize complex dependencies in spreadsheets is to represent parts of the spreadsheet in three-dimensional space as done, e.g., in [22] or [25]. In the approach proposed by Shiozawa et al. [22], for example, the spreadsheet is rendered in 3D and the user can interactively manipulate the visualization and lift cells or groups of cells. The connected cells are lifted to some extent based on a distance metric, allowing the user to better distinguish between cell dependencies, which are drawn as arrows in the 3D space. Similar to the work of Igarashi et al. mentioned above, the evaluation of the approach was limited to an informal discussion of a prototype system.

In [23], Chen and Chan proposed additional techniques for cell dependency visualization in spreadsheets. One of the main ideas is to visualize the dependencies between larger blocks of formulas in neighboring cells instead of displaying arrows between individual cells as done, e.g., in MS Excel. An alternative visualization of dependencies between larger blocks of the spreadsheet is proposed by Kankuzi and Ayalew in [26] and [27]. In their approach, the cells are first clustered and the resulting dependencies are then displayed as a tree map in an external window.

A more recent proposal to spreadsheet visualization was made by Hermans et al. in [28] and [29]. In their approach, the user can inspect the data flows within the spreadsheet on different levels of detail. On the global view, only the different worksheets of the spreadsheet and their dependencies are shown; on the lowest level, dependencies between individual cells are displayed. On an intermediate level, the spreadsheet is sliced down to smaller areas of geometrically adjacent cells. Beside the arrows that are used to indicate dependencies, their visualization method also displays the mathematical functions used in the calculations. In [28], Hermans et al. evaluate their dataflow visualization in two steps. In the first round, an interview involving 27 subjects about the general usefulness of such diagrams was conducted; the second part consisted of 9 observational case studies in which the task for the participants consisted in transferring or explaining their complex real-world spreadsheets to another person. The observations during the study and the qualitative feedback obtained in the post-experiment interviews indicated that the proposed techniques are well suited for the task of spreadsheet comprehension and helpful for auditing and validation purposes.

4.2. Visualization of related areas

Different methods and tools for identifying and visualizing semantically related or structurally similar blocks of cells were proposed in [31, 32, 33, 34]. These blocks can be highlighted using different colors to make it easier for the user to understand the logical structure of the spreadsheet or to identify irregularities as done in [35].

In the work of Mittermeir and Clermont [31], the concept of "logical areas" is introduced as a first step in their approach. Such areas can be automatically identified by looking for structurally similar (equivalent) formulas in different areas of the spreadsheet. Such areas are for example the result of a formula copy operation by the user during the construction process. Since a preliminary study with a prototype tool on 78 large real-world spreadsheets revealed that relying on logical areas alone reaches its limits for larger spreadsheets, the concept of "semantic classes" was introduced. In this semi-automated approach, the user manually specifies related areas in the spreadsheet. Based on this user-provided input and information about the spatial arrangement of potentially related cells, further reasoning about areas with high similarity in the spreadsheet can be performed. The work was later on improved in [32], where semantic classes were identified based on the information contained in label cells and a set of heuristics. An alternative method for decomposing a given spreadsheet for the purpose of visualization was presented in [33, 34]. In that work, the identification of areas is based on properties of the data flow in the spreadsheet.

Both the approaches proposed in [31] and [33, 34] were discussed in the corresponding papers using one artificial spreadsheet with a few dozen formulas as an example. In [36], Clermont et al. report that the approach from [31] was used to audit real-world spreadsheets, leading to detected error rates in spreadsheets that are in line with those reported in the literature[4]. An evaluation regarding the question to which extent the additional tool support increases the error detection rate or speeds up the inspection process was however not conducted.

In [20], Sajaniemi proposes two further visualization approaches called S2 and S3. The basic idea is to detect equivalent formulas in blocks of

[4]See, e.g., [4], for a discussion of error rates.

cells and visualize the dependencies between individual blocks. A theoretical comparison with the visualization techniques proposed in [19, 21, 30, 37, 38, 39] and the auditing functionality of MS Excel 7.0 was done, showing different advantages of their approaches. Beside the visualization approaches, Sajaniemi's work represents an interesting methodological contribution, as he proposes a systematic way of theoretically analyzing and comparing different visualization techniques.

4.3. Semantic-based visualizations

The missing semantics for the formulas of a spreadsheet, which is caused by the use of numbered cell references instead of information-carrying names, is a well-known problem in spreadsheet research. This was already discussed in an early work by Hendry et al. [37], where they proposed a system for annotating cells in order to describe their semantics.

The work by Chadwick et al. presented in [40] is based on the observation of two typical types of errors that are made by many spreadsheet users when creating formulas: (1) formulas sometimes reference the wrong cells as inputs; (2) formulas are sometimes copied incorrectly. To deal with the first problem, the authors propose different techniques to make the formulas more intuitively readable. One of the techniques is for example to transform a formula like `=SUM(F6:F9)` into `=SUM(Night Wages_Grade1:Night Wages_Grade4)` based on cell labels within the spreadsheet. Another idea is to represent complex formulas in a visual form. For this purpose, the formulas are decomposed, cell references are replaced with readable names and operators are translated into natural language such that the logic of the formula can be understood more easily. As a solution to problems arising from wrongly copied formulas, Chadwick et al. propose to use visual indicators and mark copied cells and their origin with the same color and add an additional comment to the original cell.

The different methods of the formula visualization were evaluated through a survey involving 63 students. The students had to rank the methods with respect to clarity and ease of understanding. The most visual method was ranked first in that survey; interestingly, however, the usual notation of Excel with cell references was ranked second and was better accepted than the above-described approach in which cell references were replaced with labels. The visualization that was used to highlight copied formulas was evaluated through a small user study with 5 students. The participants had to construct a spreadsheet and were provided with the additional visualization in that process. The results indicated that the participants liked the approach as they confirmed its usefulness to check a spreadsheet for correctness.

Nardi and Serrecchia [41] propose a more complex approach to reconstruct the underlying model of the spreadsheet, where a knowledge base is constructed and reasoning mechanisms are developed to describe calculation paths of individual cells with descriptive names. Although the approach was implemented prototypically, a systematic evaluation was not done.

4.4. Information Visualization approaches

In [42], Brath and Peters apply techniques from the field of Information Visualization to spreadsheet analysis. These visualizations support the developer in the process of detecting anomalies in the spreadsheet. In contrast to some of the approaches described so far, the aim is thus not to visualize the data flow or the structures of the spreadsheet but the data itself. To that purpose, a 3D representation is proposed, where the cell values are for example shown as bars instead of numbers. Higher numbers result in higher bars. Using the corresponding tool, the user can navigate through the 3D space, detect outliers or unexpected patterns in the data. The general feasibility of the method is informally discussed in the paper based on two case studies[5].

In general, a number of techniques from the field of Information Visualization, e.g., the "fisheye"-based approach described in [43], can in principle be applied to visualize large tabular data in spreadsheets for inspection purposes. The work of Ballinger et al. [24] is an example for such a work that relies, among others, on 3D diagrams and a fisheye view to visualize data dependencies. Overall, however, the number of similar works that rely on Information Visualization techniques appears to be limited.

4.5. Discussion

Quite a number of proposals have been made in the literature that aim to represent certain aspects of a spreadsheet in visual form. The purposes of the visualization include in particular spreadsheet comprehension, e.g., in an auditing context, and in particular anomaly and error detection.

Regarding the research methodology, only in very few and more recent papers a systematic and rigorous experimental evaluation of the proposed

[5]The general idea presented in [42] was later on implemented in a commercial tool by Oculusinfo Inc. `http://www.oculusinfo.com`.

methods has been done. In most cases, the validation is limited to qualitative interviews or surveys involving a comparably small set of participants or the informal discussion of prototype systems and individual use cases. The true applicability and usability for end users of many approaches is often unclear. Many works in that field would thus benefit if more systematic evaluations and user studies were performed as it is done for example in the fields of Human Computer Interaction and Information Systems. Possible evaluation approaches for visualization techniques include spreadsheet construction, inspection, or understanding exercises as done in IS spreadsheet research, e.g., in [9, 10, 44] or [45], but also observational approaches based, e.g., on think-aloud protocols or usage logs.

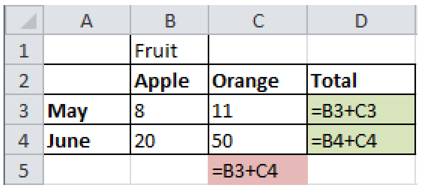
Regarding practical tools, the market-leading tool MS Excel incorporates only a small set of quite simple visualization features for spreadsheet analysis or debugging. Cell dependencies can be visualized as shown in Figure 1 or as colored rectangles highlighting the referenced cells of a formula. In addition, a small visual clue – a green triangle at the cell border – is displayed when some of the built-in error checking rules are violated. With respect to the idea of using "semantic" variable names instead of cell references, spreadsheet systems like MS Excel allow the developer to manually assign names to cells or areas to make the spreadsheets more comprehensible.

5. Static Analysis and Reports

Static code analysis or reporting-based approaches analyze the formulas of the spreadsheets and show possible faults or bad spreadsheet design that can lead to faults in the future. In contrast to automated fault localization approaches described in Section 7, the approaches in this category do not use the values in cells or information from test cases to find errors. Instead, they rather analyze the formulas themselves and the dependencies between them, look at static labels, and determine other structural characteristics of the spreadsheets.

5.1. Unit and type Inference

A major research topic in the last decade was related to "unit and type inference" [46, 47, 48, 49, 50, 51, 52, 53, 54]. The main idea of these approaches is to derive information about the units of the input cells and use this information to assess if the calculations in the formulas can be plausible with respect to the units of the involved cells. To obtain information about a cell's unit, its headers can be used. Figure 2 shows an example illustrating the idea [46]. The formulas in cell `D3` and `D4` can be considered legal. They combine apples with oranges, which are both of type fruit. In contrast, `C4` could be considered illegal, as cells with incompatible units are combined, i.e., apples from May with oranges from June. With the help of such a unit inference mechanism, the semantics of a calculation can be checked for errors. The process of deriving the unit information from header cells is called header inference.



	A	B	C	D
1		Fruit		
2		**Apple**	**Orange**	**Total**
3	**May**	8	11	=B3+C3
4	**June**	20	50	=B4+C4
5			=B3+C4	

Figure 2: Unit inference example; adapted from [46].

The idea to use a unit inference system to identify certain kinds of potential errors was introduced by Erwig and Burnett in 2002 [46]. In their initial approach, a cell could have more than one unit. However, this first work provided no explicit procedures of how the header inference should be done. In addition, there were some limitations regarding certain operators. Later on, a header inference approach was proposed in [49], so that the system could work without or with limited user interaction. Other improvements to the basic approach including various forms of more sophisticated reasoning were put forward in [47, 49, 51, 52, 53, 54]. In [53] and [54], for example, the idea is to do a semantic analysis of the labels in order to map the labels to known units of measurements. Based on this information, more precise forms of reasoning about the correctness of the calculations become possible. In contrast to the latter approaches based on semantic analysis of labels, in the work described in [55] the assumption is that the user manually enters the units and labels for the input cells and the system is then able to make the appropriate inferences for the output cells.

The idea of considering relationships between headers ("is-a", "has-a"), a different reasoning strategy and a corresponding tool capable of processing Excel documents were presented in [48] and [50]. The first work for this approach [46] included no evaluation. A first small evaluation with 28 spreadsheets was done in [49] to test the header inference and the error detection mechanism. For both sets of spreadsheets the numbers of detected errors and incorrect header and unit inferences were counted. The header and unit inferences were checked by hand and the system showed good accuracy. Regarding error detection, the system was capable of finding errors in 7 student spreadsheets. Since the total number of errors is not reported, no information about the error detection rate is available. In later papers on the topic including [53, 54, 56, 57], the systems were evaluated by comparing them with previous approaches using the EUSES spreadsheet corpus [58]. Again, the evaluation was done by counting and comparing the detected errors using different approaches.

5.2. Spreadsheet smells

The term "spreadsheet smells" was derived from code smells in software maintenance [59], where it is used for referring to bad code design. These designs are not necessarily faults themselves, but can lead to faults during the future development of the software, for example, when the software is to be refactored or expanded. A typical example for a code smell is the *duplication* of code fragments. If the same part of code is contained several times in a program, it is usually better to place it into a function so that eventual changes to the code fragment have only to be done once. Duplicated code in addition makes the code harder to read.

Spreadsheet smells are a comparably recent topic in spreadsheet research. Hermans and colleagues were among the first to adapt the concept of code smells to the spreadsheet domain, see, e.g., [60, 61, 62]. Similar ideas have already been proposed earlier in the context of spreadsheet visualization, where heuristics were used to identify irregularities in spreadsheets [35, 42].

In general, spreadsheet smells are heuristics which describe bad designs that can lead to errors when the spreadsheet is changed or when a new instance of it is created with new input data. In [60], Hermans et al. propose so-called "inter-worksheet smells". These smells indicate bad spreadsheet design based on the analysis of dependencies between different worksheets. If, for example, a formula has too many references to another worksheet, it probably should be moved to that worksheet. In addition to adapting the code smells to the spreadsheet domain, Hermans et al. also introduced metrics to discover these smells and a means to visualize them in their own worksheet dependency visualization approach [29] (see Section 4.1). "Formula smells" were discussed in [61]. These smells represent bad designs of individual formulas, e.g., when a formula is too complex. Later on, Hermans et al. propose a method for finding data clones in a spreadsheet [62]. To evaluate their spreadsheet smell approach, Hermans et al. performed both a quantitative and a qualitative evaluation in [60]. For the quantitative evaluation, the EUSES spreadsheet corpus was searched for the different types of inter-worksheet spreadsheet smells to understand how frequent these smells occur. In the qualitative evaluation, they identified smells in the spreadsheets of 10 professional spreadsheet developers and discussed the smells with the developers. The evaluation proved that the detected smells point to potential weaknesses in the spreadsheet designs. The same type of evaluation was done for the formula smells in [61].

The work of Cunha et al. in [63] is also based on the idea of spreadsheet smells. In contrast to the works by Hermans et al., they did not aim at adapting known code smells but rather tried to identify spreadsheet-specific smells by analyzing a larger corpus of spreadsheets.

5.3. Static analysis in commercial tools

Static analysis techniques are often part of commercial spreadsheet environments and spreadsheet auditing tools. As mentioned in Section 4.5, MS Excel, for example, is capable of visualizing "suspicious" formulas. A pre-defined set of rules is checked to determine if a formula is suspicious, e.g., when it refers to an empty cell or when a formula omits cells in a region. Typical spreadsheet auditing tools such as the "Spreadsheet Detective" [64][6] also strongly rely on the identification of such suspicious formulas using static analyses and produce corresponding reports. To identify these formulas, different heuristics are used, which can take the formula complexity into account, e.g., by checking if there are multiple IF-statements. Other indicators include duplicated named ranges or numbers quoted as text. Some audit tools also comprise mechanisms to support spreadsheet evolution and versioning, e.g., by listing the differences between two variants of the same spreadsheet [65]. An in-depth analysis of these tools is however beyond the scope of our work.

5.4. Discussion

The goal of static analysis techniques usually is to identify formulas or structural characteristics of spreadsheets which are considered to be indicators for potential problems. The accuracy of these methods depends on

[6]http://www.spreadsheetdetective.com

the quality of the error detection heuristics or metrics that are used to define a smell. Generally, such static analysis tools represent a family of error detection methods which can be found in commercial tools.

Type and unit systems go beyond pure analysis approaches and try to apply additional inferencing to detect additional types of potential problems and can be considered a lightweight semantic approach. While such inferencing techniques have the potential of identifying a different class of errors, there is also some danger that they detect too many "false positives".

From the perspective of the research methodology, both quantitative and qualitative methods are applied in particular in the more recent works. Evaluations are done using existing document corpora and spreadsheets created by students or professionals. However, a potential limitation when using the EUSES corpus in that context is that the intended semantics of the formulas which are considered faulty by a certain technique are for most formulas not known. Thus, we cannot determine with certainty if the formula is actually wrong and the technique was successful. From the end-user perspective, many results of a static analysis, e.g., code smells, can be quite easily communicated and explained to the user.

With respect to "smell" detection based on complex unit inference, Abraham et al. [66] conducted a think-aloud study involving 5 subjects. One goal of the study was to evaluate if the users would understand the underlying concepts well enough to correct the errors reported by their tool. Their observations indicate that the subjects, who were trained on the topic before the experiment, did understand how to interpret the feedback by the tool and correct the unit errors without the need to understand the underlying reasoning process.

6. Testing approaches

In professional software development processes, systematic testing is crucial for ensuring a high quality level of the created software artifacts. Typically, testing activities are performed by different groups of people in the various phases of the process; both manual as well as automated test procedures are common. As non-professional spreadsheet developers mostly have no proper education in Software Engineering, the testing process is assumed to be much less structured and systematic. In addition, the developer in many cases might be the only person that performs any tests.

Given the immediate-feedback nature of spreadsheets, testing can be done by simply typing in some inputs and checking if the intermediate cells and output cells contain the expected values. Commercial spreadsheet tools such as MS Excel do not provide any specific mechanisms to the user for storing such test cases or running regression tests. Furthermore, these tools do not help the developer assess if a sufficient number of tests has been made. In the following, we review approaches that aim at transferring and adapting ideas, concepts, tools and best-practice approaches from standard software testing to the specifics of spreadsheet development.

6.1. Test adequacy and test case management

A number of pioneering works in this area have been done by the research group of Burnett, Rothermel and colleagues at Oregon State University. Already in 1997, they discussed test strategies and test-adequacy criteria for form-based systems and later on proposed a visual and incremental spreadsheet testing methodology called "What You See Is What You Test" (WYSIWYT) [67, 14, 68]. During the construction of the spreadsheet, the user can interactively mark the values of some derived cells to be correct for the currently given inputs. Based on these tests, the system determines the "testedness" of the spreadsheet. This is accomplished through an automatic evaluation of a test adequacy criterion which is based on an abstract model of the spreadsheet, spreadsheet-specific "definition-use" (du) associations and dynamic execution traces. Later on, several improvements to this approach were proposed, such as scaling it up to large homogeneous spreadsheets that are often found in practice, adding support for recursion, or dealing with questions of test case reuse [69, 70, 71, 72]. The approach was ported from Forms/3 to Microsoft Excel with additional support of special features such as higher-order-functions and user defined functions [73]. In [74], Randolph et al. presented an alternative implementation of the WYSIWYT approach, which was designed in a way that it can be used in combination with different spreadsheet environments.

In [68], the results of a detailed experimental evaluation of the basic approach are reported whose aim was to assess the efficacy of the approach and how "du-adequate" test suites compare to randomly created tests. In their evaluations, they used 8 comparably small spreadsheets, in which experienced users manually injected a single fault. Then, a number of du-adequate and random pools of test cases were created. The analysis of applying these tests among other things revealed that the du-adequate pools outperformed random pools of the same size in all cases with respect to their ability of detecting the errors. A further study involving 78 subjects in which the efficiency and effectiveness of the approach was tested is described in [75].

6.2. Automated test case generation

When using the WYSIWYT approach, the spreadsheet developer receives feedback about how well his or her spreadsheet is tested. Still, the developer has to specify the test cases manually. To support the user in this process, Fisher et al. proposed techniques for the automated generation of test cases [76, 77].

In these works, two methods for generating values for a test case were evaluated. The "Random" method randomly generates values and checks if their execution uses a path of a so far unvalidated definition-use pair. The second, goal-oriented method called "Chaining" iterates through the unvalidated definition-use pairs and tries to modify the input values in a way that both the definition and the use are executed. If the generation of input values for a new test case is successful, the user only has to validate the output value to obtain a complete test case. To assess the effectiveness and efficiency of their approach an offline simulation-based study without real users based on 10 comparably small spreadsheets containing only integer type cells was performed [77]. The results clearly showed that the "Chaining" method was more effective than the "Random" method.

In [78], the AutoTest tool was presented, which implements a different strategy for automatic test case generation and uses constraint solving to search for values that lead to the execution of the desired definition-use pairs. This method is guaranteed to generate test cases for all feasible definition-use pairs. The method was compared with the previously described method from [77] using the same experimental setup and showed that AutoTest was both more effective and could generate the test cases faster.

6.3. Assertion-based testing

A very different approach for users to test and ensure the validity of their spreadsheets was presented by Burnett et al. in [79]. In this work, the concept of assertions, which can be found in some imperative languages, was transferred to spreadsheets. Assertions in the spreadsheet domain (called "guards" here) correspond to statements about pre- and post-conditions about allowed cell values in the form of Boolean expressions. The assertions are provided by the end user through a corresponding user-oriented tool and automatically checked and partially propagated through the spreadsheet in the direction of the dataflow. Whenever a conflict between an assertion and a cell value or between a user given and a propagated assertion is detected, the user is pointed to this problem through visual feedback.

In [79] and [80] different controlled experiments were performed to evaluate the approach. The experimental setup in [79] consisted of a spreadsheet testing and debugging exercise in which 59 subjects participated. About half of the subjects were using an "assertion-enabled" development environment, whereas the other group used the same system without this functionality. The analysis revealed that assertions helped users to find errors both more effectively and efficiently across a range of different error types. A post-experiment questionnaire furthermore showed that the users not only understood and liked using the assertions but that assertions are also helpful to reduce the users' typical over-confidence about the correctness of their program. Ways of how to extend the concept of guards to multiple cells were discussed in [81]; a small think-aloud study indicated that such mechanisms must be carefully designed as the expectations around the reasoning behind such complex guards was not consistent across users.

6.4. Test-driven spreadsheet development

Going beyond individual techniques for test case management and test case generation, McDaid et al. in [82] address the question if the principle of test-driven development (TDD), which received increased attention in the Software Engineering community, is applicable in spreadsheet development processes. Following this principle, the user iteratively creates test cases first that define the intended spreadsheets functionality and writes or changes formulas afterwards to fulfill the test. This continuing and systematic form of testing shall help to minimize the number of faults that remain in the final spreadsheet.

In their work, the authors argue that spreadsheets are well suited for the TDD principle and present a prototype tool. To evaluate the approach, 4 users with different background in spreadsheet expertise and TDD were asked to develop different spreadsheets and corresponding test cases. From the subsequent interviews, the authors concluded that the approach is easy to use and most of the participants stated that the approach is beneficial, even if the required time for the initial development increased measurably.

6.5. Discussion

One of the major problems of end-user programs is that they are usually not rigorously tested. As demonstrated through various experimental studies, better tool support during the development process helps users to develop spreadsheets with fewer errors. However, commercial spreadsheet systems contain limited functionality in that direction. MS Excel only provides a very basic data validation tool for describing allowed types and values for individual cells, which can be seen as a form of assertions.

One problem in that context lies in the design of user interfaces for test tools that are suitable for end users. While in-depth evaluations of the effectiveness of the test case generation or test adequacy were performed as described above, the number of experiments regarding usability aspects with real users is still somewhat limited. Another main issue is the limited awareness of end users regarding the importance and value of thorough testing and their overconfidence in the correctness of the programs. More research about how to stimulate users to provide more information to the QA process in the sense of [80] is therefore required.

In that context, a better understanding is required in which ways spreadsheet developers actually test their spreadsheets or would be willing to at least partially adopt a test-driven development principle. In [83], Hermans made an analysis based on the EUSES corpus which revealed that there are a number of users who add additional assertions in the form of regular formulas to their spreadsheets. These assertions or tests are however often incomplete and have a low coverage, which led the author to the development of an add-on tool that automatically points the user to possible improvements for such user-specified assertions.

The application of *mutation testing* techniques to spreadsheet programs was discussed in [84]. Mutation testing consists of introducing small changes to a given program and checking how many of these mutants can be eliminated by a given set of tests. In their work, Abraham and Erwig propose a set of mutation operators for spreadsheets where some of them are based on operators that are used for mutating general-purpose languages and some of them are spreadsheet-specific. Generally, mutations can be used to test the coverage or adequacy of manually or automatically created test suites. In the broader context of fault detection and removal, they can however also be used to evaluate debugging approaches as was done in the spreadsheet literature, e.g., in [85, 86, 87] or [88].

7. Automated Fault Localization & Repair

The approaches in this category address scenarios in the development process, in which the spreadsheet developer enters some test data in the spreadsheet and observes unexpected calculation results in one or more cells. Such situations arise either during the initial development or when one of the above-mentioned test methodologies is applied. Already for medium-sized spreadsheets, the set of possible "candidates" that could be the root cause of the unexpected behavior can be large, in particular when the spreadsheet consists of longer chains of calculations that involve many of the spreadsheet's cells. Without tool support, the user would have to inspect all formulas on which the cell with the erroneous value depends and check them for correctness. The goal of most of the approaches in this category therefore is to assist the user in locating the true cause of the problem more efficiently, in many cases by ranking the possible error sources (*candidates*). Some of the approaches even go beyond that and try to compute a set of possible "repairs", i.e., changes to some of the formulas to achieve the desired outcomes. In contrast to static code analysis and inspection approaches, the basis for the required calculations usually comprises a specification of input values and expected output values or test cases.

7.1. Trace-based candidate ranking

An early method for candidate ranking which has some similarities to spectrum-based fault localization methods for imperative programs was presented by Reichwein et al. in [89] and [90]. In their method, they first propose to transfer the concept of program slicing to spreadsheets in order to eliminate impossible candidates in an initial step. Their technique uses user-specified information about correct and incorrect cell values and considers those cells that theoretically contribute to an erroneous cell value to be possibly faulty. A cell's formula is more likely to be faulty, if it contributes to more values that are marked as erroneous. Similarly, a formula is more likely to be correct if it contributes to more correct cell values. If a cell contributes to an incorrect cell value but the path to it is "blocked" by a cell with a correct value, its fault likelihood is assumed to be somewhere in between. In later works [91, 92], in which besides two further heuristics for fault localization a deeper analysis of the method's effectiveness factors were discussed, this technique is called "Blocking Technique". The "Blocking Technique" was evaluated in a user study involving 20 subjects in [90].

The task of the participants, which were split into two groups of equal size, was to test a given spreadsheet. Both groups were using a tool that implemented the WYSIWYT approach. One group additionally had the described fault localization extension activated. An interview after the experiment and the analysis of the experiment data showed – among other aspects – that most users appreciated the possibility to use the fault localization and they considered it to be particularly useful to locate the "harder" faults.

A similar technique was proposed by Ayalew and Mittermeir in [93], where for a faulty cell value the cells are highlighted that have the most influence on it. Later on, Hofer et al. in [87] explicitly proposed to adapt spectrum-based fault-localization from the traditional programming domain to spreadsheets. In contrast to previous works, they use a more formal approach with similarity coefficients to calculate the fault probabilities of the spreadsheet cells. They evaluated their version of spectrum-based fault localization for spreadsheets on a subset of the EUSES spreadsheet corpus and compared the fault localization capabilities of spectrum-based fault localization to those of two model-based debugging approaches (see Section 7.2).

7.2. Constraint-based fault localization

The following approaches translate a spreadsheet into a constraint-based representation, such that additional inferences about possible reasons for an unexpected value in some of the cells can be made.

In [94], Jannach and Engler presented an approach in which they first translated the spreadsheet into a Constraint Satisfaction Problem (CSP)[95]. Then, based on user-specified test cases and information about unexpected values in some of the cells, they used the principle of Model-Based Diagnosis (MBD) to determine which cells can theoretically be the true cause for the observed and unexpected calculation outcomes. With their work, they continue a line of research in which the MBD-principle, which was originally designed to find problems in hardware artifacts, is adapted for software debugging, see, e.g., [96] or [97]. Technically, an approach similar to [97] was adopted, which is capable of dealing with multiple "positive" and "negative" test cases and at the same time supports the idea of user-provided assertions.

In a first evaluation with relatively small artificial spreadsheets containing a few dozen formulas, it was shown that the approach is – depending on the provided test cases – able to significantly reduce the number of fault candidates. Later on, the method was further improved and optimized and embedded in the Exquisite debugging tool for MS Excel [98]. An evaluation of the enhanced version on similar examples showed significant enhancements with respect to the required calculation time. Mid-sized spreadsheets containing about 150 formulas and one injected fault could for example be diagnosed within 2 seconds on a standard laptop computer.

In a later work [88], different algorithmic improvements were proposed which helped to increase the scalability of the approach. The method was evaluated using a number of real-world spreadsheets in which faults where artificially injected. Furthermore, a small user study in the form of a debugging exercise was conducted, which indicated that the users working with the Exquisite tool were both more efficient and effective than the group that did a manual inspection. The size of the study was however quite small and involved only 24 participants.

A similar approach for finding an explanation for unexpected values using a CSP representation and the MBD-principle has been proposed by Abreu et al. in [99] and [100]. While the general idea is similar to the approach of Jannach and Engler, the technical realization is slightly different. Instead of using the Hitting-Set Algorithm [101], they encode the reasoning about the correctness of individual formulas directly into the constraint representation. Therefore, they make use of an auxiliary boolean variable for each cell representing the correctness of the cell's formula. Another difference of this approach compared to the work of Jannach and Engler is that Abreu et al. rely on a single test case only. The method was evaluated using four comparably small and artificial spreadsheets for which the algorithm could find a manually injected fault very quickly (taking at most 0.17 seconds). In [102], Außerlechner et al. evaluated this constraint-based approach using different SMT[7] and constraint solvers. For their evaluation, they created a special document corpus which both comprised spreadsheets that contain only integer calculations as well as a subset of the EUSES corpus with real number calculations. Their evaluation showed that the debugging approach of Abreu et al. can be used to find faults in medium-sized spreadsheets in real-time and that the approach is capable of debugging spreadsheets containing real numbers.

In [87], Hofer et al. propose to combine their spectrum-based fault localization approach with a light-weight Model-Based Software Debugging tech-

[7]Satisfiability Modulo Theories

nique. In particular, Hofer et al. suggest to use the coefficients obtained from the SFL technique as initial probabilities for the model-based debugging procedure. To evaluate the effectiveness of their hybrid method, they compared their approach to a pure spectrum-based approach and a constraint-based diagnosis approach. In their experiments, spreadsheets from the EUSES spreadsheet corpus were mutated using a subset of the mutation operators proposed in [84]. Overall, 227 mutated spreadsheets containing from 6 to over 4,000 formulas were used in the comparison. The results showed that the combined approach led to a better ranking of the potentially faulty cells, but was slightly slower than the pure SFL method.

7.3. Repair approaches

Repair-based approaches do not only point the users to potentially problematic formulas, but also aim to additionally propose possible corrections to the given formulas in a way that unexpected values in cells can be changed to the expected ones.

A first method for automatically determining such change suggestions ("goal-directed debugging") was presented by Abraham and Erwig in [85]. In their approach, the user states the expected value for an erroneous cell and the method computes suitable change suggestions by recursively changing individual formulas and propagating the change back to preceding formulas using spreadsheet-specific change inference rules. The possible changes that yield the desired results are then ranked based on heuristics. A revised and improved version of the method ("GoalDebug") that is better suited to address different (artificial) spreadsheet fault types discussed in [84], was presented in [103]. Later on, GoalDebug was combined with the AutoTest approach (see Section 6) to further improve the debugging results with the help of more test cases and other testing-related information [104].

To test the usefulness of their initial proposal [85], a user study with 51 subjects inspecting 2 spreadsheets with seeded faults was conducted. During the study, the subjects had to locate the faults using the WYSIWYT approach (see Section 6.1) but without the goal-directed method. The experiment revealed that the users made many mistakes when testing the spreadsheets and that the proposed approach could have prevented these mistakes. Furthermore, all the seeded faults were located with their approach.

GoalDebug was evaluated later on using an offline experiment where faults were injected into spreadsheets using a set of defined mutation operators. For the experiment, 7500 variants of 15 different spreadsheets with up to 54 formulas and 100 cells [84] were created and analyzed. The baseline for their evaluation was their own previous version of the method. The evaluation showed that GoalDebug was able to deal with all 9 defined mutation types and had a "success rate" of finding a correct repair of above 90 %, which was much better than the original version of the method.

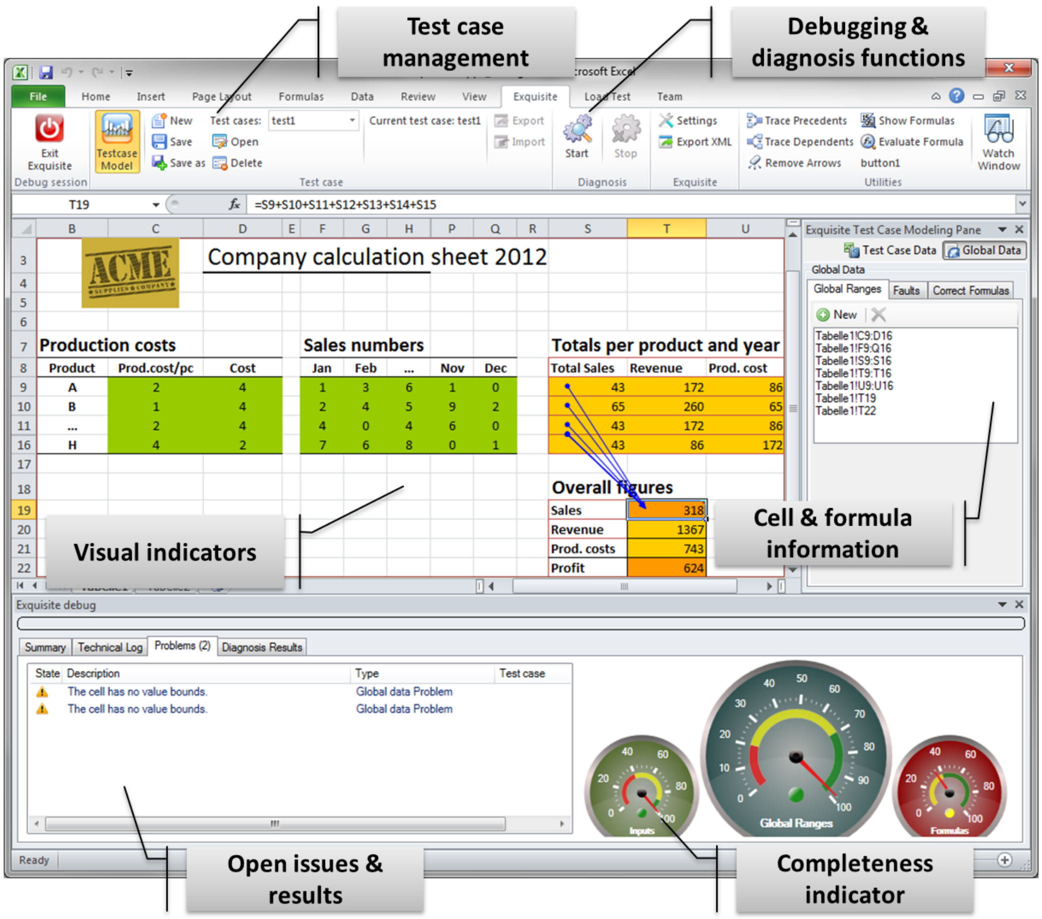
7.4. Discussion

The effectiveness of the debugging techniques reviewed in this section was mostly assessed using evaluation protocols in which certain types of faults were artificially seeded into given spreadsheets. The evaluations showed that the proposed techniques either lead to good rankings (of candidates or repair suggestions) or are able to compute a set of possible explanations. However, the spreadsheets used in the experiments often were small and the scalability of many approaches remains unclear. Besides, most of the approaches were evaluated with a non-public set of spreadsheets. Therefore, a direct comparison of the approaches is difficult. In addition, the constraint-based approaches are often limited to small spreadsheets and integer calculations.

Unfortunately, "oracle faults" are usually not discussed in the described approaches: For all approaches, the spreadsheet developer has to provide some information, e.g., the expected outcomes or which cells produce a correct output and which cells are erroneous. Most of the approaches are evaluated assuming a perfect user knowing every expected value. However, the spreadsheet developer often does not know all the required information or might accidentally provide wrong values. Further empirical evaluations should therefore consider vague or partly wrong user input.

For some of the proposed techniques, plug-in components for MS Excel have been developed, including [103] and [98], see Figure 3. Usability aspects of such tools have however not been systematically explored so far and it is unclear if they are suitable for an average or at least ambitious spreadsheet developer. More research in the sense of [105], where Parnin and Orso evaluated how and to which extent developers actually use debugging tools for imperative languages, is thus required in the spreadsheet domain.

Debugging support in commercial spreadsheet systems is very limited. Within MS Excel, one of the few features that support the user in the debugging process is the "Watch Window" as shown in Figure 4. Similar to debuggers for imperative programs, the spreadsheet developer can define watchpoints – in this case by selecting certain cells – and the current values of the cells are constantly updated and displayed in a compact form.



T19 =S9+S10+S11+S12+S13+S14+S15

Company calculation sheet 2012

Production costs

Product	Prod.cost/pc	Cost
A	2	4
B	1	4
...	2	4
H	4	2

Sales numbers

Jan	Feb	...	Nov	Dec
1	3	6	1	0
2	4	5	9	2
4	0	4	6	0
7	6	8	0	1

Totals per product and year

Total Sales	Revenue	Prod. cost
43	172	86
65	260	65
43	172	86
43	86	172

Overall figures

Sales	318
Revenue	1367
Prod. costs	743
Profit	624

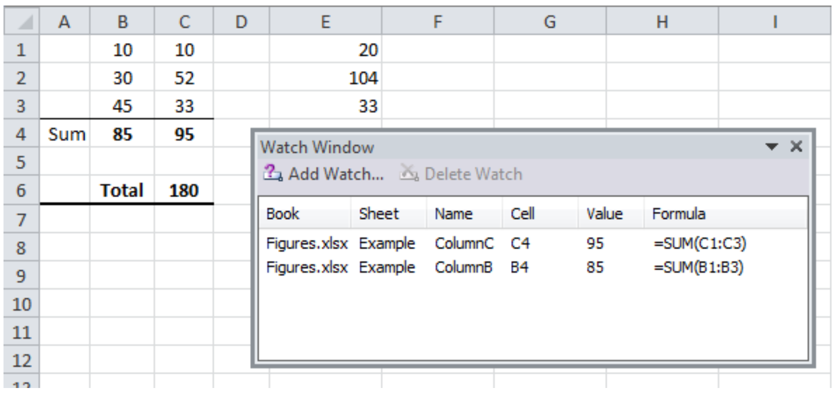
Tabelle1!C9:D16
Tabelle1!F9:Q16
Tabelle1!S9:S16
Tabelle1!T9:T16
Tabelle1!U9:U16
Tabelle1!T19
Tabelle1!T22

State	Description	Type	Test case
⚠	The cell has no value bounds.	Global data Problem	
⚠	The cell has no value bounds.	Global data Problem	

Figure 3: Debugging workbench of the Exquisite system [98]

8. Model-driven development approaches

In contrast to the approaches described in previous chapters, model-driven development approaches were not primarily designed to support the user in finding potential errors, but rather to improve the quality and structure of the spreadsheets and to prevent errors in the first place. Similar to model-driven approaches in the area of general software development, the main idea of these approaches is to introduce another layer of abstraction in the development process. Typically, the spreadsheet models in this intermediate layer introduce more abstract conceptualizations of the problem and thus serve as a bridge between the implicit idea, which the developer had of



	A	B	C	D	E	F	G	H	I
1		10	10		20				
2		30	52		104				
3		45	33		33				
4	Sum	85	95						
5									
6		Total	180						
7									
8									
9									
10									
11									
12									

Book	Sheet	Name	Cell	Value	Formula
Figures.xlsx	Example	ColumnC	C4	95	=SUM(C1:C3)
Figures.xlsx	Example	ColumnB	B4	85	=SUM(B1:B3)

Figure 4: MS Excel's watch window for cell value inspection

the spreadsheet, and the actual implementation. This way, the semantic gap between the intended idea and the spreadsheet implementation, which can become large in today's business spreadsheets [106], can be narrowed.

The abstract spreadsheet models proposed recently in the literature are used in two different phases in the development process. First, they are used as a form of "code-generators". In this scenario, parts of the spreadsheets are automatically generated from the models, thus reducing the risk of mechanical errors. Second, they are used to recover the underlying conceptual structures from an existing spreadsheet, which is similar to existing reverse engineering approaches in general software development. Following our classification scheme from Section 3, model-driven development approaches are therefore usually related to design and maintenance approaches, which we will discuss later on.

8.1. Declarative and object-oriented spreadsheet models

Isakowitz et al. were among the first to look at spreadsheet programs from a modeling perspective [38]. In their work, their main premise is that spreadsheet programs can be viewed at from a physical and a logical viewpoint, the physical being the cell's formulas and values and the logical being a set of functional relations describing the spreadsheets functionality. In their approach, spreadsheets consist of four principal components, among them the "schema" which captures the program's logic and the "data" which holds the values of the input cells. With the help of tools, this logic can be automatically extracted from a given spreadsheet and represented in a tool-independent language. In addition, the proposed system is capable of synthesizing spreadsheets from such specifications.

A similar object-oriented conceptualization of spreadsheet programs was presented later on by Paine in [107] and [108]. In the *Model Master* approach, spreadsheets are specified in a declarative way as text programs. These programs can then be passed to a compiler, which generates spreadsheets from these specifications. The logic of a spreadsheet is organized in the sense of object-oriented programming in the form of classes which encapsulate attributes and the calculation logic, see Figure 5. The comparably simple modeling language comprises a number of features including inheritance or multi-dimensional arrays to support tabular calculations.

```
company = attributes <
  incomings [ 1995:2004 ]
  outgoings [ 1995:2004 ]
  profit    [ 1995:2004 ]
>
where
  profit[ all t ] = incomings[ t ] - outgoings[ t ]
```

Figure 5: A class specification in *Model Master* [107].

Beside the automatic generation of spreadsheets from these models, the system also supports the extraction (reconstruction) of models from spreadsheets, which however requires the user to provide additional hints. The extracted models can be checked for errors or used as a standard for spreadsheet interchange. Particular aspects of structure discovery are discussed in [109][8].

To validate the general feasibility of the approach, different experiments were made in which small-sized spreadsheets were generated. The test cases used for model reconstruction were even smaller. Unfortunately and similar to the earlier work of Isakowitz et al. [38], no studies with real users were performed so far to assess the general usability of the approach at least for

[8]The work of Lentini et al. [39] is also based on the automatic extraction of the mathematical model of a given spreadsheet and a Prolog-based representation. However, their work rather focuses at the generation of a tutoring facility for a given spreadsheet and is thus only marginally relevant for our review.

advanced spreadsheet developers.

Paine described a different approach for a declarative modeling language in [110]. *Excelsior* is a spreadsheet development system which comprises a programming language built on *Prolog* and which is designed for the modular and re-usable specification of Excel spreadsheets. In addition to the standard functionality of *Prolog*, the programming language comprises specific constructs and operators to model the logic of a spreadsheet in a modular form. An example for such a specification is given in Figure 6. Based on such a design, the layout of the spreadsheet can be separated from its functionality and a compiler can be used to automatically generate a spreadsheet instance from these specifications.

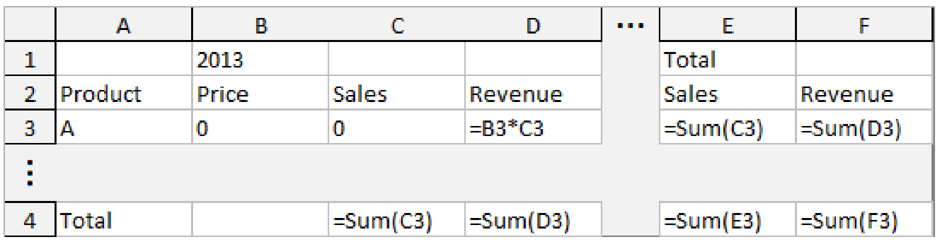
```
Year[2000] = 2000
Year[2001] = 2001
Sales[2000] = 971
Sales[2001] = 1803
Expenses[2000] = 1492
Expenses[2001] = 1560
Profit[2000] = Sales[2000] - Expenses[2000]
Profit[2001] = Sales[2001] - Expenses[2001]
Layout Year[2000:] as A2 downwards
Layout Expenses[2000:] as B2 downwards
Layout Sales[2000:] as C2 downwards
Layout Profit[2000:] as D2 downwards
```

Figure 6: A spreadsheet specification in *Excelsior* [110].

In [111], the functionality of the *Excelsior* system was tested on a larger spreadsheet. The task was to extract a model, i.e., the logical structure, of a spreadsheet with 10,000 cells and then apply several changes to it with the help of *Excelsior*. After model extraction, refactoring was found to be very easy in *Excelsior*, as only parameters had to be changed to generate a refactored and adapted spreadsheet. However, the extraction of the model was only semi-automatic and according to the authors took 2 days to complete. Moreover, no systematic evaluation to test the usability of this approach for average spreadsheet users was done.

8.2. Spreadsheet templates

In contrast to the works of Paine and colleagues, Erwig et al. proposed to rely on a *visual* and template-based method to capture certain aspects of the underlying model of a spreadsheet [112, 113, 114]. A "template" in their *Gencel* approach can in particular be used to specify repetitive areas in a spreadsheet. Figure 7 shows an example of a template specification. The design of the template can be done using a visualization that is similar to the typical UI paradigm of spreadsheet systems like MS Excel. In the example, the contents below the column headers B,C, and D are marked as being repetitive. In the model, this is indicated by the missing vertical separator lines between the column headers and the "..."-symbols between column and row headers.



	A	B	C	D	...	E	F
1		2013				Total	
2	Product	Price	Sales	Revenue		Sales	Revenue
3	A	0	0	=B3*C3		=Sum(C3)	=Sum(D3)
⋮							
4	Total		=Sum(C3)	=Sum(D3)		=Sum(E3)	=Sum(F3)

Figure 7: Spreadsheet template example; adapted from [114].

Similar to Paine's work, spreadsheet instances can be automatically generated from models. The generated spreadsheets can furthermore be altered later on in predefined ways. The supported operations include the addition or removal of groups of repetitive areas and value updates. Another feature of their approach is the use of a type system. The template-based approach also supports a reverse engineering process and the automatic reconstruction of templates from a given spreadsheet using certain heuristics [115]. To evaluate their approach, the authors discussed it in terms of the "Cognitive Dimensions of Notations" framework [116, 117] and conducted a small think-aloud study with 4 subjects [112]. Unfortunately, two of the subjects could not complete the spreadsheet development exercise because of technical difficulties; the spreadsheet created by the other participants were however error-free.

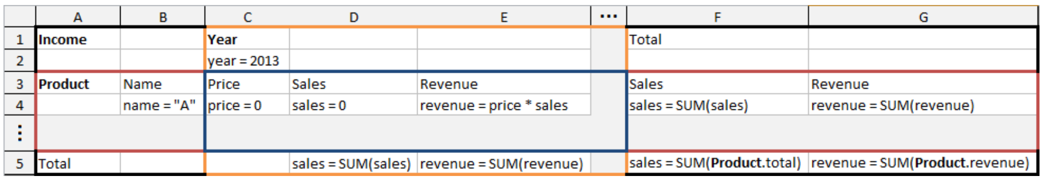
The template extraction method was evaluated in [115] with the help of a user study and a sample of 29 randomly selected spreadsheets of the EUSES spreadsheet corpus. The 23 participating users – 19 novice and 4 expert users – were asked to manually create templates for the selected spreadsheets. These manually created templates were then compared with the automatically extracted ones with respect to their correctness. The analysis revealed that the automatically generated templates were of significantly higher quality than the manually created ones and that even expert users had problems to correctly identify the underlying patterns of the spreadsheets.

8.3. Object-oriented visual models

As a continuation and extension to the template-based approach and in order to address a wider range of error types, Engels and Erwig later on proposed the concept of "ClassSheets" [118], which is similar to the work of Paine [107] mentioned above in the sense that the paradigm of object-orientation is applied to the spreadsheet domain.

Figure 8 shows an example of a ClassSheet specification, which uses a visualization similar to MS Excel. The different classes are visually separated by colored rectangles and represent semantically related cells. In contrast to the pure templates, the classes are not only syntactic structures but rather represent real-world objects or business objects in the sense of object-oriented software development. Beside the visual notation, the modeling approach comprises mechanisms to address the modeled objects rather through symbolic class names than through direct cell references.

Similar to the template-based approach described above, prototype tools were developed that support both the automated generation of spreadsheets from the models and the extraction of ClassSheet models from existing spreadsheets [119].



	A	B	C	D	E	...	F	G
1	**Income**		**Year**				Total	
2			year = 2013					
3	**Product**	Name	Price	Sales	Revenue		Sales	Revenue
4		name = "A"	price = 0	sales = 0	revenue = price * sales		sales = SUM(sales)	revenue = SUM(revenue)
⋮								
5	Total			sales = SUM(sales)	revenue = SUM(revenue)		sales = SUM(**Product**.total)	revenue = SUM(**Product**.revenue)

Figure 8: ClassSheet example; adapted from [118].

In the original paper in which ClassSheets were proposed and formalized [118], no detailed evaluation of the approach was performed. The automated extraction approach proposed in [119] was evaluated using a set of 27 spreadsheets, which contained 121 worksheets and 176 manually identified tables. According to their analysis, their tool was able to extract models from all but

13 tables. The 163 extracted models were then manually inspected. Only 12 of the models were categorized as being “bad‘” and 27 as being “acceptable”. The remaining 124 models were found to be “good”.

A number of extensions to the basic ClassSheet approach were later on proposed in the literature. In [106], Luckey et al. addressed the problem of model evolution and how such updates can be automatically transferred to already generated spreadsheets to better support a round-trip engineering process. The same problem of model evolution and the co-evolution of the model and the spreadsheet instances was addressed by Cunha et al. in [120, 121]. In [122], Cunha et al. proposed an approach to support the other update direction – the automatic transfer of changes made in the spreadsheet instances back to the spreadsheet model. Further extensions to the ClassSheet approach comprise the support of primary and foreign keys as used in relational designs, the generation of UML diagrams from ClassSheet models to support model validation or mechanisms to express constraints on allowed values for individual cells [123, 124]. For most of these extensions, no systematic evaluation has been done so far.

A different, in some sense visual approach to re-construct the underlying (object-oriented) model was proposed by Hermans et al. in [125]. Their approach is based on a library of typical patterns, which they try to locate in spreadsheets with the help of a two-dimensional parsing and pattern matching algorithm. The resulting patterns are then transformed into UML class diagrams, which can be used to better understand or improve a given spreadsheet. For the evaluation of their prototype tool, they first checked the plausibility of their patterns by measuring how often they appear in the EUSES corpus. Then, for a sample of 50 random spreadsheets, they compared the quality of generated class models with manually created ones, which led to promising results.

8.4. Relational spreadsheet models

One of the main principles of most spreadsheets is that the data is organized in tabular form. An obvious form of trying to obtain a more abstract model of the structure of a spreadsheet is to rely on approaches and principles from the design of relational databases. With the goal of ending up in higher-quality and error-free spreadsheets, Cunha et al. in [126] proposed to extract a relational database schema from the spreadsheet, which shall help the user to better understand the spreadsheet and which can consequently be used to improve the design of the spreadsheet. The main outcome of such a refactoring process should be a spreadsheet design which is more modular, has no data redundancies and provides suitable means to prevent wrong data inputs. With respect to the last aspect, Cunha et al. in [127, 128] proposed to use the underlying (extracted) relational schema to provide the user with advanced editing features including the auto-completion of values, non-editable cells and the safe deletion of rows.

In the original proposal of Cunha et al. [126], no formal evaluation of the approach was performed. An evaluation of the model-based approaches proposed in [127] and [126] was however done later on in [129]. In this user study, the goal was to assess if relying on the proposed methods can actually help to increase the effectiveness and efficiency of the spreadsheet development process. The participants of the study had to complete different development tasks and these tasks had to be done either on the original spreadsheet designs or on one of the assumedly improved ones. The results of the experiment unfortunately remained partially inconclusive and the results were not consistently better when relying on the model-based approaches.

To evaluate the advanced editing features mentioned in [128], a preliminary experiment using a subset of the EUSES spreadsheet corpus was done in that work. The initial results indicate that the tool is suited to provide helpful editing assistance for a number of spreadsheets; a more detailed study about potential productivity improvements and error rates has so far not been done.

8.5. Discussion

The model-driven development approaches discussed in this section aim to introduce additional syntactic or semantic abstraction layers into the spreadsheet development process. Overall, these additional mechanisms and conceptualizations shall help to close the semantic gap between the final spreadsheet and the actual problem in the real world, lead to higher quality levels in terms of better designs and fewer errors, and allow easier maintenance. Going beyond many model-driven approaches for standard software artifacts, automated "code" generation and support for round-trip engineering are particularly in the focus of spreadsheet researchers.

However, following a model-based approach comes with a number of challenges, which can also be found in standard software development processes. These challenges for example include the problem of the co-evolution of models and programs. Furthermore, the design of the modeling language plays an important role and often a compromise between expressivity and comprehensibility has to be made. A particular problem in that context certainly lies in the fact that the spreadsheet designers usually have no formal IT education and might have problems understanding the tools or the long-term advantages of better abstractions and structures. Furthermore, one of the main reasons of the popularity of spreadsheets lies in the fact that no structured or formal development process is required and people are used to develop spreadsheets in an ad-hoc, interactive and incremental prototyping process.

From a research perspective, many of the discussed papers only contain a preliminary evaluation or no evaluation at all. Thus, a more systematic evaluation and more user studies are required to obtain a better understanding if the proposed models are suited for typical spreadsheet developers and if they actually help them to develop spreadsheets of higher quality.

In current spreadsheet environments like MS Excel, only very limited support is provided to visually or semantically enrich the data or the calculations. One of the few features of adding semantics in a light-weight form is the assignment of symbolic names to individual cells or areas, which increases the readability of formulas. In addition, MS Excel provides some features for data organization including the option to group data cells and hide and display them as a block.

9. Design and maintenance support

The following approaches support the user in the development and maintenance processes. These approaches range from tools whose goal is to avoid wrong references over the handling of exceptional behavior to tools supporting the long-term use of spreadsheets (e.g., change-monitoring tools, add-ins for automatic refactoring and approaches that handle the reuse of formulas). All these tools play an important role in spreadsheet quality assurance as their goal is to avoid faults either by means of a clear and simple representation, by automation or by dealing with certain types of exceptional behavior.

9.1. Reference management

A major drawback of common commercial spreadsheet tools is that they provide limited support to ensure the correctness of cell references across the spreadsheet, e.g., because names of referenced cells do not carry semantic information about the content. Users often reference the wrong cells because they make off-by-one mistakes when selecting the referenced cell or accidentally use relative references instead of absolute references. Identifying such wrong references can be a demanding task. Even though systems like MS Excel support named cells and areas, most spreadsheet developers use the numbered and thus abstract cell names consisting of the row and column index.

Early approaches to address this problem – including NOPumpG [130, 131] and Action Graphics [132] – propose to give up the grid-based paradigm of spreadsheets and force the user to assign explicit names to the "cells". WYSIWYC ("What you see is what you compute") [133] is an alternative approach which retains the grid-based paradigm and proposes a new visual language for spreadsheets. The approach shall help to make the spreadsheet structures, calculations and references better visible and thus lead to a better correspondence of a spreadsheet's visual and logical structure. This should help to avoid errors caused by wrong cell references.

Unfortunately, while prototype systems have been developed, none of the above mentioned techniques have been systematically evaluated, e.g., through user studies. Therefore, it remains unclear if end users would be able to deal with such alternative development approaches and to which extent the problem of wrong cell references would actually be solved.

Finally, note that some problems of wrong cell references can be guaranteed to be avoided when (parts of) the spreadsheets are automatically generated from templates or visual models as done in the Gencel [112] and ClassSheet [118] approaches, see Section 8.2 and 8.3. In these systems, certain types of errors including reference errors can be avoided as only defined and correct update operations are allowed.

9.2. Exception Handling

The term exception handling refers to a collection of mechanisms supporting the detection, signaling and after-the-fact handling of exceptions [134]. Exceptions are defined as any unusual event that may require special processing [135]. Being aware of possible exceptional situations and handling them accordingly is an important factor to improve the quality of spreadsheets and making them more robust.

In [134], Burnett et al. propose such an approach to exception handling for spreadsheets. In their paper, they show that the *error value model* can be used for easy and adequate exception handling in spreadsheets. In the error value model, error messages (like `#DIV/0` in MS Excel) are returned instead of the expected values. The advantage of the approach using error values is that no changes to the general evaluation model in the spreadsheet paradigm are necessary. Exception handling approaches for imperative paradigms, in contrast, usually alter the execution sequence, which is not the case for spreadsheets with their static evaluation order. In addition, no special skills are required by the spreadsheet developers for exception prevention and exception handling as they can use the standard language operators (e.g., the if-then-else construct). What makes the approach of Burnett et al. different from the typical error value model in systems like MS Excel is that it supports customizable error types the end user can define to handle *application-specific* errors. Burnett et al. implemented their exception handling approach in the research system Forms/3. However, no evaluation with real users was done.

9.3. Changes and spreadsheet evolution

Spreadsheets often undergo changes and, unfortunately, changes often come with new errors that are introduced. FormulaDataSleuth [136] is a tool aimed to help the spreadsheet developer to immediately detect such errors when the spreadsheet is changed. Once the developer has specified which data areas and cells should be monitored by the tool, the system will automatically detect a number of potential problems. For the defined data areas, the tool can for example detect empty cells or input values that have a wrong data type or exceed the predefined range of allowed values. For monitored formula cells, accidentally overwritten formulas as well as range changes leading to wrong references can be identified. The authors demonstrate the usefulness of their approach by means of a running example. A deeper experimental investigation is however missing.

Understanding how a given spreadsheet evolved over time and seeing the difference between versions of a spreadsheet is often important when a spreadsheet is reused in a different project. In [137], Chambers et al. propose the *SheetDiff* algorithm, which is capable of detecting and visualizing certain types of non-trivial differences between two versions of a spreadsheet. To evaluate the approach, a number of spreadsheets from the EUSES corpus were selected. Some of them were considered to be modified versions of each other. For a number of additional spreadsheets, pre-defined change types (e.g., row insertion) were applied. The proposed algorithm was then compared with two commercial products. As measures, the correct change detection rate and the compactness of the result presentation were used. The results indicate that the new method is advantageous when compared with existing tools.

Later on, Harutyunyan et al. in [138] proposed a dynamic-programming based algorithm for difference detection called *RowColAlign*, which addressed existing problems of the greedy *SheetDiff* procedure described above. Instead of relying on manually selected or modified spreadsheets, a parameterizable test case generation technique was chosen, which allowed the authors to evaluate their method in a more systematic way.

9.4. Refactoring

Refactoring is defined as the process of changing the internal structure of a program without changing the functionality [139]. Refactoring contributes to the quality of spreadsheets in different ways, for example, by simplifying formulas and thus making them easier to understand, and by removing duplicate code thereby supporting easier and less error-prone maintenance. Refactoring in the context of spreadsheets is often concerned with the rearrangement of the columns and rows, i.e., the transformation of the design of the spreadsheet. Doing this transformation manually can be both time-intensive and prone to errors. Accordingly, different proposals have been made in the literature to automate this quality-improving maintenance task and to thereby prevent the introduction of new errors.

Badame and Dig [140] identify seven refactoring measures for spreadsheets and provide a corresponding plug-in for Microsoft Excel called RefBook. The plugin automatically detects the locations for which refactoring is required and supports the user in the refactoring process. Examples for possible refactoring steps include “Make Cell Constant”, “Guard Cell”, or “Replace Awkward Formula”. Badame and Dig evaluated their approach in different ways. In a survey involving 28 Excel users, the users preferred the refactored formula versions. In addition, a controlled lab experiment showed that people introduce faults during manual refactoring which could be avoided through automation. A retrospective analysis of spreadsheets from the EUSES corpus was finally done to validate the applicability of refactoring operators for real-world spreadsheets.

Harris and Gulwani [141] present an approach that supports complicated table transformations using user-specified examples. Their approach is based on a language for describing table transformations, called *TableProg*, and the algorithm *ProgFromEx* that takes as input a small example of the current spreadsheet and desired output spreadsheet. *ProgFromEx* automatically infers a program that implements the desired transformation. In an empirical evaluation, Harris and Gulwani applied their algorithm to 51 pairs of spreadsheet examples taken from online help forums for spreadsheets. This empirical evaluation proved that the required transformation programs could be generated for all example spreadsheets. However, sometimes a more detailed example spreadsheet than the one provided by the users was necessary.

In principle, the *Excelsior* tool mentioned in Section 8 is also suited to support spreadsheet restructuring tasks [111]. *Excelsior* supports flip and resize operations for tables. In addition, users can create several variants of a given spreadsheet. In [111], a case study was performed using one spreadsheet with several thousand cells to show the general feasibility of the approach. The depth of the evaluation thus considerably differs from the other refactoring approaches discussed in this section, which rely both on user studies and analyses based on real-world spreadsheets.

9.5. Reuse

In general, reusing existing and already validated software artefacts saves time, avoids the risk of making faults and supports maintainability [142]. This obviously also applies for spreadsheet development projects. Individual spreadsheets or parts of them are often reused in other projects. At a micro-level, even individual formulas are often used several times within a single spreadsheet. The standard solution for the reuse of formulas is to simply copy and paste the formulas. However, changing the original formula does not change its copies and forgotten updates of copied formulas thus can easily lead to faults.

The problem of reuse within spreadsheet programs was addressed by Djang and Burnett [143] and by Montigel [144]. In the approach of Djang and Burnett [143], reuse is mainly achieved through the concept of inheritance, a reuse approach that is common in object-oriented programming. Their "similarity inheritance" approach is however specifically designed to match the spreadsheet paradigm. In principle, it allows the developer to specify dependencies between (copied) spreadsheet cells in the form of multiple and mutual inheritance both on the level of individual cells and on a more coarse-grained level. The approach is illustrated based on a number of examples; an empirical evaluation is mentioned as an important next step.

Montigel [144] proposes the spreadsheet language *Wizcell*. In particular, *Wizcell* aims at facilitating reuse by making the possible semantics of copy

& paste and drag & drop actions more explicit. In particular, he sees four possible outcomes of such actions: (1) Either the copied formula is duplicated or there is a reference to the original formula. (2) Either the formulas in the copied cells refer to the cells mentioned in the original cells, or the references are changed according to the relative distance of the copy and the original. The proposed *Wizcell* language correspondingly allows the developer to specify the intended semantics, thus reducing the probability of introducing a fault. Similar to the reuse approach presented in [143], no report on an empirical evaluation is provided in [144].

9.6. Discussion

Many of the techniques and approaches presented in this section adapt existing techniques from traditional Software Engineering to the spreadsheet domain. In some cases, the authors explicitly address the problem that the basic spreadsheet development paradigm should not be changed too much and that the comprehensibility for the end user has to be maintained. However, some approaches require that the developer has a certain understanding of non-trivial programming concepts. As end users are usually non-professional programmers, the question of the applicability in practice arises.

Excelsior [111], for example, requires the user to understand concepts from logic programming. Djang and Burnett [143] build their work upon the concept of inheritance. While this term might not be used in the tool and these details are hidden from the UI through a visual representation, understanding the underlying semantics might be important for the developer to use the tools properly. NOPumpG [130, 131] and Action Graphics [132] use the concept of variables, which might not be known to a spreadsheet user. It therefore remains partially open whether all of these approaches are suited for end users without programming experience even if comparably simple visual representations are used.

The exception handling approach of Burnett et al. [134] requires no extended programming skills (except for example simple if-then-else constructs). Also Harris and Gulwani [141] consider the often limited capabilities of spreadsheet developers in their method and propose an example-based approach. Badame and Dig [140] rely on a semi-automatic approach and a plug-in to a wide-spread tool like MS Excel.

10. Discussion of current research practice

Our review showed that the way the different approaches from the literature are evaluated varies strongly. This can be partially attributed to the fact that research is carried out in different sub-fields of Computer Science as well as in Information Systems, each having their own standards and protocols.

The following major types of evaluation approaches can be found in the literature.

1. User studies: The proposed techniques and tools were evaluated in laboratory or field studies.
2. Empirical studies without users: The approaches were empirically evaluated, e.g., by applying them on operational spreadsheets or spreadsheets containing artificially injected errors. Such forms of evaluation can for example show that certain types of faults will be found when applying a given method, e.g., [88].
3. Theoretical analyses: Some researchers show by means of theoretical analyses that their approaches prevent certain types of errors, e.g., reference errors [114, 118].
4. No systematic evaluation: In some sub-areas and in particular for some older proposals, the evaluation was limited to an informal discussion based on example problems, based on unstructured feedback from a small group of users, or there was no real evaluation done at all.

Traditionally, research in various sub-fields of Computer Science is often based on offline experimental designs and simulations, whereas user studies are more common in Information Systems research, see, e.g., [145] for a review of evaluation approaches in the area of recommendation systems. In more recent proposals in particular from the Computer Science field, which is the focus of this work, theoretical or simulation-based analyses are now more often complemented with laboratory studies, e.g., in [60] or [88].

Generally, while we observe improvements with respect to research rigor and more systematic evaluations over the last years, in our view the research practice in the field can be further improved in different aspects.

10.1. Challenges of empirical evaluation approaches without users

The sample data sets used in offline experimental designs are often said to be (randomly) taken from the huge and very diverse EUSES corpus. Which documents were actually chosen and which additional criteria were applied is often not well justified. The choice can be influenced for example by the scalability of the proposed method or simply by the capabilities of some parser. Other factors that may influence the observed “success” of a new method can be the types or positions of the injected errors. These aspects are often not well documented and even when the benchmark problems are made publicly available as in [87], they may have special characteristics that are required or advantageous for a given method and, e.g., contain only one single fault or use only a restricted set of functions or cell data types.

We therefore argue that researchers should report in more detail about the basis of their evaluations. Otherwise, comparative evaluations are not easily possible in the field, in particular as source codes or the developed Excel plug-ins are usually not shared in the community. Even though different types of spreadsheets might be required for the different research proposals, one future goal could therefore be to develop a set of defined *benchmark spreadsheets*. These can be used and adapted by the research community and serve as a basis for comparative evaluations, which are barely found in current spreadsheet literature.

10.2. Challenges of doing user studies

The more recent works in the field often include reports on different types of laboratory studies to assess, for example, if users are actually capable of using a new tool or, more importantly, if the tool actually helps the users in the fault identification or removal process. Such studies can be considered to be the main evaluation instrument in IS research and the typical experimental designs of such studies include tasks like code inspection and fault localization, error detection and removal, and formula or spreadsheet construction.

Conducting reliable user studies, which are done usually in laboratory settings, is in general a challenging task even though various standard designs, procedures, and statistical analysis methods exist that are also common, e.g., in sociobehavioral sciences [146]. A discussion of general properties of valid experimental designs is beyond the scope of this work. However, in our review we observed some typical limitations in the context of spreadsheet research.

First, the number of participants in each “treatment group” – e.g., one group with and another group without tool assistance – is often quite small. Various ways including statistical power analysis exist to determine the minimum number of participants, which can however depend on the goal and type of the study, the statistical significance criterion used, or the desired confidence level and interval. Typical sample sizes in the literature are for example 61 participants assigned in two groups [147] or 90 participants that were distributed to two groups of different sizes [148].

Additional questions in that context are whether the study participants are representative for a larger population of spreadsheet users – in [149], students are considered as good surrogates – and how it can be made sure that the participants are correctly assigned to the different groups, e.g., based on their experience or a random procedure. Finally, the question arises if doing the experiment in a laboratory setting is not introducing a bias making the evaluation unrealistic. As for the latter aspect, also studies exist in which the participants accomplish the tasks at home [150]. In these cases, it is however easier for the participants to cheat. In particular for spreadsheet construction exercises, it has to be considered that the developed spreadsheets can be quite different from real operational spreadsheets, e.g., with respect to their complexity [11].

10.3. General remarks

In general, both for user studies and offline experiments in which we use artificially injected errors, the problem exists that we cannot be sure that the introduced types of faults are always representative or realistic. While a number of studies on error rates exist, Powell et al. [11] argue that it is often unclear which fault categorization scheme was used or how faults were counted that were corrected during the construction of the spreadsheet. It can thus be dangerous to make inferences about the general efficacy of a method if it was only evaluated on certain types of faults.

Field studies based on operational spreadsheets and real spreadsheet developers would obviously represent a valuable means to assess the true efficacy, e.g., of a certain fault reduction approach. Such reports are however rare as they are costly to conduct. The work presented in [148] is an example of such a study, in which experienced business managers participated and accomplished a spreadsheet construction exercise. In such settings, however, additional problems arise, e.g., that the participants could not be assigned to different treatment groups randomly as their geographical location had to be taken into account.

Finally, as in many other research fields, experimental studies are barely reproduced by other research groups to validate the findings. In addition, the reliability of the reported results can be low, e.g., because of biases by the researchers, weak experimental designs, or questions of the interpretation of the outcomes of statistical tests [151, 152].

Overall, the evaluation of tools and techniques to localize and remove faults in spreadsheets remains a challenging task as it not only involves algorithmic questions but at the same time has to be usable by people with a limited background in IT. In many cases, a comprehensive evaluation approach is therefore required which combines the necessary theoretical analysis with user studies whose design should incorporate the insights from the existing works, e.g., in the area of IS research or Human Computer Interaction.

11. Perspectives for future works

The literature review has pointed out some interesting new fields of research for spreadsheet quality assurance. In the following, we will sketch a subjective selection of possible directions to future works. In the discussion, we will limit ourselves to broader topics and not focus on specific research opportunities within the different sub-areas.

11.1. Life cycle coverage

Our review shows that a number of proposals have been made to support the developer in various stages of the spreadsheet life cycle including application design and development, testing, debugging, maintenance, comprehension and reuse. For the early development phases – like domain analysis, requirements specification and the initial design – we have however not found any proposals for automated tool support. Ko et al. in [153] argue that these early phases and tasks are mostly not explicitly executed in typical spreadsheet development activities, or more generally, end user programming scenarios. In their work, a detailed discussion and analysis of general differences between professional software engineering processes and end user software engineering can be found. Regarding requirements specification, Ko et al. for example mention that in contrast to professional software development, the source of the requirements is the same person as the programmer, e.g., because people often develop spreadsheets for themselves. With respect to design processes, one assumption is that end user programmers might not see the benefits of making the design an explicit step when translating the requirements into a program.

How to provide better tool support for the very early phases – which should ultimately lead to higher-quality spreadsheets in the end – is in our view largely open. Such approaches probably have to be accompanied with organizational measures and additional training for the end user programmer to raise the awareness of the advantages of a more structured development process, even if this process is exploratory and prototyping-based in nature. Alternative development approaches such as Example-Driven Modeling [154] or programming by example could be explored as well.

Beside tool support for the early development phases, we see a number of other areas where existing quality-ensuring or quality-improving techniques can be applied or further adapted to the spreadsheet domain. This includes better quality metrics, formal analysis methods, or techniques for spreadsheet evolution, versioning and "product lines", which in our view have not been explored deeply enough so far.

11.2. Combination of methods

We see a lot of potential for further research in the area of combining different specific techniques in hybrid systems. In [86], for example, the authors propose methods to combine the feedback of the UCheck type checking system with the results from the WYSIWYT fault localization technique based on heuristics. An evaluation using various mutations of a spreadsheet showed that the combination is advantageous, e.g., because different types of faults can be detected by the two techniques. Other works that integrate different types of information or reasoning strategies include [57, 54, 104] and more recently [87], who combine declarative debugging with trace-based candidate ranking.

Beside the integration of methods to fulfill one particular task, one possible direction of future research is to explore alternative hybridization designs, e.g., to combine methods in a sequential or parallel manner. In such a scenario, one computationally cheap method could be used to identify larger regions in the spreadsheet which most probably contain an error. More sophisticated and computationally demanding techniques could afterwards be applied within this local area to determine the exact location of the problem. Alternatively, there might be situations in which multiple techniques are available for a certain task, e.g., to rank the error candidates. Whether or not a specific technique works well for a given problem setting depends on a number of factors including the structure and the size of the spreadsheets or the types of the formulas. A possible future research direction could therefore lie in the development of algorithms which – based on heuristics, past observations, and a concise characterization of the capabilities and requirements of the different techniques – can automatically assess which of the available techniques will be the most promising given a specific spreadsheet and task.

11.3. Toward integrated user-friendly tools

Individual research efforts often aim at one particular problem, for example test support and test case management, propose one particular technique and focus on one single optimization criterion such as maximizing the test coverage. While keeping the work focused is appropriate in the context of individual scientific contributions, in reality, the different QA tasks are often related: a debugging activity, for example, can be initiated by a test activity or a maintenance task. Therefore, to be applicable in practice, one of the goals of future research is to better understand how integrated tools should be designed that support the developer in the best possible way. Such a research could for example include the discussion of suitable user interface (UI) designs, the choice of comprehensible terminology and metaphors, the question of the appropriate level of user guidance, the choice of adequate supporting visualizations, or even questions of how to integrate the tools smoothly into existing spreadsheet environments.

An example for such an end-user oriented interaction pattern for spreadsheets can be found in [80]. Using a so-called "surprise-explain-reward strategy", the goal of the work is to entice the user to make increased use of the assertion feature of the spreadsheet environment without requiring the user to change his or her usual work process. This is accomplished by automatically generating assertions about cell contents, presenting violations in the form of passive feedback, and then relying on the user's curiosity to explore the potential problems. Beckwith et al. later on continued this work in [155] and investigated gender-specific differences in the adoption of such new tool features and proposed different variations of the UI for risk-averse or low confidence users. Finally, another work that builds on psychological phenomena to increase tool adoption (and effectiveness) is presented in [156]. In this work, the authors focus on the role of the *perceivable rewards* and experiment with UI variants in which the tool's functionality is identical but the visual feedback, e.g., in terms of cell coloring is varied.

11.4. Toward a formal spreadsheet language

In the literature, a number of different *intermediate* representations are used to formally and precisely describe the logic of a given spreadsheet application. Some of these are based on standard formalisms with defined semantics including logic- and constraint-based approaches [110, 88, 99]; other papers introduce their own formalisms supporting a specific methodology or various forms of reasoning on it [85].

In particular in the latter cases, a precise definition of what can be expressed in these intermediate representations is sometimes missing, for example, if it is possible to reason about real-valued calculations or which of the more complex functions of systems like MS Excel can be expressed when using a certain intermediate representation.

In order to be able to better compare and combine different spreadsheet QA techniques in hybrid approaches as discussed above, a unified formal spreadsheet representation, problem definition language, or even a “theory of spreadsheets” could be useful. It would furthermore help making research efforts independent of specific environments or tool versions and at the same time allow for formal reasoning, e.g., about the soundness and completeness of individual fault localization techniques. Such problem definition languages are for example common in other domains such as Artificial Intelligence based planning or Constraint Satisfaction.

11.5. Provision of better abstraction mechanisms

In [157], Peyton Jones et al. argue that spreadsheets in their basic form can be considered as functional programs that only consist of statements comprising built-in functions. Thus, spreadsheet developers have no means to define reusable *abstractions* in the form of parameterizable functions. To implement the desired functionality, users therefore have to copy the formulas multiple times, which however leads to poor maintainability and lower spreadsheet quality. As a potential solution, the authors propose a user-oriented approach to design user-defined functions. A main goal of the design is to stay within the spreadsheet paradigm, which for example means that the function implementations should be specified as spreadsheets (“function sheets”) and not in the form of imperative programs as done in MS Excel. The work presented in [157] was mostly based on theoretical considerations. In their evaluation, the authors mainly focus on the expressiveness of the language and performance aspects. So far, no evaluation investigating if users are able to understand the concepts of how to define function sheets or to interpret error messages has been done.

Later on, Sestoft [158] presented a practical realization of the approach that includes recursion and higher-order functions. To design and use new function sheets in their prototype system called "Funcalc", the spreadsheet developer has to learn only three new built-in functions.

Both function sheets and the more recent ClassSheets as described in Section 8.3 represent approaches to empower spreadsheet developers with better abstraction mechanisms within the spreadsheet paradigm. As a result, these approaches should help users avoid making different types of faults and increase the general quality of the spreadsheets. Overall, we see the provision of such advanced concepts for spreadsheet design and implementation as a promising area for future research, where in particular the questions of understandability for the end user should be further investigated.

12. Summary

Errors in spreadsheet programs can have a huge impact on organizations. Unfortunately, current spreadsheet environments like MS Excel only include limited functionality to help developers create error-free spreadsheets or support them in the error detection and localization process. Over the last decades, researches in different subfields of Computer Science and Information Systems have therefore made a substantial number of proposals aimed at better tool support for spreadsheet developers during the development lifecycle.

With our literature review and the presented classification scheme we aimed to provide a basis to structure and relate the different strands of research in this area and critically reflected on current research practices. At the same time, the review and classification scheme should help to identify potential directions for future research and opportunities for combining different proposals, thereby helping to move from individual techniques and tools to integrated spreadsheet QA environments.

Acknowledgements

This work was partially supported by the European Union through the programme "Europäischer Fonds für regionale Entwicklung - Investition in unsere Zukunft" under contract number 300251802.

References

[1] R. R. Panko, D. N. Port, End User Computing: The Dark Matter (and Dark Energy) of Corporate IT, in: Proceedings of the 45th Hawaii

International Conference on System Sciences (HICSS 2012), Wailea, HI, USA, 2012, pp. 4603–4612.

[2] R. Creeth, Micro-Computer Spreadsheets: Their Uses and Abuses, Journal of Accountancy 159 (6) (1985) 90–93.

[3] S. Ditlea, Spreadsheets can be hazardous to your health, Personal Computing 11 (1) (1987) 60–69.

[4] R. R. Panko, What We Know About Spreadsheet Errors, Journal of End User Computing 10 (2) (1998) 15–21.

[5] T. Herndon, M. Ash, R. Pollin, Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff, Working Paper 322, Political Economy Research Institute, University of Massachusetts, Amherst (April 2013).

[6] C. M. Reinhart, K. S. Rogoff, Growth in a Time of Debt, American Economic Review 100 (2) (2010) 573–578.

[7] D. F. Galletta, D. Abraham, M. E. Louadi, W. Lekse, Y. A. Pollalis, J. L. Sampler, An empirical study of spreadsheet error-finding performance, Accounting, Management and Information Technologies 3 (2) (1993) 79–95.

[8] S. Thorne, A review of spreadsheet error reduction techniques, Communications of the Association for Information Systems 25, Article 24.

[9] T. Reinhardt, N. Pillay, Analysis of Spreadsheet Errors Made by Computer Literacy Students, in: Proceedings of the IEEE International Conference on Advanced Learning Technologies (ICALT 2004), Joensuu, Finland, 2004, pp. 852–853.

[10] D. F. Galletta, K. S. Hartzel, S. E. Johnson, J. L. Joseph, S. Rustagi, Spreadsheet Presentation and Error Detection: An Experimental Study, Journal of Management Information Systems 13 (3) (1996) 45–63.

[11] S. G. Powell, K. R. Baker, B. Lawson, A critical review of the literature on spreadsheet errors, Decision Support Systems 46 (1) (2008) 128–138.

[12] H. Howe, M. G. Simkin, Factors Affecting the Ability to Detect Spreadsheet Errors, Decision Sciences Journal of Innovative Education 4 (1) (2006) 101–122.

[13] J. R. Olson, E. Nilsen, Analysis of the Cognition Involved in Spreadsheet Software Interaction, Human-Computer Interaction 3 (4) (1987) 309–349.

[14] G. Rothermel, L. Li, C. Dupuis, M. Burnett, What You See Is What You Test: A Methodology for Testing Form-Based Visual programs, in: Proceedings of the 20th International Conference on Software Engineering (ICSE 1998), Kyoto, Japan, 1998, pp. 198–207.

[15] R. R. Panko, R. P. Halverson, Spreadsheets on Trial: A Survey of Research on Spreadsheet Risks, in: Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS 1996), Wailea, HI, USA, 1996, pp. 326–335.

[16] K. Rajalingham, D. R. Chadwick, B. Knight, Classification of Spreadsheet Errors, in: Proceedings of the European Spreadsheet Risks Interest Group 2nd Annual Conference (EuSpRIG 2001), Amsterdam, Netherlands, 2001.

[17] R. R. Panko, S. Aurigemma, Revising the Panko-Halverson taxonomy of spreadsheet errors, Decision Support Systems 49 (2) (2010) 235–244.

[18] M. Erwig, Software engineering for spreadsheets, IEEE Software 26 (5) (2009) 25–30.

[19] J. Davis, Tools for spreadsheet auditing, International Journal of Human-Computer Studies 45 (4) (1996) 429–442.

[20] J. Sajaniemi, Modeling Spreadsheet Audit: A Rigorous Approach to Automatic Visualization, Journal of Visual Languages & Computing 11 (1) (2000) 49–82.

[21] T. Igarashi, J. Mackinlay, B.-W. Chang, P. Zellweger, Fluid Visualization of Spreadsheet Structures, in: Proceedings of the IEEE Symposium on Visual Languages (VL 1998), Halifax, NS, Canada, 1998, pp. 118–125.

[22] H. Shiozawa, K. Okada, Y. Matsushita, 3D Interactive Visualization for Inter-Cell Dependencies of Spreadsheets, in: Proceedings of the IEEE Symposium on Information Visualization (Info Vis 1999), San Francisco, CA, USA, 1999, pp. 79–82, 148.

[23] Y. Chen, H. C. Chan, Visual Checking of Spreadsheets, in: Proceedings of the European Spreadsheet Risks Interest Group 1st Annual Conference (EuSpRIG 2000), London, United Kingdom, 2000.

[24] D. Ballinger, R. Biddle, J. Noble, Spreadsheet visualisation to improve end-user understanding, in: Proceedings of the Asia-Pacific Symposium on Information Visualisation - Volume 24 (APVIS 2003), Adelaide, Australia, 2003, pp. 99–109.

[25] K. Hodnigg, R. T. Mittermeir, Metrics-Based Spreadsheet Visualization: Support for Focused Maintenance, in: Proceedings of the European Spreadsheet Risks Interest Group 9th Annual Conference (EuSpRIG 2008), London, United Kingdom, 2008, pp. 79–94.

[26] B. Kankuzi, Y. Ayalew, An End-User Oriented Graph-Based Visualization for Spreadsheets, in: Proceedings of the 4th International Workshop on End-User Software Engineering (WEUSE 2008), Leipzig, Germany, 2008, pp. 86–90.

[27] Y. Ayalew, A Visualization-based Approach for Improving Spreadsheet Quality, in: Proceedings of the Warm Up Workshop for ACM/IEEE ICSE 2010 (WUP 2009), Cape Town, South Africa, 2009, pp. 13–16.

[28] F. Hermans, M. Pinzger, A. van Deursen, Supporting Professional Spreadsheet Users by Generating Leveled Dataflow Diagrams, in: Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011), Waikiki, Honolulu, HI, USA, 2011, pp. 451–460.

[29] F. Hermans, M. Pinzger, A. van Deursen, Breviz: Visualizing Spreadsheets using Dataflow Diagrams, in: Proceedings of the European Spreadsheet Risks Interest Group 12th Annual Conference (EuSpRIG 2011), London, United Kingdom, 2011.

[30] B. Ronen, M. A. Palley, H. C. Lucas, Jr., Spreadsheet Analysis and Design, Communications of the ACM 32 (1) (1989) 84–93.

[31] R. Mittermeir, M. Clermont, Finding High-Level Structures in Spreadsheet Programs, in: Proceedings of the 9th Working Conference on Reverse Engineering (WCRE 2002), Richmond, VA, USA, 2002, pp. 221–232.

[32] S. Hipfl, Using Layout Information for Spreadsheet Visualization, in: Proceedings of the European Spreadsheet Risks Interest Group 5th Annual Conference (EuSpRIG 2004), Klagenfurt, Austria, 2004.

[33] M. Clermont, Analyzing Large Spreadsheet Programs, in: Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003), Victoria, BC, Canada, 2003, pp. 306–315.

[34] M. Clermont, A Toolkit for Scalable Spreadsheet Visualization, in: Proceedings of the European Spreadsheet Risks Interest Group 5th Annual Conference (EuSpRIG 2004), Klagenfurt, Austria, 2008.

[35] M. Clermont, Heuristics for the Automatic Identification of Irregularities in Spreadsheets, in: Proceedings of the 1st Workshop on End-User Software Engineering (WEUSE 2005), St. Louis, MO, USA, 2005, pp. 1–6.

[36] M. Clermont, C. Hanin, R. T. Mittermeir, A Spreadsheet Auditing Tool Evaluated in an Industrial Context, in: Proceedings of the European Spreadsheet Risks Interest Group 3rd Annual Conference (EuSpRIG 2002), Cardiff, United Kingdom, 2002.

[37] D. Hendry, T. Green, CogMap: a Visual Description Language for Spreadsheets, Journal of Visual Languages & Computing 4 (1) (1993) 35–54.

[38] T. Isakowitz, S. Schocken, H. C. Lucas, Jr., Toward a Logical / Physical Theory of Spreadsheet Modeling, Transactions on Information Systems 13 (1) (1995) 1–37.

[39] M. Lentini, D. Nardi, A. Simonetta, Self-instructive spreadsheets: an environment for automatic knowledge acquisition and tutor generation, International Journal of Human-Computer Studies 52 (5) (2000) 775–803.

[40] D. Chadwick, B. Knight, K. Rajalingham, Quality Control in Spreadsheets: A Visual Approach using Color Codings to Reduce Errors in Formulae, Software Quality Control 9 (2) (2001) 133–143.

[41] D. Nardi, G. Serrecchia, Automatic Generation of Explanations for Spreadsheet Applications, in: Proceedings of the 10th Conference on Artificial Intelligence for Applications (CAIA 1994), San Antonio, TX, USA, 1994, pp. 268–274.

[42] R. Brath, M. Peters, Excel Visualizer: One Click WYSIWYG Spreadsheet Visualization, in: Proceedings of the 10th International Conference on Information Visualisation (IV 2006), London, United Kingdom, 2006, pp. 68–73.

[43] R. Rao, S. K. Card, The Table Lens: Merging Graphical and Symbolic Representations in an Interactive Focus+Context Visualization for Tabular Information, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 1994), Boston, MA, USA, 1994, pp. 318–322.

[44] P. S. Brown, J. D. Gould, An Experimental Study of People Creating Spreadsheets, ACM Transactions on Information Systems 5 (3) (1987) 258–272.

[45] S. Aurigemma, R. R. Panko, The Detection of Human Spreadsheet Errors by Humans versus Inspection (Auditing) Software, in: Proceedings of the European Spreadsheet Risks Interest Group 11th Annual Conference (EuSpRIG 2010), London, United Kingdom, 2010.

[46] M. Erwig, M. M. Burnett, Adding Apples and Oranges, in: Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages (PADL 2002), Portland, OR, USA, 2002, pp. 173–191.

[47] M. Burnett, M. Erwig, Visually Customizing Inference Rules About Apples and Oranges, in: Proceedings of the IEEE Symposia on Human Centric Computing Languages and Environments (HCC 2002), Arlington, VA, USA, 2002, pp. 140–148.

[48] Y. Ahmad, T. Antoniu, S. Goldwater, S. Krishnamurthi, A Type System for Statically Detecting Spreadsheet Errors, in: Proceedings of the

18th IEEE/ACM International Conference on Automated Software Engineering (ASE 2003), Montreal, Canada, 2003, pp. 174–183.

[49] R. Abraham, M. Erwig, Header and Unit Inference for Spreadsheets Through Spatial Analyses, in: Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC 2004), Rome, Italy, 2004, pp. 165–172.

[50] T. Antoniu, P. Steckler, S. Krishnamurthi, E. Neuwirth, M. Felleisen, Validating the Unit Correctness of Spreadsheet Programs, in: Proceedings of the 26th International Conference on Software Engineering (ICSE 2004), Edinburgh, United Kingdom, 2004, pp. 439–448.

[51] R. Abraham, M. Erwig, Type Inference for Spreadsheets, in: Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2006), Venice, Italy, 2006, pp. 73–84.

[52] R. Abraham, M. Erwig, UCheck: A Spreadsheet Type Checker for End Users, Journal of Visual Languages & Computing 18 (1) (2007) 71–95.

[53] C. Chambers, M. Erwig, Automatic Detection of Dimension Errors in Spreadsheets, Journal of Visual Languages & Computing 20 (4) (2009) 269–283.

[54] C. Chambers, M. Erwig, Reasoning About Spreadsheets with Labels and Dimensions, Journal of Visual Languages & Computing 21 (5) (2010) 249–262.

[55] M. J. Coblenz, A. J. Ko, B. A. Myers, Using objects of measurement to detect spreadsheet errors, in: Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC 2005), 2005, pp. 314–316.

[56] C. Chambers, M. Erwig, Dimension Inference in Spreadsheets, in: Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC 2008), Herrsching am Ammersee, Germany, 2008, pp. 123–130.

[57] C. Chambers, M. Erwig, Combining Spatial and Semantic Label Analysis, in: Proceedings of the IEEE Symposium on Visual Languages

and Human Centric Computing (VL/HCC 2009), Corvallis, OR, USA, 2009, pp. 225–232.

[58] M. Fisher, G. Rothermel, The EUSES Spreadsheet Corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms, SIGSOFT Software Engineering Notes 30 (4) (2005) 1–5.

[59] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 1999.

[60] F. Hermans, M. Pinzger, A. van Deursen, Detecting and Visualizing Inter-Worksheet Smells in Spreadsheets, in: Proceedings of the 34th International Conference on Software Engineering (ICSE 2012), Zurich, Switzerland, 2012, pp. 441–451.

[61] F. Hermans, M. Pinzger, A. van Deursen, Detecting Code Smells in Spreadsheet Formulas, in: Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM 2012), Riva del Garda, Trento, Italy, 2012, pp. 409–418.

[62] F. Hermans, B. Sedee, M. Pinzger, A. v. Deursen, Data Clone Detection and Visualization in Spreadsheets, in: Proceedings of the 35th International Conference on Software Engineering (ICSE 2013), San Francisco, CA, USA, 2013, pp. 292–301.

[63] J. Cunha, J. a. P. Fernandes, H. Ribeiro, J. a. Saraiva, Towards a Catalog of Spreadsheet Smells, in: Proceedings of the 12th International Conference on Computational Science and Its Applications (ICCSA 2012), Salvador de Bahia, Brazil, 2012, pp. 202–216.

[64] D. Nixon, M. O'Hara, Spreadsheet Auditing Software, in: Proceedings of the European Spreadsheet Risks Interest Group 2nd Annual Conference (EuSpRIG 2001), Amsterdam, Netherlands, 2001.

[65] J. Hunt, An approach for the automated risk assessment of structural differences between spreadsheets (diffxl), in: Proceedings of the European Spreadsheet Risks Interest Group 10th Annual Conference (EuSpRIG 2009), Paris, France, 2009.

[66] R. Abraham, M. Erwig, S. Andrew, A type system based on end-user vocabulary, in: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007), Coeur d'Alene, Idaho, USA, 2007, pp. 215–222.

[67] G. Rothermel, L. Li, M. Burnett, Testing Strategies for Form-Based Visual Programs, in: Proceedings of the 8th International Symposium on Software Reliability Engineering (ISSRE 1997), Albuquerque, NM, USA, 1997, pp. 96–107.

[68] G. Rothermel, M. Burnett, L. Li, C. Dupuis, A. Sheretov, A Methodology for Testing Spreadsheets, ACM Transactions on Software Engineering and Methodology 10 (1) (2001) 110–147.

[69] M. Burnett, A. Sheretov, G. Rothermel, Scaling Up a "What You See Is What You Test" Methodology to Spreadsheet Grids, in: Proceedings of the IEEE Symposium on Visual Languages (VL 1999), Tokyo, Japan, 1999, pp. 30–37.

[70] M. Burnett, A. Sheretov, B. Ren, G. Rothermel, Testing Homogeneous Spreadsheet Grids with the "What You See Is What You Test" Methodology, IEEE Transactions on Software Engineering 28 (6) (2002) 576–594.

[71] M. Burnett, B. Ren, A. Ko, C. Cook, G. Rothermel, Visually Testing Recursive Programs in Spreadsheet Languages, in: Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments (HCC 2001), Stresa, Italy, 2001, pp. 288–295.

[72] M. Fisher, II, D. Jin, G. Rothermel, M. Burnett, Test Reuse in the Spreadsheet paradigm, in: Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE 2003), Denver, CO, USA, 2002, pp. 257–268.

[73] M. Fisher, G. Rothermel, T. Creelan, M. Burnett, Scaling a Dataflow Testing Methodology to the Multiparadigm World of Commercial Spreadsheets, in: Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE 2006), Raleigh, NC, USA, 2006, pp. 13–22.

[74] N. Randolph, J. Morris, G. Lee, A Generalised Spreadsheet Verification Methodology, in: Proceedings of the 25th Australasian Conference on Computer Science (ACSC 2002), 2002, pp. 215–222.

[75] K. Rothermel, C. Cook, M. Burnett, J. Schonfeld, T. R. G. Green, G. Rothermel, WYSIWYT Testing in the Spreadsheet Paradigm: An Empirical Evaluation, in: Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, 2000, pp. 230–239.

[76] M. Fisher, II, M. Cao, G. Rothermel, C. Cook, M. Burnett, Automated Test Case Generation for Spreadsheets, in: Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), Orlando, FL, USA, 2002, pp. 141–151.

[77] M. Fisher, II, G. Rothermel, D. Brown, M. Cao, C. Cook, M. Burnett, Integrating Automated Test Generation into the WYSIWYT Spreadsheet Testing Methodology, ACM Transactions on Software Engineering and Methodology 15 (2) (2006) 150–194.

[78] R. Abraham, M. Erwig, AutoTest: A Tool for Automatic Test Case Generation in Spreadsheets, in: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2006), Brighton, United Kingdom, 2006, pp. 43–50.

[79] M. Burnett, C. Cook, O. Pendse, G. Rothermel, J. Summet, C. Wallace, End-User Software Engineering with Assertions in the Spreadsheet Paradigm, in: Proceedings of the 25th International Conference on Software Engineering (ICSE 2003), Portland, Oregon, 2003, pp. 93–103.

[80] A. Wilson, M. Burnett, L. Beckwith, O. Granatir, L. Casburn, C. Cook, M. Durham, G. Rothermel, Harnessing Curiosity to Increase Correctness in End-User Programming, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2003), 2003, pp. 305–312.

[81] L. Beckwith, M. Burnett, C. Cook, Reasoning about Many-to-Many Requirement Relationships in Spreadsheets, in: Proceedings of the

IEEE Symposia on Human Centric Computing Languages and Environments (HCC 2002), Arlington, VA, USA, 2002, pp. 149–157.

[82] K. McDaid, A. Rust, B. Bishop, Test-Driven Development: Can it Work for Spreadsheets?, in: Proceedings of the 4th International Workshop on End-User Software Engineering (WEUSE 2008), Leipzig, Germany, 2008, pp. 25–29.

[83] F. Hermans, Improving Spreadsheet Test Practices, in: Proceedings of the 23rd Annual International Conference on Computer Science and Software Engineering (CASCON 2013), Markham, Ontario, Canada, 2013, pp. 56–69.

[84] R. Abraham, M. Erwig, Mutation Operators for Spreadsheets, IEEE Transactions on Software Engineering 35 (1) (2009) 94–108.

[85] R. Abraham, M. Erwig, Goal-Directed Debugging of Spreadsheets, in: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2005), Dallas, TX, USA, 2005, pp. 37–44.

[86] R. A. Joseph Lawrance, Margaret Burnett, M. Erwig, Sharing reasoning about faults in spreadsheets: An empirical study, in: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2006), 2006, pp. 35–42.

[87] B. Hofer, A. Riboira, F. Wotawa, R. Abreu, E. Getzner, On the Empirical Evaluation of Fault Localization Techniques for Spreadsheets, in: Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE 2013), Rome, Italy, 2013, pp. 68–82.

[88] D. Jannach, T. Schmitz, Model-based diagnosis of spreadsheet programs - A constraint-based debugging approach, Automated Software Engineering to appear.

[89] J. Reichwein, G. Rothermel, M. Burnett, Slicing Spreadsheets: An Integrated Methodology for Spreadsheet Testing and Debugging, in: Proceedings of the 2nd Conference on Domain-Specific Languages (DSL 1999), Austin, Texas, 1999, pp. 25–38.

[90] J. R. Ruthruff, S. Prabhakararao, J. Reichwein, C. Cook, E. Creswick, M. Burnett, Interactive, Visual Fault Localization Support for End-User Programmers, Journal of Visual Languages & Computing 16 (1-2) (2005) 3–40.

[91] J. R. Ruthruff, M. Burnett, G. Rothermel, An Empirical Study of Fault Localization for End-User Programmers, in: Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), St. Louis, MO, USA, 2005, pp. 352–361.

[92] J. R. Ruthruff, M. Burnett, G. Rothermel, Interactive Fault Localization Techniques in a Spreadsheet Environment, IEEE Transactions on Software Engineering 32 (4) (2006) 213–239.

[93] Y. Ayalew, R. Mittermeir, Spreadsheet Debugging, in: Proceedings of the European Spreadsheet Risks Interest Group 4th Annual Conference (EuSpRIG 2003), Dublin, Ireland, 2003.

[94] D. Jannach, U. Engler, Toward model-based debugging of spreadsheet programs, in: Proceedings of the 9th Joint Conference on Knowledge-Based Software Engineering (JCKBSE 2010), Kaunas, Lithuania, 2010, pp. 252–264.

[95] E. Tsang, Foundations of Constraint Satisfaction, Academic Press, 1993.

[96] C. Mateis, M. Stumptner, D. Wieland, F. Wotawa, Model-Based Debugging of Java Programs, in: Proceedings of the Fourth International Workshop on Automated Debugging (AADEBUG 2000), Munich, Germany, 2000.

[97] A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner, Consistency-based diagnosis of configuration knowledge bases, Artificial Intelligence 152 (2) (2004) 213–234.

[98] D. Jannach, A. Baharloo, D. Williamson, Toward an integrated framework for declarative and interactive spreadsheet debugging, in: Procedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2013), Angers, France, 2013, pp. 117–124.

[99] R. Abreu, A. Riboira, F. Wotawa, Constraint-based Debugging of Spreadsheets, in: Proceedings of the 15th Ibero-American Conference on Software Engineering (CIbSE 2012), Buenos Aires, Argentina, 2012, pp. 1–14.

[100] R. Abreu, A. Riboira, F. Wotawa, Debugging Spreadsheets: A CSP-based Approach, in: Proceedings of the 23rd IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW 2012), Dallas, TX, USA, 2012, pp. 159–164.

[101] R. Reiter, A Theory of Diagnosis from First Principles, Artificial Intelligence 32 (1) (1987) 57–95.

[102] S. Außerlechner, S. Fruhmann, W. Wieser, B. Hofer, R. Spörk, C. Mühlbacher, F. Wotawa, The Right Choice Matters! SMT Solving Substantially Improves Model-Based Debugging of Spreadsheets, in: Proceedings of the 13th International Conference on Quality Software (QSIC 2013), Nanjing, China, 2013, pp. 139–148.

[103] R. Abraham, M. Erwig, GoalDebug: A Spreadsheet Debugger for End Users, in: Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, 2007, pp. 251–260.

[104] R. Abraham, M. Erwig, Test-driven goal-directed debugging in spreadsheets, in: Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC 2008), Herrsching am Ammersee, Germany, 2008, pp. 131–138.

[105] C. Parnin, A. Orso, Are Automated Debugging Techniques Actually Helping Programmers?, in: Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA 2011), Toronto, Canada, 2011, pp. 199–209.

[106] M. Luckey, M. Erwig, G. Engels, Systematic Evolution of Model-Based Spreadsheet Applications, Journal of Visual Languages & Computing 23 (5) (2012) 267–286.

[107] J. Paine, Model Master: an object-oriented spreadsheet front-end, in: Proceedings of the CALECO Conference on Using Computer Technology in Economics and Business (CALECO 1997), Bristol, United Kingdom, 1997, pp. 84–92.

[108] J. Paine, Ensuring Spreadsheet Integrity with Model Master, in: Proceedings of the European Spreadsheet Risks Interest Group 2nd Annual Conference (EuSpRIG 2001), Amsterdam, Netherlands, 2001.

[109] J. Paine, Spreadsheet Structure Discovery with Logic Programming, in: Proceedings of the European Spreadsheet Risks Interest Group 5th Annual Conference (EuSpRIG 2004), Klagenfurt, Austria, 2004.

[110] J. Paine, Excelsior: Bringing the Benefits of Modularisation to Excel, in: Proceedings of the European Spreadsheet Risks Interest Group 6th Annual Conference (EuSpRIG 2005), London, United Kingdom, 2005.

[111] J. Paine, E. Tek, D. Williamson, Rapid Spreadsheet Reshaping with Excelsior: multiple drastic changes to content and layout are easy when you represent enough structure, in: Proceedings of the European Spreadsheet Risks Interest Group 7th Annual Conference (EuSpRIG 2006), Cambridge, United Kingdom, 2006.

[112] M. Erwig, R. Abraham, I. Cooperstein, S. Kollmansberger, Automatic Generation and Maintenance of Correct Spreadsheets, in: Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), St. Louis, MO, USA, 2005, pp. 136–145.

[113] R. Abraham, M. Erwig, S. Kollmansberger, E. Seifert, Visual Specifications of Correct Spreadsheets, in: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2005), Dallas, TX, USA, 2005, pp. 189–196.

[114] M. Erwig, R. Abraham, S. Kollmansberger, I. Cooperstein, Gencel: A Program Generator for Correct Spreadsheets, Journal of Functional Programming 16 (3) (2006) 293–325.

[115] R. Abraham, M. Erwig, Inferring Templates from Spreadsheets, in: Proceedings of the 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, 2006, pp. 182–191.

[116] T. R. G. Green, M. Petre, Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework, Journal of Visual Languages & Computing 7 (2) (1996) 131–174.

[117] A. Blackwell, T. R. G. Green, Notational Systems – the Cognitive Dimensions of Notations Framework, HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science (2003) 103–134.

[118] G. Engels, M. Erwig, ClassSheets: Automatic Generation of Spreadsheet Applications from Object-Oriented Specifications, in: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), Long Beach, CA, USA, 2005, pp. 124–133.

[119] J. Cunha, M. Erwig, J. Saraiva, Automatically Inferring ClassSheet Models from Spreadsheets, in: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2010), Madrid, Spain, 2010, pp. 93–100.

[120] J. Cunha, J. Visser, T. Alves, J. a. Saraiva, Type-Safe Evolution of Spreadsheets, in: Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering: Part of the Joint European Conferences on Theory and Practice of Software (FASE 2011/ETAPS 2011), Saarbrücken, Germany, 2011, pp. 186–201.

[121] J. Cunha, J. Mendes, J. Saraiva, J. Fernandes, Embedding and Evolution of Spreadsheet Models in Spreadsheet Systems, in: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2011), Pittsburgh, PA, USA, 2011, pp. 179–186.

[122] J. Cunha, J. Fernandes, J. Mendes, H. Pacheco, J. Saraiva, Bidirectional Transformation of Model-Driven Spreadsheets, in: Proceedings of the 5th International Conference on Theory and Practice of Model Transformations (ICMT 2012), Springer Lecture Notes in Computer Science, Prague, Czech Republic, 2012, pp. 105–120.

[123] J. Cunha, J. Fernandes, J. Mendes, J. Saraiva, Extension and Implementation of ClassSheet Models, in: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2012), Innsbruck, Austria, 2012, pp. 19–22.

[124] J. Cunha, J. a. P. Fernandes, J. a. Saraiva, From Relational ClassSheets to UML+OCL, in: Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC 2012), Trento, Italy, 2012, pp. 1151–1158.

[125] F. Hermans, M. Pinzger, A. van Deursen, Automatically Extracting Class Diagrams from Spreadsheets, in: Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP 2010), Maribor, Slovenia, 2010, pp. 52–75.

[126] J. Cunha, J. a. Saraiva, J. Visser, From Spreadsheets to Relational Databases and Back, in: Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2009), Savannah, GA, USA, 2009, pp. 179–188.

[127] J. Cunha, J. Saraiva, J. Visser, Discovery-Based Edit Assistance for Spreadsheets, in: Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC 2009), Corvallis, OR, USA, 2009, pp. 233–237.

[128] J. Cunha, J. a. Saraiva, J. Visser, Model-Based Programming Environments for Spreadsheets, in: Proceedings of the 16th Brazilian Conference on Programming Languages (SBLP 2012), Natal, Brazil, 2012, pp. 117–133.

[129] L. Beckwith, J. Cunha, J. Fernandes, J. Saraiva, End-Users Productivity in Model-Based Spreadsheets: An Empirical Study, in: Proceedings of the 3rd International Symposium on End-User Development (IS-EUD 2011), Springer Lecture Notes in Computer Science, Torre Canne, Italy, 2011, pp. 282–288.

[130] N. Wilde, C. Lewis, Spreadsheet-based interactive graphics: from prototype to tool, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 1990), Seattle, WA, USA, 1990, pp. 153–160.

[131] C. Lewis, NoPumpG: Creating Interactive Graphics with Spreadsheet Machinery, Visual Programming Environments: Paradigms and Systems (1990) 526–546.

[132] C. Hughes, J. Moshell, Action Graphics: A Spreadsheet-based Language for Animated Simulation, Visual Languages and Applications (1990) 203–235.

[133] N. P. Wilde, A WYSIWYC (What You See Is What You Compute) Spreadsheet, in: Proceedings of the IEEE Symposium on Visual Languages (VL 1993), Bergen, Norway, 1993, pp. 72–76.

[134] M. M. Burnett, A. Agrawal, P. van Zee, Exception Handling in the Spreadsheet Paradigm, IEEE Transactions on Software Engineering 26 (10) (2000) 923–942.

[135] R. W. Sebesta, Concepts of Programming Languages (4th ed.), Addison-Wesley-Longman, 1999.

[136] B. Bekenn, R. Hooper, Reducing Spreadsheet Risk with FormulaDataSleuth, in: Proceedings of the European Spreadsheet Risks Interest Group 9th Annual Conference (EuSpRIG 2008), London, United Kingdom, 2008.

[137] C. Chambers, M. Erwig, M. Luckey, SheetDiff: A Tool for Identifying Changes in Spreadsheets, in: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2010), Madrid, Spain, 2010, pp. 85–92.

[138] A. Harutyunyan, G. Borradaile, C. Chambers, C. Scaffidi, Planted-model evaluation of algorithms for identifying differences between spreadsheets, in: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2012), Innsbruck, Austria, 2012, pp. 7–14.

[139] P. O'Beirne, Spreadsheet Refactoring, in: Proceedings of the European Spreadsheet Risks Interest Group 11th Annual Conference (EuSpRIG 2010), London, United Kingdom, 2010.

[140] S. Badame, D. Dig, Refactoring meets Spreadsheet Formulas, in: Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM 2012), Riva del Garda, Trento, Italy, 2012, pp. 399–409.

[141] W. R. Harris, S. Gulwani, Spreadsheet Table Transformations from Examples, in: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI 2011), San Jose, CA, USA, 2011, pp. 317–328.

[142] Y. Ye, G. Fischer, Reuse-Conducive Development Environments, Automated Software Engineering 12 (2) (2005) 199–235.

[143] R. W. Djang, M. M. Burnett, Similarity Inheritance: A New Model of Inheritance for Spreadsheet VPLs, in: Proceedings of the IEEE Symposium on Visual Languages (VL 1998), Halifax, NS, Canada, 1998, pp. 134–141.

[144] M. Montigel, Portability and Reuse of Components for Spreadsheet Languages, in: Proceedings of the IEEE CS International Symposium on Human-Centric Computing Languages and Environments (HCC 2002), Arlington, VA, USA, 2002, pp. 77–79.

[145] D. Jannach, M. Zanker, M. Ge, M. Gröning, Recommender systems in computer science and information systems - a landscape of research, in: Proceedings of the 13th International Conference on E-Commerce and Web Technologies (EC-WEB 2012), Vienna, 2012, pp. 76–87.

[146] L. P. S. Elazar J. Pedhazur, Measurement Design and Analysis: An Integrated Approach, Lawrence Erlbaum Assoc Inc, 1991.

[147] Using a structured design approach to reduce risks in end user spreadsheet development, Information and Management 37 (1) (2000) 1–12.

[148] F. Karlsson, Using two heads in practice, in: Proceedings of the 4th International Workshop on End-user Software Engineering (WEUSE 2008), Leipzig, Germany, 2008, pp. 43–47.

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

[150] R. R. Panko, R. H. S. Jr., Hitting the wall: errors in developing and code inspecting a ‘simple’ spreadsheet model, Decision Support Systems 22 (4) (1998) 337–353.

[151] J. P. A. Ioannidis, Why most published research findings are false, PLoS Medizine 2 (8).

[152] R. Nuzzo, Scientific method: Statistical errors, Nature 506 (2014) 150–152.

[153] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, S. Wiedenbeck, The State of the Art in End-User Software Engineering, ACM Computing Surveys 43 (3) (2011) 21:1–21:44.

[154] S. R. Thorne, D. Ball, Z. Lawson, A Novel Approach to Formulae Production and Overconfidence Measurement to Reduce Risk in Spreadsheet Modelling, in: Proceedings of the European Spreadsheet Risks Interest Group 5th Annual Conference (EuSpRIG 2004), Klagenfurt, Austria, 2004.

[155] L. Beckwith, S. Sorte, M. Burnett, S. Wiedenbeck, T. Chintakovid, C. Cook, Designing features for both genders in end-user programming environments, in: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2005), Dallas, TX, 2005, pp. 153–160.

[156] J. R. Ruthruff, A. Phalgune, L. Beckwith, M. M. Burnett, C. R. Cook, Rewarding "good" behavior: End-user debugging and rewards, in: Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC 2004), 2004, pp. 115–122.

[157] S. P. Jones, A. Blackwell, M. Burnett, A user-centred approach to functions in excel, in: Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden, 2003, pp. 165–176.

[158] P. Sestoft, J. Z. Sørensen, Sheet-defined functions: Implementation and initial evaluation, in: End-User Development, Vol. 7897 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 88–103.

Model-based diagnosis of spreadsheet programs: A constraint-based debugging approach

[Placeholder]

Dietmar Jannach
TU Dortmund, Germany
dietmar.jannach@tu-dortmund.de

Thomas Schmitz
TU Dortmund, Germany
thomas.schmitz@tu-dortmund.de

Automated Software Engineering, 23.1, March 2016

DOI: https://doi.org/10.1007/s10515-014-0141-7

MergeXplain: Fast Computation of Multiple Conflicts for Diagnosis

[Placeholder]

Kostyantyn Shchekotykhin
Alpen-Adria University Klagenfurt, Austria
kostyantyn.shchekotykhin@aau.at

Dietmar Jannach
TU Dortmund, Germany
dietmar.jannach@tu-dortmund.de

Thomas Schmitz
TU Dortmund, Germany
thomas.schmitz@tu-dortmund.de

IJCAI '15, July 25–31, 2015, Buenos Aires, Argentina

ISBN: 978-1-57735-738-4

Journal of Artificial Intelligence Research 55 (2016) 835-887 Submitted 10/2015; published 04/2016

Parallel Model-Based Diagnosis on Multi-Core Computers

Dietmar Jannach DIETMAR.JANNACH@TU-DORTMUND.DE
Thomas Schmitz THOMAS.SCHMITZ@TU-DORTMUND.DE
TU Dortmund, Germany

Kostyantyn Shchekotykhin KOSTYANTYN.SHCHEKOTYKHIN@AAU.AT
Alpen-Adria University Klagenfurt, Austria

Abstract

Model-Based Diagnosis (MBD) is a principled and domain-independent way of analyzing why a system under examination is not behaving as expected. Given an abstract description (model) of the system's components and their behavior when functioning normally, MBD techniques rely on observations about the actual system behavior to reason about possible causes when there are discrepancies between the expected and observed behavior. Due to its generality, MBD has been successfully applied in a variety of application domains over the last decades.

In many application domains of MBD, testing different hypotheses about the reasons for a failure can be computationally costly, e.g., because complex simulations of the system behavior have to be performed. In this work, we therefore propose different schemes of parallelizing the diagnostic reasoning process in order to better exploit the capabilities of modern multi-core computers. We propose and systematically evaluate parallelization schemes for Reiter's hitting set algorithm for finding all or a few leading minimal diagnoses using two different conflict detection techniques. Furthermore, we perform initial experiments for a basic depth-first search strategy to assess the potential of parallelization when searching for one single diagnosis. Finally, we test the effects of parallelizing "direct encodings" of the diagnosis problem in a constraint solver.

1. Introduction

Model-Based Diagnosis (MBD) is a subfield of Artificial Intelligence that is concerned with the automated determination of possible causes when a *system* is not behaving as expected. In the early days of MBD, the diagnosed "systems" were typically hardware artifacts like electronic circuits. In contrast to earlier heuristic diagnosis approaches which connected symptoms with possible causes, e.g., through expert rules (Buchanan & Shortliffe, 1984), MBD techniques rely on an abstract and explicit representation (model) of the examined system. Such models contain both information about the system's structure, i.e., the list of components and how they are connected, as well as information about the behavior of the components when functioning correctly. When such a model is available, the expected behavior (outputs) of a system given some inputs can thus be calculated. A *diagnosis problem* arises whenever the expected behavior conflicts with the observed system behavior. MBD techniques at their core construct and test hypotheses about the faultiness of individual components of the system. Finally, a *diagnosis* is considered as a subset of the components that, if assumed to be faulty, can explain the observed behavior of the system.

Reiter (1987) suggests a formal logical characterization of the diagnosis problem "from first principles" and proposed a breadth-first tree construction algorithm to determine all diagnoses for a given problem. Due to the generality of the used knowledge-representation language and the suggested algorithms for the computation of diagnoses, MBD has been later on applied to a variety of application problems other than hardware. The application fields of MBD, for example, include the diagnosis of knowledge bases and ontologies, process specifications, feature models, user interface specifications and user preference statements, and various types of software artifacts including functional and logic programs as well as VHDL, Java or spreadsheet programs (Felfernig, Friedrich, Jannach, & Stumptner, 2004; Mateis, Stumptner, Wieland, & Wotawa, 2000; Jannach & Schmitz, 2014; Wotawa, 2001b; Felfernig, Friedrich, Isak, Shchekotykhin, Teppan, & Jannach, 2009; Console, Friedrich, & Dupré, 1993; Friedrich & Shchekotykhin, 2005; Stumptner & Wotawa, 1999; Friedrich, Stumptner, & Wotawa, 1999; White, Benavides, Schmidt, Trinidad, Dougherty, & Cortés, 2010; Friedrich, Fugini, Mussi, Pernici, & Tagni, 2010).

In several of these application fields, the search for diagnoses requires repeated computations based on modified versions of the original model to test the different hypotheses about the faultiness of individual components. In several works the original problem is converted into a Constraint Satisfaction Problem (CSP) and a number of relaxed versions of the original CSP have to be solved to construct a new node in the search tree (Felfernig et al., 2004; Jannach & Schmitz, 2014; White et al., 2010). Depending on the application domain, the computation of CSP solutions or the check for consistency can, however, be computationally intensive and actually represents the most costly operation during the construction of the search tree. Similar problems arise when other underlying reasoning techniques, e.g., for ontology debugging (Friedrich & Shchekotykhin, 2005), are used.

Current MBD algorithms are sequential in nature and generate one node at a time. Therefore, they do not exploit the capabilities of today's multi-core computer processors, which can nowadays be found even in mobile devices. In this paper, we propose new schemes to parallelize the diagnostic reasoning process to better exploit the available computing resources of modern computer hardware. In particular, this work comprises the following algorithmic contributions and insights based on experimental evaluations:

- We propose two parallel versions of Reiter's (1987) sound and complete *Hitting Set* (HS) algorithm to speed up the process of finding all diagnoses, which is a common problem setting in the above-described MBD applications. Both approaches can be considered as "window-based" parallelization schemes, which means that only a limited number of search nodes is processed in parallel at each point in time.

- We evaluate two different conflict detection techniques in a multi-core setting, where the goal is to find a few "leading" diagnoses. In this set of experiments, multiple conflicts can be computed at the construction of each tree node using the novel MergeXplain method (MXP) (Shchekotykhin, Jannach, & Schmitz, 2015) and more processing time is therefore implicitly allocated for conflict generation.

- We demonstrate that speedups can also be achieved through parallelization for scenarios in which we search for one single diagnosis, e.g., when using a basic parallel depth-first strategy.

- We measure the improvements that can be achieved through parallel constraint solving when using a "direct" CSP-based encoding of the diagnosis problem. This experiment illustrates that parallelization in the underlying solvers, in particular when using a direct encoding, can be advantageous.

We evaluate the proposed parallelization schemes through an extensive set of experiments. The following problem settings are analyzed.

(i) Standard benchmark problems from the diagnosis research community;

(ii) Mutated CSPs from a Constraint Programming competition and from the domain of CSP-based spreadsheet debugging (Jannach & Schmitz, 2014);

(iii) Faulty OWL ontologies as used for the evaluation of MBD-based debugging techniques of very expressive ontologies (Shchekotykhin, Friedrich, Fleiss, & Rodler, 2012);

(iv) Synthetically generated problems which allow us to vary the characteristics of the underlying diagnosis problem.

The results show that using parallelization techniques can help to achieve substantial speedups for the diagnosis process (a) across a variety of application scenarios, (b) without exploiting any specific knowledge about the structure of the underlying diagnosis problem, (c) across different problem encodings, and (d) also for application problems like ontology debugging which cannot be efficiently encoded as SAT problems.

The outline of the paper is as follows. In the next section, we define the main concepts of MBD and introduce the algorithm used to compute diagnoses. In Section 3, we present and systematically evaluate the parallelization schemes for Reiter's HS-tree method when the goal is to find all minimal diagnoses. In Section 4, we report the results of the evaluations when we implicitly allocate more processing time for conflict generation using MXP for conflict detection. In Section 5 we assess the potential gains for a comparably simple randomized depth-first strategy and a hybrid technique for the problem of finding one single diagnosis. The results of the experiments for the direct CSP encoding are reported in Section 6. In Section 7 we discuss previous works. The paper ends with a summary and an outlook in Section 8.

2. Reiter's Diagnosis Framework

This section summarizes Reiter's (1987) diagnosis framework which we use as a basis for our work.

2.1 Definitions

Reiter (1987) formally characterized Model-Based Diagnosis using first-order logic. The main definitions can be summarized as follows.

Definition 2.1. *(Diagnosable System) A diagnosable* system *is described as a pair* (SD, Comps) *where* SD *is a system description (a set of logical sentences) and* Comps *represents the system's components (a finite set of constants).*

The connections between the components and the normal behavior of the components are described in terms of logical sentences. The normal behavior of the system components is usually described in SD with the help of a distinguished negated unary predicate $\neg\textsc{ab}(.)$, meaning "not abnormal".

A diagnosis problem arises when some observation $o \in \textsc{Obs}$ of the system's input-output behavior (again expressed as first-order sentences) deviates from the expected system behavior. A *diagnosis* then corresponds to a subset of the system's components which we assume to behave abnormally (be faulty) and where these assumptions must be consistent with the observations. In other words, the malfunctioning of these components *can* be a possible reason for the observations.

Definition 2.2. *(Diagnosis) Given a diagnosis problem* (SD, Comps, Obs)*, a diagnosis is a subset minimal set* $\Delta \subseteq \textsc{Comps}$ *such that* $\text{SD} \cup \textsc{Obs} \cup \{\textsc{ab}(c) | c \in \Delta\} \cup \{\neg\textsc{ab}(c) | c \in \textsc{Comps} \backslash \Delta\}$ *is consistent.*

According to Definition 2.2, we are only interested in *minimal* diagnoses, i.e., diagnoses which contain no superfluous elements and are thus not supersets of other diagnoses. Whenever we use the term *diagnosis* in the remainder of the paper, we therefore mean *minimal diagnosis*. Whenever we refer to non-minimal diagnoses, we will explicitly mention this fact.

Finding all diagnoses can in theory be done by simply trying out all possible subsets of Comps and checking their consistency with the observations. Reiter (1987), however, proposes a more efficient procedure based on the concept of conflicts.

Definition 2.3. *(Conflict) A conflict for* (SD, Comps, Obs) *is a set* $\{c_1, ..., c_k\} \subseteq \textsc{Comps}$ *such that* $\text{SD} \cup \textsc{Obs} \cup \{\neg\textsc{ab}(c_1), ..., \neg\textsc{ab}(c_k)\}$ *is inconsistent.*

A conflict corresponds to a subset of components which, if assumed to behave normally, are not consistent with the observations. A conflict c is considered to be *minimal*, if no proper subset of c exists which is also a conflict.

2.2 Hitting Set Algorithm

Reiter (1987) then discusses the relationship between conflicts and diagnoses and claims in his Theorem 4.4 that the set of diagnoses for a collection of (minimal) conflicts F is equivalent to the set $\mathcal{H}$ of *minimal hitting sets*[1] of F.

To determine the minimal hitting sets and therefore the diagnoses, Reiter proposes a breadth-first search procedure and the construction of a hitting set tree (HS-tree), whose construction is guided by conflicts. In the logic-based definition of the MBD problem (Reiter, 1987), the conflicts are computed by calls to a *Theorem Prover (TP)*. The TP component itself is considered as a "black box" and no assumptions are made about how the conflicts are determined. Depending on the application scenario and problem encoding, one can, however, also use specific algorithms like QuickXplain (Junker, 2004), Progression (Marques-Silva, Janota, & Belov, 2013) or MergeXplain (Shchekotykhin et al., 2015), which guarantee that the computed conflict sets are minimal.

The main principle of the HS-tree algorithm is to create a search tree where each node is either labeled with a conflict or represents a diagnosis. In the latter case the node is not further expanded. Otherwise, a child node is generated for each element of the node's

1. Given a collection C of subsets of a finite set S, a hitting set for C is a subset of S which contains at least one element from each subset in C. This corresponds to the set cover problem.

conflict and each outgoing edge is labeled with one component of the node's conflict. In the subsequent expansions of each node the components that were used to label the edges on the path from the root of the tree to the current node are assumed to be faulty. Each newly generated child node is again either a diagnosis or will be labeled with a conflict that does not contain any component that is already assumed to be faulty at this stage. If no conflict can be found for a node, the path labels represent a diagnosis in the sense of Definition 2.2.

2.2.1 Example

In the following example we will show how the HS-tree algorithm and the QuickXplain (QXP) conflict detection technique can be combined to locate a fault in a specification of a CSP. A CSP instance I is defined as a tuple (V, D, C), where $V = \{v_1, \ldots, v_n\}$ is a set of variables, $D = \{D_1, \ldots, D_n\}$ is a set of domains for each of the variables in V, and $C = \{C_1, \ldots, C_k\}$ is a set of constraints. An assignment to any subset $X \subseteq V$ is a set of pairs $A = \{\langle v_1, d_1\rangle, \ldots, \langle v_k, d_m\rangle\}$ where $v_i \in X$ is a variable and $d_j \in D_i$ is a value from the domain of this variable. An assignment comprises exactly one variable-value pair for each variable in X. Each *constraint* $C_i \in C$ is defined over a list of variables S, called scope, and forbids or allows certain simultaneous assignments to the variables in its scope. An assignment A to S *satisfies* a constraint C_i if A comprises an assignment allowed by C_i. An assignment A is a *solution* to I if it satisfies all constraints C.

Consider a CSP instance I with variables $V = \{a, b, c\}$ where each variable has the domain $\{1, 2, 3\}$ and the following set of constraints are defined:

$$C1: \ a > b, \quad C2: \ b > c, \quad C3: \ c = a, \quad C4: \ b < c$$

Obviously, no solution for I exists and our diagnosis problem consists in finding subsets of the constraints whose definition is faulty. The engineer who has modeled the CSP could, for example, have made a mistake when writing down $C2$, which should have been $b < c$. Eventually, $C4$ was added later on to correct the problem, but the engineer forgot to remove $C2$. Given the faulty definition of I, two minimal conflicts exist, namely $\{\{C1, C2, C3\}, \{C2, C4\}\}$, which can be determined with the help of QXP. Given these two conflicts, the HS-tree algorithm will finally determine three minimal hitting sets $\{\{C2\}, \{C1, C4\}, \{C3, C4\}\}$, which are diagnoses for the problem instance. The set of diagnoses also contains the true cause of the error, the definition of $C2$.

Let us now review in more detail how the HS-tree/QXP combination works for the example problem. We illustrate the tree construction in Figure 1. In the logic-based definition of Reiter, the HS-tree algorithm starts with a check if the observations Obs are consistent with the system description SD and the components Comps. In our application setting this corresponds to a check if there exists any solution for the CSP instance.[2] Since this is not the case, a QXP-call is made, which returns the conflict $\{C1, C2, C3\}$, which is used as a label for the root node (①) of the tree. For each element of the conflict, a child node is created and the conflict element is used as a path label. At each tree node, again the consistency of SD, Obs, and Comps is tested; this time, however, all the elements that appear

2. Comps are the constraints $\{C1 \ldots C4\}$ and SD corresponds to the semantics/logic of the constraints when working correctly, e.g., $AB(C1) \vee (a > b)$. Obs is empty in this example but could be a partial value assignment (test case) in another scenario.

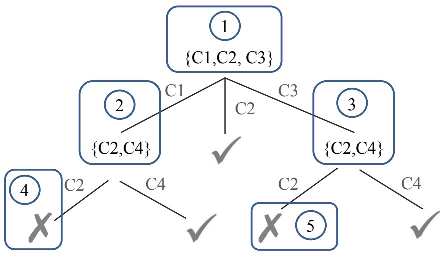


Figure 1: Example for HS-tree construction.

as labels on the path from the root node to the current node are considered to be abnormal. In the CSP diagnosis setting, this means that we check if there is any solution to a modified version of our original CSP from which we remove the constraints that appear as labels on the path from the root to the current node.

At node ②, $C1$ is correspondingly considered to be abnormal. As removing $C1$ from the CSP is, however, not sufficient and no solution exists for the relaxed problem, another call to QXP is made, which returns the conflict $\{C2, C4\}$. $\{C1\}$ is therefore not a diagnosis and the new conflict is used as a label for node ②. The algorithm then proceeds in breadth-first style and tests if assuming $\{C2\}$ or $\{C3\}$ to be individually faulty is "consistent with the observations", which in our case means that a solution to the relaxed CSP exists. Since $\{C2\}$ is a diagnosis – at least one solution exists if $C2$ is removed from the CSP definition – the node is marked with ✓ and not further expanded. At node ③, which does not correspond to a diagnosis, the already known conflict $\{C2, C4\}$ can be reused as it has no overlap with the node's path label and no call to TP (QXP) is required. At the last tree level, the nodes ④ and ⑤ are not further expanded ("closed" and marked with ✗) because $\{C2\}$ has already been identified as a diagnosis at the previous level and the resulting diagnoses would be supersets of $\{C2\}$. Finally, the sets $\{C1, C4\}$ and $\{C3, C4\}$ are identified as additional diagnoses.

2.2.2 Discussion

Soundness and Completeness According to Reiter (1987), the breadth-first construction scheme and the node closing rule ensure that only minimal diagnoses are computed. At the end of the HS-tree construction process, each set of edge labels on the path from the root of the tree to a node marked with ✓ corresponds to a diagnosis.[3]

Greiner, Smith, and Wilkerson (1989), later on, identified a potential problem in Reiter's algorithm for cases in which the conflicts returned by TP are not guaranteed to be minimal. An extension of the algorithm based on an HS-DAG (directed acyclic graph) structure was proposed to solve the problem.

In the context of our work, we only use methods that return conflicts which are guaranteed to be minimal. For example, according to Theorem 1 in the work of Junker (2004), given a set of formulas and a sound and complete consistency checker, QXP always returns

3. Reiter (1987) states in Theorem 4.8 that given a set of conflict sets F, the HS-tree algorithm outputs a pruned tree T such that the set $\{H(n)|n$ is a node of T labeled with ✓$\}$ corresponds to the set $\mathcal{H}$ of all minimal hitting sets of F where $H(n)$ is a set of arc labels on the path from the node n to the root.

either a *minimal* conflict or 'no conflict'. This minimality guarantee in turn means that the combination of the HS-tree algorithm and QXP is *sound* and *complete*, i.e., all returned solutions are actually (minimal) diagnoses and no diagnosis for the given set of conflicts will be missed. The same holds when computing multiple conflicts at a time with MXP (Shchekotykhin et al., 2015).

To simplify the presentation of our parallelization approaches, we will therefore rely on Reiter's original HS-tree formulation; an extension to deal with the HS-DAG structure (Greiner et al., 1989) is possible.

On-Demand Conflict Generation and Complexity In many of the above-mentioned applications of MBD to practical problems, the conflicts have to be computed "on-demand", i.e., during tree construction, because we cannot generally assume that the set of minimal conflicts is given in advance. Depending on the problem setting, finding these conflicts can therefore be the computationally most intensive part of the diagnosis process.

Generally, finding hitting sets for a collection of sets is known to be an NP-hard problem (Garey & Johnson, 1979). Moreover, deciding if an additional diagnosis exists when conflicts are computed on demand is NP-complete even for propositional Horn theories (Eiter & Gottlob, 1995). Therefore, a number of heuristics-based, approximate and thus incomplete, as well as problem-specific diagnosis algorithms have been proposed over the years. We will discuss such approaches in later sections. In the next section, we, however, focus on (worst-case) application scenarios where the goal is to find *all minimal diagnoses* for a given problem, i.e., we focus on complete algorithms.

Consider, for example, the problem of debugging program specifications (e.g., constraint programs, knowledge bases, ontologies, or spreadsheets) with MBD techniques as mentioned above. In these application domains, it is typically not sufficient to find *one* minimal diagnosis. In the work of Jannach and Schmitz (2014), for example, the spreadsheet developer is presented with a ranked list of all sets of formulas (diagnoses) that represent possible reasons why a certain test case has failed. The developer can then either inspect each of them individually or provide additional information (e.g., test cases) to narrow down the set of candidates. If only one diagnosis was computed and presented, the developer would have no guarantee that it is the true cause of the problem, which can lead to limited acceptance of the diagnosis tool.

3. Parallel HS-Tree Construction

In this section we present two sound and complete parallelization strategies for Reiter's HS-tree method to determine all minimal diagnoses.

3.1 A Non-recursive HS-Tree Algorithm

We use a non-recursive version of Reiter's sequential HS-tree algorithm as a basis for the implementation of the two parallelization strategies. Algorithm 1 shows the main loop of a breadth-first procedure, which uses a list of open nodes to be expanded as a central data structure.

The algorithm takes a diagnosis problem (DP) instance as input and returns the set Δ of diagnoses. The DP is given as a tuple (SD, Comps, Obs), where SD is the system

Algorithm 1: diagnose: Main algorithm loop.

```
Input: A diagnosis problem (SD, Comps, Obs)
Result: The set $\Delta$ of diagnoses

$\Delta = \emptyset$; paths = $\emptyset$; conflicts = $\emptyset$;
nodesToExpand = ⟨generateRootNode(SD, Comps, Obs)⟩;
while nodesToExpand $\neq \langle\ \rangle$ do
    newNodes = ⟨ ⟩;
    node = head(nodesToExpand) ;
    foreach $c \in$ node.conflict do
        generateNode(node, c, $\Delta$, paths, conflicts, newNodes);
    nodesToExpand = tail(nodesToExpand) $\oplus$ newNodes;
return $\Delta$;
```

Algorithm 2: generateNode: Node generation logic.

```
Input: An existingNode to expand, a conflict element $c \in$ Comps,
       the sets $\Delta$, paths, conflicts, newNodes

newPathLabel = existingNode.pathLabel $\cup$ {c};
if ($\nexists\ l \in \Delta : l \subseteq$ newPathLabel) $\wedge$ checkAndAddPath(paths, newPathLabel) then
    node = new Node(newPathLabel);
    if $\exists\ S \in$ conflicts : $S \cap$ newPathLabel $= \emptyset$ then
        node.conflict = S;
    else
        newConflicts = checkConsistency(SD, Comps, Obs, node.pathLabel);
        node.conflict = head(newConflicts);
    if node.conflict $\neq \emptyset$ then
        newNodes = newNodes $\oplus$ ⟨node⟩;
        conflicts = conflicts $\cup$ newConflicts;
    else
        $\Delta = \Delta \cup$ {node.pathLabel};
```

description, Comps the set of components that can potentially be faulty and Obs a set of observations. The method generateRootNode creates the initial node, which is labeled with a conflict and an empty path label. Within the *while* loop, the first element of a "first-in-first-out" (FIFO) list of open nodes nodesToExpand is taken as the current element. The function generateNode (Algorithm 2) is called for each element of the node's conflict and adds new leaf nodes, which still have to be explored, to a global list. These new nodes are then appended ($\oplus$) to the remaining list of open nodes in the main loop, which continues until no more elements remain for expansion.[4] Algorithm 2 (generateNode) implements the node generation logic, which includes Reiter's proposals for conflict re-use, tree pruning, and the management of the lists of known conflicts, paths and diagnoses. The method determines the path label for the new node and checks if the new path label is not a superset of an already found diagnosis.

Algorithm 3: checkAndAddPath: Adding a new path label with a redundancy check.

```
Input: The previously explored paths, the newPathLabel to be explored
Result: Boolean stating if newPathLabel was added to paths

if $\nexists\, l \in paths : l = newPathLabel$ then
    paths = paths ∪ newPathLabel;
    return true;
return false;
```

The function checkAndAddPath (Algorithm 3) is then used to check if the node was not already explored elsewhere in the tree. The function returns *true* if the new path label was successfully inserted into the list of known paths. Otherwise, the list of known paths remains unchanged and the node is "closed".

For new nodes, either an existing conflict is reused or a new one is created with a call to the consistency checker (Theorem Prover), which tests if the new node is a diagnosis or returns a set of minimal conflicts otherwise. Depending on the outcome, a new node is added to the list *nodesToExpand* or a diagnosis is stored. Note that Algorithm 2 has no return value but instead modifies the sets Δ, *paths*, *conflicts*, and *newNodes*, which were passed as parameters.

3.2 Level-Wise Parallelization

Our first parallelization scheme examines all nodes *of one tree level* in parallel and proceeds with the next level once all elements of the level have been processed. In the example shown in Figure 1, this would mean that the computations (consistency checks and theorem prover calls) required for the three first-level nodes labeled with $\{C1\}$, $\{C2\}$, and $\{C3\}$ can be done in three parallel threads. The nodes of the next level are explored when all threads of the previous level are finished.

Using this Level-Wise Parallelization (LWP) scheme, the breadth-first character is maintained. The parallelization of the computations is generally feasible because the consistency checks for each node can be done independently from those done for the other nodes on the same level. Synchronization is only required to make sure that no thread starts exploring a path which is already under examination by another thread.

Algorithm 4 shows how the sequential Algorithm 1 can be adapted to support this parallelization approach. Again, we maintain a list of open nodes to be expanded. The difference is that we run the expansion of all these nodes in parallel and collect all the

4. A limitation regarding the search depth or the number of diagnoses to find can be easily integrated into this scheme.

Algorithm 4: diagnoseLW: Level-Wise Parallelization.

```
Input: A diagnosis problem (SD, Comps, Obs)
Result: The set $\Delta$ of diagnoses
$\Delta = \emptyset$; $conflicts = \emptyset$; paths = $\emptyset$;
nodesToExpand = $\langle$generateRootNode(SD, Comps, Obs)$\rangle$;
while nodesToExpand $\neq \langle\ \rangle$ do
    newNodes = $\langle\ \rangle$;
    foreach node $\in$ nodesToExpand do
        foreach c $\in$ node.conflict do          // Do computations in parallel
            threads.execute(generateNode(node, c, $\Delta$, paths, conflicts, newNodes));
    threads.await();                      // Wait for current level to complete
    nodesToExpand = newNodes;                                // Prepare next level
return $\Delta$;
```

nodes of the next level in the variable *newNodes*. Once the current level is finished, we overwrite the list *nodesToExpand* with the list containing the nodes of the next level.

The Java-like API calls used in the pseudo-code in Algorithm 4 have to be interpreted as follows. The statement *threads.execute()* takes a function as a parameter and schedules it for execution in a pool of threads of a given size. With a thread pool of, e.g., size 2, the generation of the first two nodes would be done in parallel and the next ones would be queued until one of the threads has finished. With this mechanism, we can ensure that the number of threads executed in parallel is less than or equal to the number of hardware threads or CPUs.

The statement *threads.await()* is used for synchronization and blocks the execution of the subsequent code until all scheduled threads are finished. To guarantee that the same path is not explored twice, we make sure that no two threads in parallel add a node with the same path label to the list of known paths. This can be achieved by declaring the function checkAndAddPath as a "critical section" (Dijkstra, 1968), which means that no two threads can execute the function in parallel. Furthermore, we have to make the access to the global data structures (e.g., the already known conflicts or diagnoses) thread-safe, i.e., ensure that no two threads can simultanuously manipulate them.[5]

3.3 Full Parallelization

In LWP, there can be situations where the computation of a conflict for a specific node takes particularly long. This, however, means that even if all other nodes of the current level are finished and many threads are idle, the expansion of the HS-tree cannot proceed before the level is completed. Algorithm 5 shows our proposed Full Parallelization (FP) algorithm variant, which immediately schedules every expandable node for execution and thereby avoids such potential CPU idle times at the end of each level.

5. Controlling such concurrency aspects is comparably simple in modern programming languages like Java, e.g., by using the `synchronized` keyword.

```
Algorithm 5: diagnoseFP: Full Parallelization.
Input: A diagnosis problem (SD, Comps, Obs)
Result: The set $\Delta$ of diagnoses
$\Delta = \emptyset$; paths = $\emptyset$; conflicts = $\emptyset$;
nodesToExpand = $\langle$generateRootNode(SD, Comps, Obs)$\rangle$;
size = 1; lastSize = 0;
while ($size \neq lastSize$) $\vee$ ($threads.activeThreads \neq 0$) do
    for $i = 1$ to $size - lastSize$ do
        node = nodesToExpand.get[lastSize + i];
        foreach $c \in node.conflict$ do
            threads.execute(generateNodeFP(node, c, $\Delta$, paths, conflicts,
              nodesToExpand));
    lastSize = size;
    wait();
    size = nodesToExpand.length();
return $\Delta$;
```

The main loop of the algorithm is slightly different and basically monitors the list of nodes to expand. Whenever new entries in the list are observed, i.e., when the last observed list size is different from the current one, it retrieves the recently added elements and adds them to the thread queue for execution. The algorithm returns the diagnoses when no new elements are added since the last check and no more threads are active.[6]

With FP, the search does not necessarily follow the breadth-first strategy anymore and non-minimal diagnoses are found during the process. Therefore, whenever we find a new diagnosis d, we have to check if the set of known diagnoses Δ contains supersets of d and remove them from Δ.

The updated generateNode method is listed in Algorithm 6. When updating the shared data structures (*nodesToExpand*, *conflicts*, and Δ), we again make sure that the threads do not interfere with each other. The mutual exclusive section is marked with the `synchronized` keyword.

When compared to LWP, FP does not have to wait at the end of each level if a specific node takes particularly long to generate. On the other hand, FP needs more synchronization between threads, so that in cases where the last nodes of a level are finished at the same time, LWP could also be advantageous. We will evaluate this aspect in Section 3.5.

3.4 Properties of the Algorithms

Algorithm 1 together with Algorithms 2 and 3 corresponds to an implementation of the HS-tree algorithm (Reiter, 1987). Algorithm 1 implements the breadth-first search strategy – point (1) in Reiter's HS-tree algorithm – since the nodes stored in the list *nodesToExpand*

6. The functions `wait()` and `notify()` implement the semantics of pausing a thread and awaking a paused thread in the Java programming language and are used to avoid active waiting loops.

Algorithm 6: generateNodeFP: Extended node generation logic.

Input: An *existingNode* to expand, $c \in$ Comps,
sets Δ, *paths*, *conflicts*, *nodesToExpand*

```
newPathLabel = existingNode.pathLabel $\cup$ {c};
if ($\nexists\, l \in \Delta : l \subseteq newPathLabel$) $\wedge$ checkAndAddPath(paths, newPathLabel) then
    node = new Node(newPathLabel);
    if $\exists\, S \in conflicts : S \cap newPathLabel = \emptyset$ then
        node.conflict = S;
    else
        newConflicts = checkConsistency(SD, Comps, Obs, node.pathLabel);
        node.conflict = head(newConflicts);
    synchronized
        if $node.conflict \neq \emptyset$ then
            nodesToExpand = nodesToExpand $\oplus$ $\langle$node$\rangle$;
            conflicts = conflicts $\cup$ newConflicts;
        else if $\nexists\, d \in \Delta : d \subseteq newPathLabel$ then
            $\Delta = \Delta \cup$ {node.pathLabel};
            for $d \in \Delta : d \supseteq newPathLabel$ do
                $\Delta = \Delta \setminus d$;
notify();
```

are processed iteratively in a first-in-first-out order (see lines 5 and 8). Algorithm 2 first checks if the pruning rules (i) and (ii) of Reiter can be applied in line 2. These rules state that a node can be pruned if (i) there exists a diagnosis or (ii) there is a set of labels corresponding to some path in the tree such that it is a subset of the set of labels on the path to the node. Pruning rule (ii) is implemented through Algorithm 3. Pruning rule (iii) of Reiter's algorithm is not necessary since in our settings a *TP*-call guarantees to return minimal conflicts.

Finally, point (2) of Reiter's HS-tree algorithm description is implemented in the lines 4-8 of Algorithm 2. Here, the algorithm checks if there is a conflict that can be reused as a node label. In case no reuse is possible, the algorithm calls the theorem prover *TP* to find another minimal conflict. If a conflict is found, the node is added to the list of open nodes *nodesToExpand*. Otherwise, the set of node path labels is added to the set of diagnoses. This corresponds to the situation in Reiter's algorithm where we would mark a node in the HS-tree with the ✓ symbol. Note that we do not label any nodes with ✗ as done in Reiter's algorithms since we simply do not store such nodes in the expansion list.

Overall, we can conclude that our HS-tree algorithm implementation (Algorithm 1 to 3) has the same properties as Reiter's original HS-tree algorithm. Namely, each hitting set returned by the algorithm is minimal (soundness) and all existing minimal hitting sets are found (completeness).

3.4.1 Level-Wise Parallelization (LWP)

Theorem 3.1. *Level-Wise Parallelization is sound and complete.*

Proof. The proof is based on the fact that LWP uses the same expansion and pruning techniques as the sequential algorithm (Algorithms 2 and 3). The main loop in line 3 applies the same procedure as the original algorithm with the only difference that the executions of Algorithm 2 are done in parallel for each level of the tree. Therefore, the only difference between the sequential algorithm and LWP lies in the order in which the nodes of one level are labeled and generated.

Let us assume that there are two nodes n_1 and n_2 in the tree and that the sequential HS-tree algorithm will process n_1 before n_2. Assuming that neither n_1 nor n_2 correspond to diagnoses, the sequential Algorithm 1 would correspondingly first add the child nodes of n_1 to the queue of open nodes and later on append the child nodes of n_2.

If we parallelize the computations needed for the generation of n_1 and n_2 in LWP, it can happen that the computations for n_1 need longer than those for n_2. In this case the child nodes of n_2 will be placed in the queue first. The order of how these nodes are subsequently processed is, however, irrelevant for the computation of the minimal hitting sets, since neither the labeling nor the pruning rules are influenced by it. In fact, the labeling of any node n only depends on whether or not a minimal conflict set f exists such that $H(n) \cap f = \emptyset$, but not on the other nodes on the same level. The pruning rules state that any node n can be pruned if there exists a node n' labeled with ✓ such that $H(n') \subseteq H(n)$, i.e., supersets of already found diagnoses can be pruned. If n and n' are on the same level, then $|H(n)| = |H(n')|$. Consequently, the pruning rule is applied only if $H(n) = H(n')$. Therefore, the order of nodes, i.e., which of the nodes is pruned, is irrelevant and no minimal hitting set is lost. Consequently, LWP is complete.

Soundness of the algorithm follows from the fact that LWP constructs the hitting sets always in the order of increasing cardinality. Therefore, LWP will always return only minimal hitting sets even in scenarios in which we should stop after k diagnoses are found, where $1 \geqslant k < N$ is a predefined constant and N is the total number of diagnoses of a problem. □

3.4.2 Full Parallelization (FP)

The minimality of the hitting sets encountered during the search is not guaranteed by FP, since the algorithm schedules a node for processing immediately after its generation (line 8 of Algorithm 5). The special treatment in the generateNodeFP function ensures that no supersets of already found hitting sets are added to Δ and that supersets of a newly found hitting set will be removed in a thread-safe manner (lines 13 – 16 of Algorithm 6). Due to this change in generateNodeFP, the analysis of soundness and completeness has to be done for two distinct cases.

Theorem 3.2. *Full Parallelization is sound and complete, if applied to find all diagnoses up to some cardinality.*

Proof. FP stops if either (i) no further hitting set exists, i.e., all leaf nodes of a tree are labeled either with ✓ or with ✗, or (ii) the predefined cardinality (tree-depth) is reached. In this latter case, every leaf node of the tree is labeled either with ✓, ✗, or a minimal conflict set. Case (ii) can be reduced to (i) by removing all branches from the tree that are labeled with a minimal conflict. These branches are irrelevant since they can only contribute to minimal hitting sets of higher cardinality. Therefore, without loss of generality, we can limit our discussion to case (i).

According to the definition of generateNodeFP, the tree is built using the same pruning rule as done in the sequential HS-tree algorithm. As a consequence, the tree generated by FP must comprise *at least* all nodes of the tree that is generated by the sequential HS-tree procedure. Therefore, according to Theorem 4.8 in the work of Reiter (1987) the tree T generated by FP must comprise a set of leaf nodes labeled with ✓ such that the set $\{H(n)|n$ is a node of T labeled by ✓$\}$ corresponds to the set $\mathcal{H}$ of all minimal hitting sets. Moreover, the result returned by FP comprises only minimal hitting sets, because generateNodeFP removes all hitting sets from $\mathcal{H}$ which are supersets of other hitting sets. Consequently, FP is sound and complete, when applied to find all diagnoses. □

Theorem 3.3. *Full Parallelization cannot guarantee completeness and soundness when applied to find the first k diagnoses, i.e. $1 \geqslant k < N$, where N is the total number of diagnoses of a problem.*

Proof. The proof can be done by constructing an example for which FP returns at least one non-minimal hitting set in the set Δ, thus violating Definition 2.2. For instance, this situation might occur if FP is applied to find one single diagnosis for the example problem presented in Section 2.2.1. Let us assume that the generation of the node corresponding to the path $C2$ is delayed, e.g., because the operating system scheduled another thread for execution first, and node 4 is correspondingly generated first. In this case, the algorithm would return the non-minimal hitting set $\{C1, C2\}$ which is not a diagnosis. □

Note that the elements of the set Δ returned by FP in this case can be turned to diagnoses by applying a minimization algorithm like Inv-QuickXplain (Shchekotykhin, Friedrich, Rodler, & Fleiss, 2014), an algorithm that adopts the principles of QuickXplain and applies a divide-and-conquer strategy to find one minimal diagnosis for a given set of inconsistent constraints.

Given a hitting set H and a diagnosis problem, the algorithm is capable of computing a *minimal* hitting set $H' \subseteq H$ requiring only $O(|H'| + |H'| \log(|H|/|H'|))$ calls to the theorem prover TP. The first part, $|H'|$, reflects the computational costs of determining whether or not H' is minimal. The second part represents the number of subproblems that must be considered by the divide-and-conquer algorithm in order to find the minimal hitting set H'.

3.5 Evaluation

To determine which performance improvements can be achieved through the various forms of parallelization proposed in this paper, we conducted a series of experiments with diagnosis problems from a number of different application domains. Specifically, we used electronic circuit benchmarks from the DX Competition 2011 Synthetic Track, faulty descriptions of Constraint Satisfaction Problems (CSPs), as well as problems from the domain of ontology debugging. In addition, we ran experiments with synthetically created diagnosis problems to analyze the impact of varying different problem characteristics. All diagnosis algorithms evaluated in this paper were implemented in Java unless noted otherwise. Generally, we use wall clock times as our performance measure.

In the main part of the paper, we will focus on the results for the DX Competition problems as this is the most widely used benchmark. The results for the other problem setups will be presented and discussed in the appendix of the paper. In most cases, the results for the DX Competition problems follow a similar trend as those that are achieved with the other experiments.

In this section we will compare the HS-tree parallelization schemes LWP and FP with the sequential version of the algorithm, when the goal is to *find all diagnoses*.

3.5.1 Dataset and Procedure

For this set of experiments, we selected the first five systems of the DX Competition 2011 Synthetic Track (see Table 1) (Kurtoglu & Feldman, 2011). For each system, the competition specifies 20 scenarios with injected faults resulting in different faulty output values. We used the system description and the given input and output values for the diagnosis process. The additional information about the injected faults was of course ignored. The problems were converted into Constraint Satisfaction Problems. In the experiments we used Choco (Prud'homme, Fages, & Lorca, 2015) as a constraint solver and QXP for conflict detection, which returns one minimal conflict when called during node construction.

As the computation times required for conflict identification strongly depend on the order of the possibly faulty constraints, we shuffled the constraints for each test and repeated all tests 100 times. We report the wall clock times for the actual diagnosis task; the times required for input and output are independent from the HS-tree construction scheme and not relevant for our benchmarks. For the parallel approaches, we used a thread pool of size four.[7]

Table 1 shows the characteristics of the systems in terms of the number of constraints (#C) and the problem variables (#V).[8] The numbers of the injected faults (#F) and the numbers of the calculated diagnoses (#D) vary strongly because of the different scenarios for each system. For both columns we show the ranges of values over all scenarios. The columns $\overline{\#\text{D}}$ and $\overline{|\text{D}|}$ indicate the average number of diagnoses and their average cardinality. As can be seen, the search tree for the diagnosis can become extremely broad with up to 6,944 diagnoses with an average diagnosis size of only 3.38 for the system c432.

3.5.2 Results

Table 2 shows the averaged results when searching for all minimal diagnoses. We first list the running times in milliseconds for the sequential version (Seq.) and then the improvements of LWP or FP in terms of *speedup* and *efficiency* with respect to the sequential version. Speedup S_p is computed as $S_p = T_1/T_p$, where T_1 is the wall time when using 1 thread (the sequential algorithm) and T_p the wall time when p parallel threads are used. A speedup of

7. Having four hardware threads is a reasonable assumption on standard desktop computers and also mobile devices. The hardware we used for the evaluation in this chapter – a laptop with an Intel i7-3632QM CPU, 16GB RAM, running Windows 8 – also had four cores with hyperthreading. The results of an evaluation on server hardware with 12 cores are reported later in this Section.
8. For systems marked with *, the search depth was limited to their actual number of faults to ensure that the sequential algorithm terminates within a reasonable time frame.

System	#C	#V	#F	#D	$\overline{\#D}$	$\overline{\lvert D \rvert}$
74182	21	28	4 - 5	30 - 300	139.0	4.66
74L85	35	44	1 - 3	1 - 215	66.4	3.13
74283*	38	45	2 - 4	180 - 4,991	1,232.7	4.42
74181*	67	79	3 - 6	10 - 3,828	877.8	4.53
c432*	162	196	2 - 5	1 - 6,944	1,069.3	3.38

Table 1: Characteristics of the selected DXC benchmarks.

2 would therefore mean that the needed computation times were halved; a speedup of 4, which is the theoretical optimum when using 4 threads, means that the time was reduced to one quarter. The efficiency E_p is defined as S_p/p and compares the speedup with the theoretical optimum. The fastest algorithm for each system is highlighted in bold.

System	**Seq.(QXP)**	**LWP(QXP)**		**FP(QXP)**	
	[ms]	$\mathbf{S_4}$	$\mathbf{E_4}$	$\mathbf{S_4}$	$\mathbf{E_4}$
74182	65	2.23	0.56	**2.28**	**0.57**
74L85	209	2.55	0.64	**2.77**	**0.69**
74283*	371	2.53	0.63	**2.66**	**0.67**
74181*	21,695	1.22	0.31	**3.19**	**0.80**
c432*	85,024	1.47	0.37	**3.75**	**0.94**

Table 2: Observed performance gains for the DXC benchmarks when searching for all diagnoses.

In all tests, both parallelization approaches outperform the sequential algorithm. Furthermore, the differences between the sequential algorithm and one of the parallel approaches were statistically significant ($p < 0.05$) in 95 of the 100 tested scenarios. For all systems, FP was more efficient than LWP and the speedups range from 2.28 to 3.75 (i.e., up to a reduction of running times of more than 70%). In 59 of the 100 scenarios the differences between LWP and FP were statistically significant. A trend that can be observed is that the efficiency of FP is higher for the more complex problems. The reason is that for these problems the time needed for the node generation is much larger in absolute numbers than the additional overhead times that are required for thread synchronization.

3.5.3 Adding More Threads

In some use cases the diagnosis process can be done on powerful server architectures that often have even more CPU cores than modern desktop computers. In order to assess to which extent more than 4 threads can help to speed up the diagnosis process, we tested the different benchmarks on a server machine with 12 CPU cores. For this test we compared FP with 4, 8, 10, and 12 threads to the sequential algorithm.

The results of the DXC benchmark problems are shown in Table 3. For all tested systems the diagnosis process was faster using 8 instead of 4 threads and substantial speedups up to 5.20 could be achieved compared to the sequential diagnosis, which corresponds to a runtime reduction of 81%. For all but one system, the utilization of 10 threads led to additional speedups. Using 12 threads was the fastest for 3 of the 5 tested systems. The efficiency, however, degrades as more threads are used, because more time is needed for the synchronization between threads. Using more threads than the hardware actually has cores did not result in additional speedups for any of the tested systems. The reason is that for most of the time all threads are busy with conflict detection, e.g., finding solutions to CSPs, and use almost 100% of the processing power assigned to them.

System	**Seq.(QXP)**	**FP(QXP)**							
	[ms]	**S_4**	**E_4**	**S_8**	**E_8**	**S_{10}**	**E_{10}**	**S_{12}**	**E_{12}**
74182	58	2.09	0.52	2.43	0.30	2.52	0.25	**2.54**	**0.21**
74L85	184	2.53	0.63	3.29	0.41	3.35	0.34	**3.38**	**0.28**
74283	51,314	3.04	0.76	4.38	0.55	4.42	0.44	**4.50**	**0.37**
74181*	13,847	3.45	0.86	**5.20**	**0.65**	5.11	0.51	5.19	0.43
c432*	43,916	3.43	0.86	4.77	0.60	**5.00**	**0.50**	4.74	0.39

Table 3: Observed performance gains for the DXC benchmarks on a server with 12 hardware threads.

3.5.4 Additional Experiments

The details of additional experiments that were conducted to compare the proposed parallelization schemes with the sequential HS-Tree algorithm are presented in Section A.1 in the appendix. The results show that significant speedups can also be achieved for other Constraint Satisfaction Problems (Section A.1.1) and ontologies (Section A.1.2). The appendix furthermore contains an analysis of effects when adding more threads to the benchmarks of the CSPs and ontologies (Section A.1.3) and presents the results of a simulation experiment in which we systematically varied different problem characteristics (Section A.1.4).

3.5.5 Discussion

Overall, the results of the evaluations show that both parallelization approaches help to improve the performance of the diagnosis process, as for all tested scenarios both approaches achieved speedups. In most cases FP is faster than LWP. However, depending on the specifics of the given problem setting, using LWP can be advantageous in some situations, e.g., when the time needed to generate each node is very small or when the conflict generation time does not vary strongly. In these cases the synchronization overhead needed for FP is higher than the cost of waiting for all threads to finish. For the tested ontologies in Section A.1.2, this was the case in four of the tested scenarios.

Although FP is on average faster than LWP and significantly better than the sequential HS-tree construction approach, for some of the tested scenarios its efficiency is still far from the optimum of 1. This can be explained by different effects. For example, the effect of *false sharing* can happen if the memory of two threads is allocated to the same block (Bolosky & Scott, 1993). Then every access to this memory block is synchronized although the two threads do not really share the same memory. Another possible effect is called *cache* *contention* (Chandra, Guo, Kim, & Solihin, 2005). If threads work on different computing cores but share the same memory, cache misses can occur more often depending on the problem characteristics and thus the theoretical optimum cannot be reached in these cases.

4. Parallel HS-Tree Construction with Multiple Conflicts Per Node

Both in the sequential and the parallel version of the HS-tree algorithm, the Theorem Prover *TP* call corresponds to an invocation of QXP. Whenever a new node of the HS-tree is created, QXP searches for exactly one new conflict in case none of the already known conflicts can be reused. This strategy has the advantage that the call to TP immediately returns after one conflict has been determined. This in turn means that the other parallel execution threads immediately "see" this new conflict in the shared data structures and can, in the best case, reuse it when constructing new nodes.

A disadvantage of computing only one conflict at a time with QXP is that the search for conflicts is *restarted* on each invocation. We recently proposed a new conflict detection technique called MergeXplain (MXP) (Shchekotykhin et al., 2015), which is capable of computing multiple conflicts in one call. The general idea of MXP is to continue the search after the identification of the first conflict and look for additional conflicts in the remaining constraints (or logical sentences) in a divide-and-conquer approach.

When combined with a sequential HS-tree algorithm, the effect is that during tree construction more time is initially spent for conflict detection before the construction continues with the next node. In exchange, the chances of having a conflict available for reuse increase for the next nodes. At the same time, the identification of some of the conflicts is less time-intensive as smaller sets of constraints have to be investigated due to the divide-and-conquer approach of MXP. An experimental evaluation on various benchmark problems shows that substantial performance improvements are possible in a sequential HS-tree scenario when the goal is to find *a few leading diagnoses* (Shchekotykhin et al., 2015).

In this section, we explore the benefits of using MXP with the parallel HS-tree construction schemes proposed in the previous section. When using MXP in combination with multiple threads, the implicit effect is that more CPU processing power is devoted to conflict generation as the individual threads need more time to complete the construction of a new node. In contrast to the sequential version, the other threads can continue with their work in parallel.

In the next section, we will briefly review the MXP algorithm before we report the results of the empirical evaluation on our benchmark datasets (Section 4.2).

4.1 Background – QuickXplain and MergeXplain

Algorithm 7 shows the QXP conflict detection technique of Junker (2004) applied to the problem of finding a conflict for a diagnosis problem during HS-tree construction.

QXP operates on two sets of constraints[9] which are modified through recursive calls. The "background theory" $\mathcal{B}$ comprises the constraints that will *not* be considered anymore to be part of a conflict at the current stage. At the beginning, this set contains SD, Obs,

9. We use the term constraints here as in the original formulation. As QXP is independent from the underlying reasoning technique, the elements of the sets could be general logical sentences as well.

Algorithm 7: QuickXplain (QXP)

```
Input: A diagnosis problem (SD, Comps, Obs), a set visitedNodes of elements
Output: A set containing one minimal conflict $CS \subseteq \mathcal{C}$
1  $\mathcal{B}$ = SD ∪ Obs ∪ {ab(c)|c ∈ visitedNodes}; $\mathcal{C} = \{\neg$ab$(c) | c \in$ Comps\visitedNodes};
2  if isConsistent($\mathcal{B} \cup \mathcal{C}$) then return 'no conflict';
3  else if $\mathcal{C} = \emptyset$ then return $\emptyset$;
4  return {c|¬ab(c) ∈ getConflict($\mathcal{B}, \mathcal{B}, \mathcal{C}$)};

   function getConflict ($\mathcal{B}$, $D$, $\mathcal{C}$)
5      if $D \neq \emptyset \wedge \neg$ isConsistent($\mathcal{B}$) then return $\emptyset$;
6      if $|\mathcal{C}| = 1$ then return $\mathcal{C}$;
7      Split $\mathcal{C}$ into disjoint, non-empty sets $\mathcal{C}_1$ and $\mathcal{C}_2$
8      $D_2 \leftarrow$ getConflict ($\mathcal{B} \cup \mathcal{C}_1$, $\mathcal{C}_1$, $\mathcal{C}_2$)
9      $D_1 \leftarrow$ getConflict ($\mathcal{B} \cup D_2$, $D_2$, $\mathcal{C}_1$)
10     return $D_1 \cup D_2$;
```

and the set of nodes on the path to the current node of the HS-tree (*visited nodes*). The set $\mathcal{C}$ represents the set of constraints in which we search for a conflict.

If there is no conflict or $\mathcal{C}$ is empty, the algorithm immediately returns. Otherwise getConflict is called, which corresponds to Junker's QXP method with the minor difference that getConflict does not require a strict partial order for the set of constraints $\mathcal{C}$. We introduce this variant of QXP since we cannot always assume that prior fault information is available that would allow us to generate this order.

The rough idea of QXP is to relax the input set of faulty constraints $\mathcal{C}$ by partitioning it into two sets $\mathcal{C}_1$ and $\mathcal{C}_2$. If $\mathcal{C}_1$ is a conflict, the algorithm continues partitioning $\mathcal{C}_1$ in the next recursive call. Otherwise, i.e., if the last partitioning has split all conflicts of $\mathcal{C}$ so that there are no conflicts left in $\mathcal{C}_1$, the algorithm extracts a conflict from the sets $\mathcal{C}_1$ and $\mathcal{C}_2$. This way, QXP finally identifies individual constraints which are inconsistent with the remaining consistent set of constraints and the background theory.

MXP builds on the ideas of QXP but computes multiple conflicts in one call (if they exist). The general procedure is shown in Algorithm 8. After the initial consistency checks, the method findConflicts is called, which returns a tuple $\langle \mathcal{C}', \Gamma \rangle$, where $\mathcal{C}'$ is a set of remaining consistent constraints and Γ is a set of found conflicts. The function recursively splits the set $\mathcal{C}$ of constraints in two halves. These parts are individually checked for consistency, which allows us to exclude larger consistent subsets of $\mathcal{C}$ from the search process. Besides the potentially identified conflicts, the calls to findConflicts also return two sets of constraints which are consistent ($\mathcal{C}'_1$ and $\mathcal{C}'_2$). If the union of these two sets is not consistent, we look for a conflict within $\mathcal{C}'_1 \cup \mathcal{C}'_1$ (and the background theory) in the style of QXP.

More details can be found in our earlier work, where also the results of an in-depth experimental analysis are reported (Shchekotykhin et al., 2015).

```
Algorithm 8: MergeXplain (MXP)
   Input: A diagnosis problem (SD, Comps, Obs), a set visitedNodes of elements
   Output: $\Gamma$, a set of minimal conflicts
 1 $\mathcal{B} = \text{SD} \cup \text{Obs} \cup \{\text{ab}(c) | c \in visitedNodes\}$; $\mathcal{C} = \{\neg\text{ab}(c) | c \in \text{Comps} \setminus \Delta\}$;
 2 if $\neg isConsistent(\mathcal{B})$ then return 'no solution';
 3 if $isConsistent(\mathcal{B} \cup \mathcal{C})$ then return $\emptyset$;
 4 $\langle \_, \Gamma \rangle \leftarrow$ findConflicts$(\mathcal{B}, \mathcal{C})$
 5 return $\{c | \neg\text{ab}(c) \in \Gamma\}$;

   function findConflicts $(\mathcal{B}, \mathcal{C})$ returns tuple $\langle \mathcal{C}', \Gamma \rangle$
 6     if $isConsistent(\mathcal{B} \cup \mathcal{C})$ then return $\langle \mathcal{C}, \emptyset \rangle$;
 7     if $|\mathcal{C}| = 1$ then return $\langle \emptyset, \{\mathcal{C}\} \rangle$;
 8     Split $\mathcal{C}$ into disjoint, non-empty sets $\mathcal{C}_1$ and $\mathcal{C}_2$
 9     $\langle \mathcal{C}'_1, \Gamma_1 \rangle \leftarrow$ findConflicts$(\mathcal{B}, \mathcal{C}_1)$
10     $\langle \mathcal{C}'_2, \Gamma_2 \rangle \leftarrow$ findConflicts$(\mathcal{B}, \mathcal{C}_2)$
11     $\Gamma \leftarrow \Gamma_1 \cup \Gamma_2$;
12     while $\neg isConsistent(\mathcal{C}'_1 \cup \mathcal{C}'_2 \cup \mathcal{B})$ do
13         $X \leftarrow$ getConflict$(\mathcal{B} \cup \mathcal{C}'_2, \mathcal{C}'_2, \mathcal{C}'_1)$
14         $CS \leftarrow X \cup$ getConflict$(\mathcal{B} \cup X, X, \mathcal{C}'_2)$
15         $\mathcal{C}'_1 \leftarrow \mathcal{C}'_1 \setminus \{\alpha\}$ where $\alpha \in X$
16         $\Gamma \leftarrow \Gamma \cup \{CS\}$
17     return $\langle \mathcal{C}'_1 \cup \mathcal{C}'_2, \Gamma \rangle$;
```

4.2 Evaluation

In this section we evaluate the effects of parallelizing the diagnosis process when we use MXP instead of QXP to calculate the conflicts. As in (Shchekotykhin et al., 2015) we focus on finding a limited set of (five) minimal diagnoses.

4.2.1 Implementation Variants

Using MXP during parallel tree construction implicitly means that more time is allocated for conflict generation than when using QXP before proceeding to the next node. To analyze to which extent the use of MXP is beneficial we tested three different strategies of using MXP within the full parallelization method FP.

Strategy (1): In this configuration we simply called MXP instead of QXP during node generation. Whenever MXP finds a conflict, it is added to the global list of known conflicts and can be (re-)used by other parallel threads. The thread that executes MXP during node generation continues with the next node when MXP returns.

Strategy (2): This strategy implements a variant of MXP which is slightly more complex. Once MXP finds the first conflict, the method immediately returns this conflict such that the calling thread can continue exploring additional nodes. At the same time, a new background thread is started which continues the search for additional conflicts, i.e., it completes the work of the MXP call. In addition, whenever MXP finds a new conflict it checks if any other already running node generation thread could have reused the conflict if it had been available beforehand. If this is the case, the search for conflicts of this *other* thread is stopped as no new conflict is needed anymore. Strategy (2) could in theory result in better CPU utilization, as we do not have to wait for a MXP call to finish before we can continue building the HS-tree. However, the strategy also leads to higher synchronization costs between the threads, e.g., to notify working threads about newly identified conflicts.

Strategy (3): Finally, we parallelized the conflict detection procedure itself. Whenever the set $\mathcal{C}$ of constraints is split into two parts, the first recursive call of findConflicts is queued for execution in a thread pool and the second call is executed in the current thread. When both calls are finished, the algorithm continues.

We experimentally evaluated all three configurations on our benchmark datasets. Our results showed that Strategy (2) did not lead to measurable performance improvements when compared to Strategy (1). The additional communication costs seem to be higher than what can be saved by executing the conflict detection process in the background in its own thread. Strategy (3) can be applied in combination with the other strategies, but similar to the experiments reported for the sequential HS-tree construction (Shchekotykhin et al., 2015), no additional performance gains could be observed due to the higher synchronization costs. The limited effectiveness of Strategies (2) and (3) can in principle be caused by the nature of our benchmark problems and these strategies might be more advantageous in different problem settings. In the following, we will therefore only report the results of applying Strategy (1).

4.2.2 Results for the DXC Benchmark Problems

The results for the DXC benchmarks are shown in Table 4. The left side of the table shows the results when using QXP and the right hand side shows the results for MXP. The speedups shown in the FP columns refer to the respective sequential algorithms using the same conflict detection technique.

Using MXP instead of QXP is favorable when using a sequential HS-tree algorithm as also reported in the work about MXP (Shchekotykhin et al., 2015). The reduction of running times ranges from 17% to 44%. The speedups obtained through FP when using MXP are comparable to FP using QXP and range from 1.33 to 2.10, i.e., they lead to a reduction of the running times of up to 52%. These speedups were achieved *in addition* to the speedups that the sequential algorithm using MXP could already achieve over QXP.

The best results are printed in bold face in Table 4 and using MXP in combination with FP consistently performs best. Overall, using FP in combination with MXP was 38% to 76% faster than the sequential algorithm using QXP. These tests indicate that our parallelization method works well also for conflict detection techniques that are more complex than QXP and, as in this case, return more than one conflict for each call. In addition, investing more time for conflict detection in situations where the goal is to find a few leading diagnoses proves to be a promising strategy.

4.2.3 Additional Experiments and Discussion

Again we ran additional experiments on constraint problems and ontology debugging problems. The detailed results are provided in Section A.2.

System	Seq.(QXP)	FP(QXP)		Seq.(MXP)	FP(MXP)	
	[ms]	S_4	E_4	[ms]	S_4	E_4
74182	12	1.26	0.32	10	**1.52**	**0.38**
74L85	15	1.36	0.34	12	**1.33**	**0.33**
74283	49	1.58	0.39	35	**1.48**	**0.37**
74181	699	1.99	0.55	394	**2.10**	**0.53**
c432	3,714	1.77	0.44	2,888	**1.72**	**0.43**

Table 4: Observed performance gains for the DXC benchmarks (QXP vs MXP).

Overall, the results obtained when embedding MXP in the sequential algorithm confirm the results by Shchekotykhin et al. (2015) that using MXP is favorable over QXP for all but a few very small problem instances. However, we can also observe that allocating more time for conflict detection with MXP in a parallel processing setup can help to further speedup the diagnosis process when we search for a number of leading diagnoses. The best-performing configuration across all experiments is using the Full Parallelization method in combination with MXP as this setup led to the shortest computation times in 20 out of the 25 tested scenarios (DX benchmarks, CSPs, ontologies).

5. Parallelized Depth-First and Hybrid Search

In some application domains of MBD, finding all minimal diagnoses is either not required or simply not possible because of the computational complexity or application-specific constraints on the allowed response times. For such settings, a number of algorithms have been proposed over the years, which for example try to find one or a few minimal diagnoses very quickly or find all diagnoses of a certain cardinality (Metodi, Stern, Kalech, & Codish, 2014; Feldman, Provan, & van Gemund, 2010b; de Kleer, 2011). In some cases, the algorithms can in principle be extended or used to find all diagnoses. They are, however, not optimized for this task.

Instead of analyzing the various heuristic, stochastic or approximative algorithms proposed in the literature individually with respect to their potential for parallelization, we will analyze in the next section if parallelization can be helpful already for the simple class of depth-first algorithms. In that context, we will also investigate if measurable improvements can be achieved without using any (domain-specific) heuristic. Finally, we will propose a hybrid strategy which combines depth-first and full-parallel HS-tree construction and will conduct additional experiments to assess if this strategy can be advantageous for the task of quickly finding one minimal diagnosis.

5.1 Parallel Random Depth-First Search

The section introduces a parallelized depth-first search algorithm to quickly find one single diagnosis. As the different threads explore the tree in a partially randomized form, we call the scheme Parallel Random Depth-First Search (PRDFS).

5.1.1 Algorithm Description

Algorithm 9 shows the main program of a recursive implementation of PRDFS. Similar to the HS-tree algorithm, the search for diagnoses is guided by conflicts. This time, however, the algorithm greedily searches in a depth-first manner. Once a diagnosis is found, it has to be checked for minimality because the diagnosis can contain redundant elements. The "minimization" of a non-minimal diagnosis can be achieved by calling a method like Inv-QuickXplain (Shchekotykhin et al., 2014) or by simply trying to remove one element of the diagnosis after the other and checking if the resulting set is still a diagnosis.

Algorithm 9: diagnosePRDFS: Parallelized random depth-first search.

```
Input: A diagnosis problem (SD, Comps, Obs),
       the number minDiags of diagnoses to find
Result: The set $\Delta$ of diagnoses

1 $\Delta = \emptyset$; conflicts = $\emptyset$;
2 rootNode = getRootNode(SD, Comps, Obs);
3 for $i = 1$ to nbThreads do
4     threads.execute(expandPRDFS(rootNode, minDiags, $\Delta$, conflicts));
5 while $|\Delta| < minDiags$ do
6     wait();
7 threads.shutdownNow();
8 return $\Delta$;
```

The idea of the parallelization approach in the algorithm is to start multiple threads from the root node. All of these threads perform the depth-first search in parallel, but pick the next conflict element to explore in a randomized manner.

The logic for expanding a node is shown in Algorithm 10. First, the conflict of the given node is copied, so that changes to this set of constraints will not affect the other threads. Then, as long as not enough diagnoses were found, a randomly chosen constraint from the current node's conflict is used to generate a new node. The expansion function is then immediately called recursively for the new node, thereby implementing the depth-first strategy. Any identified diagnosis is minimized before being added to the list of known diagnoses. Similar to the previous parallelization schemes, the access to the global lists of known conflicts has to be made thread-safe. When the specified number of diagnoses is found or all threads are finished, the statement *threads.shutdownNow()* immediately stops the execution of all threads that are still running and the results are returned. The semantics of *threads.execute()*, *wait()*, and *notify()* are the same as in Section 3.

5.1.2 Example

Let us apply the depth-first method to the example from Section 2.2.1. Remember that the two conflicts for this problem were $\{\{C1, C2, C3\}, \{C2, C4\}\}$. A partially expanded tree for this problem can be seen in Figure 2.

Algorithm 10: expandPRDFS: Parallel random depth-first node expansion.

Input: An *existingNode* to expand, the number *minDiags* of diagnoses to find, the sets Δ and *conflicts*

```
C = existingNode.conflict.clone();            // Copy existingNode's conflict
while $|\Delta| < minDiags \wedge |C| > 0$ do
    Randomly pick a constraint c from C
    $C = C \backslash \{c\}$;
    newPathLabel = existingNode.pathLabel ∪ {c};
    node = new Node(newPathlabel);
    if $\exists\ S \in conflicts : S \cap newPathLabel = \emptyset$ then
        node.conflict = S;
    else
        node.conflict = checkConsistency(SD, Comps, Obs, node.pathLabel);
    if $node.conflict \neq \emptyset$ then                // New conflict found
        conflicts = conflicts ∪ node.conflict;
        // Recursive call implements the depth-first search strategy
        expandPRDFS(node, minDiags, Δ, conflicts);
    else                                          // Diagnosis found
        diagnosis = minimize(node.pathLabel);
        $\Delta = \Delta \cup \{diagnosis\}$;
        if $|\Delta| \geqslant minDiags$ then
            notify();
```

In the example, first the root node ① is created and again the conflict $\{C1, C2, C3\}$ is found. Next, the random expansion would, for example, pick the conflict element $C1$ and generate node ②. For this node, the conflict $\{C2, C4\}$ will be computed because $\{C1\}$ alone is not a diagnosis. Since the algorithm continues in a depth-first manner, it will then pick one of the label elements of node ②, e.g., $C2$ and generate node ③. For this node, the consistency check succeeds, no further conflict is computed and the algorithm has found a diagnosis. The found diagnosis $\{C1, C2\}$ is, however, not minimal as it contains the redundant element $C1$. The function Minimize, which is called at the end of Algorithm 10, will therefore remove the redundant element to obtain the correct diagnosis $\{C2\}$.

If we had used more than one thread in this example, one of the parallel threads would have probably started expanding the root node using the conflict element $C2$ (node ④). In that case, the single element diagnosis $\{C2\}$ would have been identified already at the first level. Adding more parallel threads can therefore help to increase the chances to find one hitting set faster as different parts of the HS-tree are explored in parallel.

Instead of the random selection strategy, more elaborate schemes to pick the next nodes are possible, e.g., based on application-specific heuristics or fault probabilities. One could also better synchronize the search efforts of the different threads to avoid duplicate calculations. We conducted experiments with an algorithm variant that used a shared and

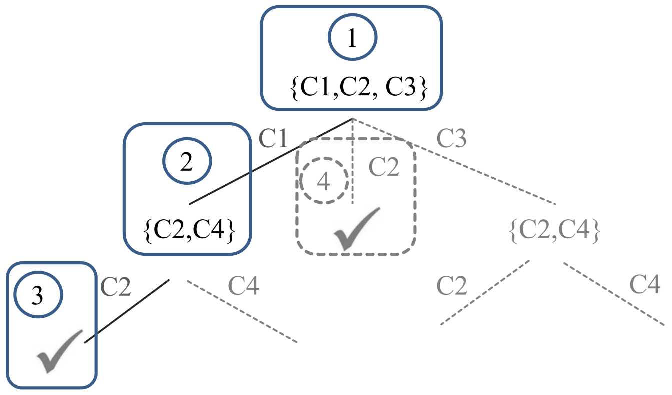


Figure 2: Example for HS-tree construction with PRDFS.

synchronized list of open nodes to avoid that two threads generate an identical sub-tree in parallel. We did, however, not observe significantly better results than with the method shown in Algorithm 9 probably due to the synchronization overhead.

5.1.3 Discussion of Soundness and Completeness

Every single thread in the depth-first algorithm systematically explores the full search space based on the conflicts returned by the Theorem Prover. Therefore, all existing diagnoses will be found when the parameter *minDiags* is equal or higher than the number of actually existing diagnoses.

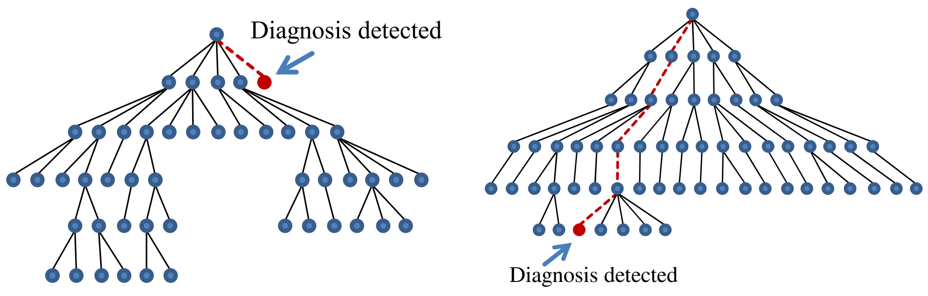
Whenever a (potentially non-minimal) diagnosis is encountered, the minimization process ensures that only minimal diagnoses are stored in the list of diagnoses. The duplicate addition of the same diagnosis by one or more threads in the last lines of the algorithm is prevented because we consider diagnoses to be equal if they contain the same set of elements and Δ as a set by definition cannot contain the same element twice.

Overall, the algorithm is designed to find one or a few diagnoses quickly. The computation of *all* minimal diagnoses is possible with the algorithm but highly inefficient, e.g., due to the computational costs of minimizing the diagnoses.

5.2 A Hybrid Strategy

Let us again consider the problem of finding *one* minimal diagnosis. One can easily imagine that the choice of the best parallelization strategy, i.e., breadth-first or depth-first, can depend on the specifics of the given problem setting and the actual size of the existing diagnoses. If a single-element diagnosis exists, exploring the first level of the HS-tree in a breadth-first approach might be the best choice (see Figure 3(a)). A depth-first strategy might eventually include this element in a non-minimal diagnosis, but would then have to do a number of additional calculations to ensure the minimality of the diagnosis.

If, in contrast, the smallest actually existing diagnosis has a cardinality of, e.g., five, the breadth-first scheme would have to fully explore the first four HS-tree levels before finding the five-element diagnosis. The depth-first scheme, in contrast, might quickly find a superset of the five-element diagnosis, e.g., with six elements, and then only needs six additional consistency checks to remove the redundant element from the diagnosis (Figure 3(b)).



(a) Breadth-first strategy is advantageous. (b) Depth-first strategy is advantageous.

Figure 3: Two problem configurations for which different search strategies are favorable.

Since we cannot know the cardinality of the diagnoses in advance, we propose a hybrid strategy, in which half of the threads adopt a depth-first strategy and the other half uses the fully parallelized breadth-first regime. To implement this strategy, the Algorithms 5 (FP) and 9 (PRDFS) can be started in parallel and each algorithm is allowed to use one half or some other defined share of the available threads. The coordination between the two algorithms can be done with the help of shared data structures that contain the known conflicts and diagnoses. When enough diagnoses (e.g. one) are found, all running threads can be terminated and the results are returned.

5.3 Evaluation

We evaluated the different strategies for efficiently *finding one minimal diagnosis* on the same set of benchmark problems that were used in the previous sections. The experiment setup was identical except that the goal was to find one arbitrary diagnosis and that we included the additional depth-first algorithms. In order to measure the potential benefits of parallelizing the depth-first search, we ran the benchmarks for PRDFS both with 4 threads and with 1 thread, where the latter setup corresponds to a Random Depth First Search (RDFS) without parallelization.

5.3.1 Results for the DXC Benchmark Problems

The results for the DXC benchmark problems are shown in Table 5. Overall, for all tested systems, each of the approaches proposed in this paper can help to speed up the process of finding one single diagnosis. In 88 of the 100 evaluated scenarios at least one of the tested approaches was statistically significantly faster than the sequential algorithm. For the other 12 scenarios, finding one single diagnosis was too simple so that only modest but no significant speedups compared to the sequential algorithm were obtained.

When comparing the individual parallel algorithms, the following observations can be made:

- For most of the examples, the PRDFS method is faster than the breadth-first search implemented in the FP technique. For one benchmark system, the PRDFS approach can even achieve a speedup of 11 compared to the sequential algorithm, which corresponds to a runtime reduction of 91%.
- When compared with the non-parallel RDFS, PRDFS could achieve higher speedups for all tested systems except the most simple one, which only took 16 ms even for the sequential algorithm. Overall, parallelization can therefore be advantageous also for depth-first strategies.
- The performance of the Hybrid strategy lies in between the performances of its components PRDFS and FP for 4 of the 5 tested systems. For these systems, it is closer to the faster one of the two. Adopting the hybrid strategy can therefore represent a good choice when the structure of the problem is not known in advance, as it combines both ideas of breadth-first and depth-first search and is able to quickly find a diagnosis for problem settings with unknown characteristics.

System	**Seq.**	**FP**		**RDFS**	**PRDFS**		**Hybrid**	
	[ms]	**S$_4$**	**E$_4$**	**[ms]**	**S$_4$**	**E$_4$**	**S$_4$**	**E$_4$**
74182	16	1.37	0.34	**9**	0.84	0.21	0.84	0.21
74L85	13	**1.34**	**0.33**	11	1.06	0.27	1.05	0.26
74283	54	1.67	0.42	25	**1.22**	**0.31**	1.06	0.26
74181	691	2.08	0.52	74	**1.23**	**0.31**	1.04	0.26
c432	2,789	1.89	0.47	1,435	**2.96**	**0.74**	1.81	0.45

Table 5: Observed performance gains for DXC benchmarks for finding one diagnosis.

5.3.2 Additional Experiments

The detailed results obtained through additional experiments are again provided in the appendix. The measurements include the results for CSPs (Section A.3.1) and ontologies (Section A.3.2), as well as results that were obtained by systematically varying the characteristics of synthetic diagnosis problems (Section A.3.3). The results indicate that applying a depth-first parallelization strategy in many cases is advantageous for the CSP problems. The tests on the ontology problems and the simulation results however reveal that depending on the problem structure there are cases in which a breadth-first strategy can be more beneficial.

5.3.3 Discussion

The experiments show that the parallelization of the depth-first search strategy (PRDFS) can help to further reduce the computation times when we search for one single diagnosis.

In most evaluated cases, PRDFS was faster than its sequential counterpart. In some cases, however, the obtained improvements were quite small or virtually non-existent, which can be explained as follows.

- For the very small scenarios, the parallel depth-first search cannot be significantly faster than the non-parallel variant because the creation of the *first* node is not parallelized. Therefore a major fraction of the tree construction process is not parallelized at all.

- There are problem settings in which all existing diagnoses have the same size. All parallel depth-first searching threads therefore have to explore the tree to a certain depth and none of the threads can immediately return a diagnosis that is much smaller than one determined by another thread. E.g., given a diagnosis problem, where all diagnoses have size 5, all threads have to explore the tree to at least level 5 to find a diagnosis and are also very likely to find a diagnosis on that level. Therefore, in this setting no thread can be much faster than the others.

- Finally, we again suspect problems of cache contention and a correspondingly increased number of cache misses, which leads to a general performance deterioration and overhead caused by the multiple threads.

Overall, the obtained speedups again depend on the problem structure. The hybrid technique represents a good compromise for most cases as it is faster than the sequential breadth first search approach for most of the tested scenarios (including the CSPs, ontologies, and synthetically created diagnosis problems presented in Section A.3). Also, it is more efficient than PRDFS in some cases for which breadth first search is better than depth first search.

6. Parallel Direct CSP Encodings

As an alternative to conflict-guided diagnosis approaches like Reiter's hitting set technique, so-called "direct encodings" have become more popular in the research community in recent years (Feldman, Provan, de Kleer, Robert, & van Gemund, 2010a; Stern, Kalech, Feldman, & Provan, 2012; Metodi et al., 2014; Mencia & Marques-Silva, 2014; Mencía, Previti, & Marques-Silva, 2015; Marques-Silva, Janota, Ignatiev, & Morgado, 2015).[10]

The general idea of direct encodings is to generate a specific representation of a diagnosis problem instance with some knowledge representation language and then use the theorem prover (e.g., a SAT solver or constraint engine) to compute the diagnoses directly. These methods support the generation of one or multiple diagnoses by calling a theorem prover only once. Nica, Pill, Quaritsch, and Wotawa (2013) made a number of experiments in which they compared conflict-directed search with such direct encodings and showed that for several problem settings, using the direct encoding was advantageous.

In this part of the paper, our goal is to evaluate whether the parallelization of the search process – in that case inside the constraint engine – can help to improve the efficiency of the diagnostic reasoning process. The goal of this chapter is therefore rather to *quantify* to which extent the internal parallelization of a solver is useful than to present a new algorithmic contribution.

10. Such direct encodings may not always be possible in MBD settings as discussed above.

6.1 Using Gecode as a Solver for Direct Encodings

For our evaluation we use the Gecode constraint solver (Schulte, Lagerkvist, & Tack, 2016). In particular, we use the parallelization option of Gecode to test its effects on the diagnosis running times.[11] The chosen problem encoding is similar to the one used by Nica and Wotawa (2012). This allows us to make our results comparable with those obtained in previous works. In addition, the provided encoding is represented in a language which is supported by multiple solvers.

6.1.1 Example

Let us first show the general idea on a small example. Consider the following CSP[12] consisting of the integer variables $a1$, $a2$, $b1$, $b2$, $c1$ and the constraints X_1, X_2, and X_3 which are defined as:

$$X_1 :\ b1 = a1 \times 2, X_2 :\ b2 = a2 \times 3, X_3 :\ c1 = b1 \times b2.$$

Let us assume that the programmer made a mistake and X_3 should actually be $c1 = b1 + b2$. Given a set of expected observations (a test case) $a1 = 1, a2 = 6, d1 = 20$, MBD can be applied by considering the constraints as the possibly faulty components.

In a direct encoding the given CSP is extended with a definition of an array $AB = [ab_1, ab_2, ab_3]$ of boolean (0/1) variables which encode whether a corresponding constraint is considered as faulty or not. The constraints are rewritten as follows:

$$X_1' :\ ab_1 \vee (b1 = a1 \times 2), \quad X_2' :\ ab_2 \vee (b2 = a2 \times 3), \quad X_3' :\ ab_3 \vee (c1 = b1 \times b2).$$

The observations can be encoded through equality constraints which bind the values of the observed variables. In our example, these constraints would be:

$$O_1 :\ a1 = 1, \quad O_2 :\ a2 = 6, \quad O_3 :\ d1 = 20$$

In order to find a diagnosis of cardinality 1, we additionally add the constraint

$$ab_1 + ab_2 + ab_3 = 1$$

and let the solver search for a solution. In this case, X_3 would be identified as the only possible diagnosis, i.e., ab_3 would be set to "1" by the solver.

6.1.2 Parallelization Approach of Gecode

When using such a direct encoding, a parallelization of the diagnosis process, as shown for Reiter's approach, cannot be done because it is embedded in the underlying search procedure. However, modern constraint solvers, such as *Gecode*, *or-tools* and many other solvers of those that participated in the MiniZinc Challenge (Stuckey, Feydy, Schutt, Tack, & Fischer, 2014), internally implement parallelization strategies to better utilize today's multi-core computer architectures (Michel, See, & Van Hentenryck, 2007; Chu, Schulte, &

11. A state-of-the-art SAT solver capable of parallelization could have been used for this analysis as well.
12. Adapted from an earlier work (Jannach & Schmitz, 2014).

Stuckey, 2009). In the following, we will therefore evaluate through a set of experiments, if these solver-internal parallelization techniques can help to speed up the diagnosis process when a direct encoding is used.[13]

Gecode implements an adaptive work stealing strategy (Chu et al., 2009) for its parallelization. The general idea can be summarized as follows. As soon as a thread finishes processing its nodes of the search tree, it "steals" some of the nodes from non-idle threads. In order to decide from which thread the work should be stolen, an adaptive strategy uses balancing heuristics that estimate the density of the solutions in a particular part of the search tree. The higher the likelihood of containing a solution for a given branch, the more work is stolen from this branch.

6.2 Problem Encoding

In our evaluation we use MiniZinc as a constraint modeling language. This language can be processed by different solvers and allows us to model diagnosis problems as CSPs as shown above.

6.2.1 Finding One Diagnosis

To find a single diagnosis for a given diagnosis problem (SD, Comps, Obs), we generate a direct encoding in MiniZinc as follows.

(1) For the set of components Comps we generate an array `ab = [`$ab_1, \dots, ab_n$`]` of boolean variables.

(2) For each formula $sd_i \in$ SD we add a constraint of the form
    `constraint` $ab[i] \vee (sd_i)$;
and for each observation $o_j \in$ Obs the model is extended with a constraint
    `constraint` o_j;

(3) Finally, we add the search goal and an output statement:
    `solve minimize sum(`i `in` $1..n$`)(bool2int(`$ab[i]$`));`
    `output[show(ab)];`

The first statement of the last part (`solve minimize`), instructs the solver to search for a (single) solution with a minimal number of abnormal components, i.e., a diagnosis with minimum cardinality. The second statement (`output`) projects all assignments to the set of abnormal variables, because we are only interested in knowing which components are faulty. The assignments of the other problem variables are irrelevant.

6.2.2 Finding All Diagnoses

The problem encoding shown above can be used to quickly find one/all diagnoses of minimum cardinality. It is, however, not sufficient for scenarios where the goal is to find all diagnoses of a problem. We therefore propose the following sound and complete algorithm which repeatedly modifies the constraint problem to systematically identify all diagnoses.

13. In contrast to the parallelization approaches presented in the previous sections, we do not propose any new parallelization schemes here but rather rely on the existing ones implemented in the solver.

Technically, the algorithm first searches for all diagnoses of size 1 and then increases the desired cardinality of the diagnoses step by step.

Algorithm 11: directDiag: Computation of all diagnoses using a direct encoding.

```
Input: A diagnosis problem (SD, Comps, Obs), maximum cardinality $k$
Result: The set $\Delta$ of diagnoses
$\Delta = \emptyset$; $\mathcal{C} = \emptyset$; $card = 1$;
if $k > |\text{Comps}|$ then $k = |\text{Comps}|$;
$\mathcal{M}$ = generateModel (SD, Comps, Obs);
while $card \leqslant k$ do
    $\mathcal{M}$ = updateModel ($\mathcal{M}$, $card$, $\mathcal{C}$);
    $\Delta'$ = computeDiagnoses($\mathcal{M}$);
    $\mathcal{C} = \mathcal{C} \cup$ generateConstraints($\Delta'$);
    $\Delta = \Delta \cup \Delta'$;
    $card = card + 1$;
return $\Delta$;
```

Procedure Algorithm 10 shows the main components of the direct diagnosis method used in connection with a parallel constraint solver to find all diagnoses. The algorithm starts with the generation of a MiniZinc model (generateModel) as described above. The only difference is that we will now search for all solutions of a given cardinality; further details about the encoding of the search goals are given below.

In each iteration, the algorithm modifies the model by updating the cardinality of the searched diagnoses and furthermore adds new constraints corresponding to the already found diagnoses (updateModel). This updated model is then provided to a MiniZinc interpreter (constraint solver), which returns a set of solutions Δ'. Each element $\delta_i \in \Delta'$ corresponds to a diagnosis of the cardinality *card*.

In order to exclude supersets of the already found diagnoses Δ' in future iterations, we generate a constraint for each $\delta_i \in \Delta'$ with the formulas j to l (generateConstraints):

$$\texttt{constraint}\ ab[j] = \mathit{false} \vee \cdots \vee ab[l] = \mathit{false};$$

These constraints ensure that an already found diagnosis or supersets of it cannot be found again. They are added to the model $\mathcal{M}$ in the next iteration of the main loop. The algorithm continues until all diagnoses with cardinalities up to k are computed.

Changes in Encoding To calculate all diagnoses of a given size, we first instruct the solver to search for all possible solutions when provided with a constraint problem.[14] In addition, while keeping steps (1) and (2) from Section 6.2.1 we replace the lines of step (3)

14. This is achieved by calling `MiniZinc` with the `--all-solutions` flag.

by the following statements:

```
constraint sum(i in 1..n)(bool2int(ab[i])) = card;
solve satisfy;
output[show(ab)];
```

The first statement constrains the number of *abnormal* variables that can be true to a certain value, i.e., the given cardinality *card*. The second statement tells the solver to find *all* variable assignments that satisfy the constraints. The last statement again guarantees that the solver only considers the solutions to be different when they are different with respect to the assignments of the abnormal variables.

Soundness and Completeness Algorithm 10 implements an iterative deepening approach which guarantees the minimality of the diagnoses in Δ. Specifically, the algorithm constructs diagnoses in the order of increasing cardinality by limiting the number of *ab* variables that can be set to *true* in a model. The computation starts with $card = 1$, which means that only one *ab* variable can be true. Therefore, only diagnoses of cardinality 1, i.e., comprising only one abnormal variable, can be returned by the solver. For each found diagnosis we then add a constraint that requires at least one of the abnormal variables of this diagnosis to be *false*. Therefore, neither this diagnosis nor its supersets can be found in the subsequent iterations. These constraints implement the pruning rule of the HS-tree algorithm. Finally, Algorithm 10 repeatedly increases the cardinality parameter *card* by one and continues with the next iteration. The algorithm continues to increment the cardinality until *card* becomes greater than the number of components, which corresponds to the largest possible cardinality of a diagnosis. Consequently, given a diagnosis problem as well as a sound and complete constraint solver, Algorithm 10 returns all diagnoses of the problem.

6.3 Evaluation

To evaluate if speedups can be achieved through parallelization also for a direct encoding, we again used the first five systems of the DXC Synthetic Track and tested all scenarios using the Gecode solver without parallelization and with 2 and 4 parallel threads.

6.3.1 Results

We evaluated two different configurations. In setup (A), the task was to find one single diagnosis of minimum cardinality. In setup (B), the iterative deepening procedure from Section 6.2.2 was used to find all diagnoses up to the size of the actual error.

The results for setup (A) are shown in Table 6. We can observe that using the parallel constraint solver pays off except for the tiny problems for which the overall search time is less than 200 ms. Furthermore, adding more worker threads is also beneficial for the larger problem sizes and a speedup of up to 1.25 was achieved for the most complex test case which took about 1.5 seconds to solve.

The same pattern can be observed for setup (B). The detailed results are listed in Table 7. For the tiny problems, the internal parallelization of the Gecode solver does not lead to performance improvements but slightly slows down the whole process. As soon as the problems become more complex, parallelization pays off and we can observe a speedup of 1.55 for the most complex of the tested cases, which corresponds to a runtime reduction of 35%.

System	**Direct Encoding**				
	Abs. [ms]	$\mathbf{S}_2$	$\mathbf{E}_2$	$\mathbf{S}_4$	$\mathbf{E}_4$
74182	**27**	0.85	0.42	0.79	0.20
74L85	**30**	0.89	0.44	0.79	0.20
74283	**32**	0.85	0.43	0.79	0.20
74181	200	1.04	0.52	**1.15**	0.29
c432	1,399	1.17	0.58	**1.25**	0.31

Table 6: Observed performance gains for DXC benchmarks for finding *one diagnosis* with a direct encoding using one (column Abs.), two, and four threads.

System	**Direct Encoding**				
	Abs. [ms]	$\mathbf{S}_2$	$\mathbf{E}_2$	$\mathbf{S}_4$	$\mathbf{E}_4$
74182	**136**	0.84	0.42	0.80	0.20
74L85	**60**	0.83	0.41	0.77	0.19
74283	**158**	0.93	0.47	0.92	0.23
74181	1,670	1.19	0.59	**1.33**	0.33
c432	229,869	1.22	0.61	**1.55**	0.39

Table 7: Observed performance gains for DXC benchmarks for finding *all diagnoses* with a direct encoding using one (column Abs.), two, and four threads.

6.3.2 Summary and Remarks

Overall, our experiments show that parallelization can be beneficial when a direct encoding of the diagnosis problem is employed, in particular when the problems are non-trivial.

Comparing the absolute running times of our Java implementation using the open source solver Choco with the optimized C++ implementation of Gecode is generally not appropriate and for most of the benchmark problems, Gecode works faster on an absolute scale. Note, however, that this is not true in all cases. In particular when searching for all diagnoses up to the size of the actual error for the most complex system c432, even Reiter's non-parallelized Hitting Set algorithm was much faster (85 seconds) than using the direct encoding based on iterative deepening (230 seconds). This is in line with the observation of Nica et al. (2013) that direct encodings are not always the best choice when searching for all diagnoses.

A first analysis of the run-time behavior of Gecode shows that the larger the problem is, the more time is spent by the solver in each iteration to reconstruct its internal structures, which can lead to a measurable performance degradation. Note that in our work we relied on a MiniZinc encoding of the diagnosis problem to be independent of the specifics of the underlying constraint engine. An implementation that relies on the direct use of the API of a specific CSP solver might help to address certain performance issues. Nevertheless, such an implementation must be solver-specific and will not allow us to switch solvers easily as it is now possible with MiniZinc..

7. Relation to Previous Works

In this section we explore works that are related to our approach. First we examine different approaches for the computation of diagnoses. Then we will focus on general methods for parallelizing search algorithms.

7.1 Computation of Diagnoses

Computing minimal hitting sets for a given set of conflicts is a computationally hard problem as already discussed in Section 2.2.2 and several approaches were proposed over the years to deal with the issue. These approaches can be divided into exhaustive and approximate ones. The former perform a sound and complete search for all minimal diagnoses, whereas the latter improve the computational efficiency in exchange for completeness, e.g., they search for only one or a small set of diagnoses.

Approximate approaches can for example be based on stochastic search techniques like genetic algorithms (Li & Yunfei, 2002) or greedy stochastic search (Feldman et al., 2010b). The greedy method proposed by Feldman et al. (2010b), for example, uses a two-step approach. In the first phase, a random and possibly non-minimal diagnosis is determined by a modified DPLL[15] algorithm. The algorithm always finds one random diagnosis at each invocation due to the random selection of propositional variables and their assignments. In the second step, the algorithm minimizes the diagnosis returned by the DPLL technique by repeatedly applying random modifications. It randomly chooses a negative literal which denotes that a corresponding component is faulty and flips its value to positive. The obtained candidate as well as the diagnosis problem are provided to the DPLL algorithm to check whether the candidate is a diagnosis or not. In case of success the obtained diagnosis is kept and another random flip is done. Otherwise, the negative literal is labeled with "failure" and another negative literal is randomly selected. The algorithm stops if the number of "failures" is greater than some predefined constant and returns the best diagnosis found so far.

In the approach of Li and Yunfei (2002) a genetic algorithm takes a number of conflict sets as input and generates a set of bit-vectors (chromosomes), where every bit encodes a truth value of an atom over the AB(.) predicate. In each iteration the algorithm applies genetic operations, such as mutation, crossover, etc., to obtain new chromosomes. Subsequently, all obtained bit-vectors are evaluated by a "hitting set" fitting function which eliminates bad candidates. The algorithm stops after a predefined number of iterations and returns the best diagnosis.

In general, such approximate approaches are not directly comparable with our LWP and FP techniques, since they are incomplete and do not guarantee the minimality of returned

15. Davis-Putnam-Logemann-Loveland.

hitting sets. Our goal in contrast is to improve the performance while at the same time maintaining both the completeness and the soundness property.

Another way of finding approximate solutions is to use heuristic search approaches. For example, Abreu and van Gemund (2009) proposed the Staccato algorithm which applies a number of heuristics for pruning the search space. More "aggressive" pruning techniques result in better performance of the search algorithms. However, they also increase the probability that some of the diagnoses will not be found. In this approach the "aggressiveness" of the heuristics can be varied by input parameters depending on the application goals.

More recently, Cardoso and Abreu (2013) suggested a distributed version of the Staccato algorithm, which is based on the Map-Reduce scheme (Dean & Ghemawat, 2008) and can therefore be executed on a cluster of servers. Other more recent algorithms focus on the efficient computation of one or more minimum cardinality (*minc*) diagnoses (de Kleer, 2011). Both in the distributed approach and in the minimum cardinality scenario, the assumption is that the (possibly incomplete) set of conflicts is already available as an input at the beginning of the hitting-set construction process. In the application scenarios that we address with our work, finding the conflicts is considered to be the computationally expensive part and we do not assume to know the minimal conflicts in advance but have to compute them "on-demand" as also done in other works (Felfernig, Friedrich, Jannach, Stumptner, et al., 2000; Friedrich & Shchekotykhin, 2005; Williams & Ragno, 2007); see also the work by Pill, Quaritsch, and Wotawa (2011) for a comparison of conflict computation approaches.

Exhaustive approaches are often based on HS-trees like the work of Wotawa (2001a) – a tree construction algorithm that reduces the number of pruning steps in presence of non-minimal conflicts. Alternatively, one can use methods that compute diagnoses without the explicit computation of conflict sets, i.e., by solving a problem dual to minimal hitting sets (Satoh & Uno, 2005). Stern et al. (2012), for example, suggest a method that explores the duality between conflicts and diagnoses and uses this symmetry to guide the search. Other approaches exploit the structure of the underlying problem, which can be hierarchical (Autio & Reiter, 1998), tree-structured (Stumptner & Wotawa, 2001), or distributed (Wotawa & Pill, 2013). These algorithms are very similar to the HS-tree algorithm and, consequently, can be parallelized in a similar way. As an example, consider the Set-Enumeration Tree (SE-tree) algorithm (Rymon, 1994). This algorithm, similarly to Reiter's HS-tree approach, uses breadth-first search with a specific expansion procedure that implements the pruning and node selection strategies. Both the LWP and and the FP parallelization variant can be used with the SE-tree algorithm and comparable speedups are expected.

7.2 Parallelization of Search Algorithms

Historically, the parallelization of search algorithms was approached in three different ways (Burns, Lemons, Ruml, & Zhou, 2010):

(i) *Parallelization of node processing*: When applying this type of parallelization, the tree is expanded by one single process, but the computation of labels or the evaluation of heuristics is done in parallel.

(ii) *Window-based processing*: In this approach, sets of nodes, called “windows”, are processed by different threads in parallel. The windows are formed by the search algorithm according to some predefined criteria.

(iii) *Tree decomposition approaches*: Here, different sub-trees of the search tree are assigned to different processes (Ferguson & Korf, 1988; Brüngger, Marzetta, Fukuda, & Nievergelt, 1999).

In principle, all three types of parallelization can be applied in some form to the HS-tree generation problem.

Applying strategy (i) in the MBD problem setting would mean to parallelize the process of conflict computation, e.g., through a parallel variant of QXP or MXP. We have tested a partially parallelized version of MXP, which however did not lead to further performance improvements when compared to a single-threaded approach on the evaluated benchmark problems (Shchekotykhin et al., 2015). The experiments in Section 4 however show that using MXP in combination with LWP or FP – thereby implicitly allocating more CPU time for the computation of multiple conflicts during the construction of a single node – can be advantageous. Other well-known conflict or prime implicate computation algorithms (Junker, 2004; Marques-Silva et al., 2013; Previti, Ignatiev, Morgado, & Marques-Silva, 2015) in contrast were not designed for parallel execution or the computation of multiple conflicts.

Strategy (ii) – computing sets of nodes (windows) in parallel – was for example applied by Powley and Korf (1991). In their work the windows are determined by different thresholds of a heuristic function of *Iterative Deepening A**. Applying the strategy to an HS-tree construction problem would mean to categorize the nodes to be expanded according to some criterion, e.g., the probability of finding a diagnosis, and to allocate the different groups to individual threads. In the absence of such window criteria, LWP and FP could be seen as extreme cases with window size one, where each open node is allocated to one thread on a processor. The experiments done throughout the paper suggest that independent of the parallelization strategy (LWP or FP) the number of parallel threads (windows) should not exceed the number of physically available computing threads to obtain the best performance.

Finally (iii), the strategy exploring different sub-trees during the search with different processes can, for example, be applied in the context of MBD techniques when using Binary HS-Tree (BHS) algorithms (Pill & Quaritsch, 2012). Given a set of conflict sets, the BHS method generates a root node and labels it with the input set of conflicts. Then, it selects one of the components occurring in the conflicts and generates two child nodes, such that the left node is labeled with all conflicts comprising the selected component and the right node with the remaining ones. Consequently, the diagnosis tree is decomposed into two subtrees and can be processed in parallel. The main problem for this kind of parallelization is that the conflicts are often not known in advance and have to be computed during search.

Anglano and Portinale (1996) suggested another approach in which they ultimately parallelized the diagnosis problem based on structural problem characteristics. In their work, they first map a given diagnosis problem to a Behavioral Petri Net (BPN). Then, the obtained BPN is manually partitioned into subnets and every subnet is provided to a different Parallel Virtual Machine (PVM) for parallel processing. The relationship of their work to our LWP and FP parallelization schemes is limited and our approaches also do not require a manual problem decomposition step.

In general, parallelized versions of domain-independent search algorithms like A^* can be applied to MBD settings. However, the MBD problem has some specifics that make the application of some of these algorithms difficult. For instance, the PRA* method and its variant HDA* discussed in the work of Burns et al. (2010) use a mechanism to minimize the memory requirements by retracting parts of the search tree. These "forgotten" parts are later on re-generated when required. In our MBD setting, the generation of nodes is however the most costly part, which is why the applicability of HDA* seems limited. Similarly, duplicate detection algorithms like PBNF (Burns et al., 2010) require the existence of an abstraction function that partitions the original search space into blocks. In general MBD settings, we however cannot assume that such a function is given.

In order to improve the performance we have therefore to avoid the parallel generation of duplicate nodes by different threads, which we plan to investigate in our future work. A promising starting point for this research could be the work by Phillips, Likhachev, and Koenig (2014). The authors suggest a variant of the A* algorithm that generates only independent nodes in order to reduce the costs of node generation. Two nodes are considered as independent if the generation of one node does not lead to a change of the heuristic function of the other node. The generation of independent nodes can be done in parallel without the risk of the repeated generation of an already known state. The main difficulty when adopting this algorithm for MBD is the formulation of an admissible heuristic required to evaluate the independence of the nodes for arbitrary diagnosis problems. However, for specific problems that can be encoded as CSPs, Williams and Ragno (2007) present a heuristic that depends on the number of unassigned variables at a particular search node.

Finally, parallelization was also used in the literature to speed up the processing of very large search trees that do not fit in memory. Korf and Schultze (2005), for instance, suggest an extension of a hash-based delayed duplicate detection algorithm that allows a search algorithm to continue search while other parts of the search tree are written to or read from the hard drive. Such methods can in theory be used in combination with our LWP or FP parallelization schemes in case of complex diagnosis problems. We plan to explore the use of (externally) saved search states in the context of MBD as part of our future works.

8. Summary

In this work, we propose and systematically evaluate various parallelization strategies for Model-Based Diagnosis to better exploit the capabilities of multi-core computers. We show that parallelization can be advantageous in various problem settings and diagnosis approaches. These approaches include the conflict-driven search for all or a few minimal diagnoses with different conflict detection techniques and the (heuristic) depth-first search in order to quickly determine a single diagnosis. The main benefits of our parallelization approaches are that they can be applied independent of the underlying reasoning engine and for a variety of diagnostic problems which cannot be efficiently represented as SAT or CSP problems. In addition to our HS-tree based parallelization approaches, we also show that parallelization can be beneficial for settings in which a direct problem encoding is possible and modern parallel solver engines are available.

Our evaluations have furthermore shown that the speedups of the proposed parallelization methods can vary according to the characteristics of the underlying diagnosis problem.

In our future work, we plan to explore techniques that analyze these characteristics in order to predict in advance which parallelization method is best suited to find one single or all diagnoses for the given problem.

Regarding algorithmic enhancements, we furthermore plan to investigate how information about the underlying problem structure can be exploited to achieve a better distribution of the work on the parallel threads and to thereby avoid duplicate computations. Furthermore, we plan to explore the usage of parallel solving schemes for the dual algorithms, i.e., algorithms that compute diagnoses directly without the computation of minimal conflicts (Satoh & Uno, 2005; Felfernig, Schubert, & Zehentner, 2012; Stern et al., 2012; Shchekotykhin et al., 2014).

The presented algorithms were designed for the use on modern multi-core computers which today usually have less than a dozen cores. Our results show that the *additional* performance improvements that we obtain with the proposed techniques become smaller when adding more and more CPUs. As part of our future works we therefore plan to develop algorithms that can utilize specialized environments that support massive parallelization. In that context, a future topic of research could be the adaption of the parallel HS-tree construction to GPU architectures. GPUs, which can have thousands of computing cores, have proved to be superior for tasks which can be parallelized in a suitable way. Campeotto, Palù, Dovier, Fioretto, and Pontelli (2014) for example used a GPU to parallelize a constraint solver. However, it is not yet fully clear whether tree construction techniques can be efficiently parallelized on a GPU, as many data structures have to be shared across all nodes and access to them has to be synchronized.

Acknowledgements

This paper significantly extends and combines our previous work (Jannach, Schmitz, & Shchekotykhin, 2015; Shchekotykhin et al., 2015).

We would like to thank Håkan Kjellerstrand and the Gecode team for their support. We are also thankful for the various helpful comments and suggestions made by the anonymous reviewers of JAIR, DX'14, DX'15, AAAI'15, and IJCAI'15.

This work was supported by the Carinthian Science Fund (KWF) contract KWF-3520/26767/38701, the Austrian Science Fund (FWF) and the German Research Foundation (DFG) under contract numbers I 2144 N-15 and JA 2095/4-1 (Project "Debugging of Spreadsheet Programs").

Appendix A.

In this appendix we report the results of additional experiments that were made on different benchmark problems as well as results of simulation experiments on artificially created problem instances.

- Section A.1 contains the results for the LWP and FP parallelization schemes proposed in Section 3.
- Section A.2 reports additional measurements regarding the use of MergeXplain within the parallel diagnosis process, see Section 4.

- Section A.3 finally provides additional results of the parallelization of the depth-first strategies discussed in Section 5.

A.1 Additional Experiments for the LWP and FP Parallelization Strategies

In addition to the experiments with the DXC benchmark systems reported in Section 3.5, we made additional experiments with Constraint Satisfaction Problems, ontologies, and artificial Hitting Set construction problems. Furthermore, we examined the effects of further increasing the number of available threads for the benchmarks of the CSPs and ontologies.

A.1.1 Diagnosing Constraint Satisfaction Problems

Data Sets and Procedure In this set of experiments we used a number of CSP instances from the 2008 CP solver competition (Lecoutre, Roussel, & van Dongen, 2008) in which we injected faults.[16] The diagnosis problems were created as follows. We first generated a random solution using the original CSP formulations. From each solution, we randomly picked about 10% of the variables and stored their value assignments, which then served as test cases. These stored variable assignments correspond to the *expected outcomes* when all constraints are formulated correctly. Next, we manually inserted errors (mutations) in the constraint problem formulations[17], e.g., by changing a "less than" operator to a "more than" operator, which corresponds to a mutation-based approach in software testing. The diagnosis task then consists of identifying the possibly faulty constraints using the partial test cases. In addition to the benchmark CSPs we converted a number of spreadsheet diagnosis problems (Jannach & Schmitz, 2014) to CSPs to test the performance gains on realistic application settings.

Table 8 shows the problem characteristics including the number of injected faults (#F), the number of diagnoses (#D), and the average diagnosis size ($\overline{|D|}$). In general, we selected CSPs which are quite diverse with respect to their size.

Results The measurement results using 4 threads and searching for all diagnoses are given in Table 9. Improvements could be achieved for all problem instances. With the exception of the smallest problem *mknap-1-5* all speedups achieved by LWP and FP are statistically significant. For some problems, the improvements are very strong (with a running time reduction of over 50%), whereas for others the improvements are modest. On average, FP is also faster than LWP. However, FP is not consistently better than LWP and often the differences are small.

The observed results indicate that the performance gains depend on a number of factors including the size of the conflicts, the computation times for conflict detection, and the problem structure itself. While on average FP is faster than LWP, the characteristics of the problem settings seem to have a considerable impact on the speedups that can be obtained by the different parallelization strategies.

16. To be able to do a sufficient number of repetitions, we picked instances with comparably small running times.
17. The mutated CSPs can be downloaded at `http://ls13-www.cs.tu-dortmund.de/homepage/hp_downloads/jair/csps.zip`.

Scenario	#C	#V	#F	#D	$\overline{\lvert D \rvert}$
c8	523	239	8	4	6.25
costasArray-13	87	88	2	2	2.5
domino-100-100	100	100	3	81	2
graceful–K3-P2	60	15	4	117	2.94
mknap-1-5	7	39	1	2	1
queens-8	28	8	15	9	10.9
hospital payment	38	75	4	120	3.8
profit calculation	28	140	5	42	4.24
course planning	457	583	2	3024	2
preservation model	701	803	1	22	1
revenue calculation	93	154	4	1452	3

Table 8: Characteristics of selected problem settings.

Scenario	**Seq.(QXP)**	**LWP(QXP)**		**FP(QXP)**	
	[ms]	**S$_4$**	**E$_4$**	**S$_4$**	**E$_4$**
c8	559	**1.10**	**0.27**	1.07	0.27
costasArray-13	4,013	2.16	0.54	**2.58**	**0.65**
domino-100-100	1,386	**3.08**	**0.77**	3.05	0.76
graceful–K3-P2	1,965	2.75	0.69	**2.99**	**0.75**
mknap-1-5	314	**1.03**	**0.26**	1.02	0.25
queens-8	141	1.57	0.39	**1.65**	**0.41**
hospital payment	12,660	1.64	0.41	**1.73**	**0.43**
profit calculation	197	1.71	0.43	**2.00**	**0.50**
course planning	22,130	2.58	0.65	**2.61**	**0.65**
preservation model	167	1.46	0.37	**1.48**	**0.37**
revenue calculation	778	**2.81**	**0.70**	2.58	0.64

Table 9: Results for CSP benchmarks and spreadsheets when searching for all diagnoses.

A.1.2 Diagnosing Ontologies

Data Sets and Procedure In recent works, MBD techniques are used to locate faults in description logic ontologies (Friedrich & Shchekotykhin, 2005; Shchekotykhin et al., 2012; Shchekotykhin & Friedrich, 2010), which are represented in the Web Ontology Language (OWL) (Grau, Horrocks, Motik, Parsia, Patel-Schneider, & Sattler, 2008). When testing such an ontology, the developer can – similarly to an earlier approach (Felfernig, Friedrich, Jannach, Stumptner, & Zanker, 2001) – specify a set of "positive" and "negative" test cases. The test cases are sets of logical sentences which must be *entailed* by the ontology (positive) or *not entailed* by the ontology (negative). In addition, the ontology itself, which is a set of logical sentences, has to be consistent and coherent (Baader, Calvanese, McGuinness, Nardi, & Patel-Schneider, 2010). A diagnosis (debugging) problem in this context arises, if one of these requirements is not fulfilled.

In the work by Shchekotykhin et al. (2012), two interactive debugging approaches were tested on a set of faulty real-world ontologies (Kalyanpur, Parsia, Horridge, & Sirin, 2007) and two randomly modified large real-world ontologies. We use the same dataset to evaluate the performance gains when applying our parallelization schemes to the ontology debugging problem. The details of the different tested ontologies are given in Table 10. The characteristics of the problems are described in terms of the description logic (DL) used to formulate the ontology, the number of axioms (#A), concepts (#C), properties (#P), and individuals (#I). In terms of the first-order logic, concepts and properties correspond to unary and binary predicates, whereas individuals correspond to constants. Every letter of a DL name, such as $\mathcal{ALCHF}^{(D)}$, corresponds to a syntactic feature of the language. E.g., $\mathcal{ALCHF}^{(D)}$ is an $\mathcal{A}$ttributive concept $\mathcal{L}$anguage with $\mathcal{C}$omplement, properties $\mathcal{H}$ierarchy, $\mathcal{F}$unctional properties and Datatypes. As an underlying description logic reasoner, we used Pellet (Sirin, Parsia, Grau, Kalyanpur, & Katz, 2007). The manipulation of the knowledge bases during the diagnosis process was accomplished with the OWL-API (Horridge & Bechhofer, 2011).

Note that the considered ontology debugging problem is different from the other diagnosis settings discussed so far as it cannot be efficiently encoded as a CSP or SAT problem. The reason is that the decision problems, such as the checking of consistency and concept satisfiability, for the ontologies given in Table 10 are ExpTime-complete (Baader et al., 2010). This set of experiments therefore helps us to explore the benefits of parallelization for problem settings in which the computation of conflict sets is very hard. Furthermore, the application of the parallelization approaches on the ontology debugging problem demonstrates the generality of our methods, i.e., we show that our methods are applicable to a wide range of diagnosis problems and only require the existence of a sound and complete consistency checking procedure.

Due to the generality of Reiter's general approach and, correspondingly, our implementation of the diagnosis procedures, the technical integration of the OWL-DL reasoner into our software framework is relatively simple. The only difference to the CSP-based problems is that instead of calling Choco's *solve()* method inside the Theorem Prover, we make a call to the Pellet reasoner via the OWL-API to check the consistency of an ontology.

Ontology	DL	#A	#C/#P/#I	#D	$\overline{\lvert D \rvert}$
Chemical	$\mathcal{ALCHF}^{(D)}$	144	48/20/0	6	1.67
Koala	$\mathcal{ALCON}^{(D)}$	44	21/5/6	10	2.3
Sweet-JPL	$\mathcal{ALCHOF}^{(D)}$	2,579	1,537/121/50	13	1
miniTambis	$\mathcal{ALCN}$	173	183/44/0	48	3
University	$\mathcal{SOIN}^{(D)}$	49	30/12/4	90	3.67
Economy	$\mathcal{ALCH}^{(D)}$	1,781	339/53/482	864	7.17
Transportation	$\mathcal{ALCH}^{(D)}$	1,300	445/93/183	1,782	8
Cton	$\mathcal{SHF}$	33,203	17,033/43/0	15	4
Opengalen-no-propchains	$\mathcal{ALCHIF}^{(D)}$	9,664	4,713/924/0	110	4.13

Table 10: Characteristics of the tested ontologies.

Results The obtained results – again using a thread pool of size four – are shown in Table 11. Again, in every case parallelization is advantageous when compared to the sequential version and in some cases the obtained speedups are substantial. Regarding the comparison of the LWP and FP variants, there is no clear winner across all test cases. LWP seems to be advantageous for most of the problems that are more complex with respect to their computation times. For the problems that can be easily solved, FP is sometimes slightly better. A clear correlation between other problem characteristics like the complexity of the knowledge base in terms of its size could not be identified within this set of benchmark problems.

Ontology	**Seq.(QXP)**	**LWP(QXP)**		**FP(QXP)**	
	[ms]	**S_4**	**E_4**	**S_4**	**E_4**
Chemical	237	**1.44**	**0.36**	1.33	0.33
Koala	16	**1.42**	**0.36**	1.27	0.32
Sweet-JPL	7	1.47	0.37	**1.55**	**0.39**
miniTambis	135	1.43	0.36	**1.46**	**0.37**
University	85	1.66	0.41	**1.68**	**0.42**
Economy	355	**2.20**	**0.55**	1.90	0.48
Transportation	1,696	**2.72**	**0.68**	2.33	0.58
Cton	203	**1.27**	**0.32**	1.22	0.30
Opengalen-no-propchains	11,044	1.59	0.40	**1.86**	**0.47**

Table 11: Results for ontologies when searching for all diagnoses.

A.1.3 Adding More Threads

Constraint Satisfaction Problems Table 12 shows the results of the CSP benchmarks and spreadsheets when using up to 12 threads. In this test utilizing more than 4 threads was advantageous in all but one small scenario. However, for 7 of the 11 tested scenarios doing the computations with more than 8 threads did not pay off. This indicates that choosing the right degree of parallelization can depend on the characteristics of a diagnosis problem. The diagnosis of the *mknap-1-5* problem, for example, cannot be sped up with parallelization as it only contains one single conflict that is found at the root node. In contrast, the *graceful-K3-P2* problem benefits from the use of up to 12 threads and we could achieve a speedup of 4.21 for this scenario, which corresponds to a runtime reduction of 76%.

Ontologies The results of diagnosing the ontologies with up to 12 threads are shown in Table 13. For the tested ontologies, which are comparably simple debugging cases, using more than 4 threads payed off in only 3 of 7 cases. The best results when diagnosing these 3 ontologies were obtained when 8 threads were used. For one ontology using more than 4 threads was even slower than the sequential algorithm. This again indicates that the effectiveness of parallelization depends on the characteristics of the diagnosis problem and adding more threads can be even slightly counterproductive.

A.1.4 Systematic Variation of Problem Characteristics

Procedure To better understand in which way the problem characteristics influence the performance gains, we used a suite of artificially created hitting set construction problems

Scenario	**Seq.(QXP)**	**FP(QXP)**							
	[ms]	$\mathbf{S}_4$	$\mathbf{E}_4$	$\mathbf{S}_8$	$\mathbf{E}_8$	$\mathbf{S}_{10}$	$\mathbf{E}_{10}$	$\mathbf{S}_{12}$	$\mathbf{E}_{12}$
c8	444	1.05	0.26	1.07	0.13	**1.08**	**0.11**	1.07	0.09
costasArray-13	3,854	2.69	0.67	**2.88**	**0.36**	2.84	0.28	2.80	0.23
domino-100-100	213	2.04	0.51	**2.30**	**0.29**	2.22	0.22	2.00	0.17
graceful–K3-P2	1,743	3.03	0.76	4.12	0.51	4.18	0.42	**4.21**	**0.35**
mknap-1-5	4,141	1.00	0.25	1.00	0.13	1.00	0.10	**1.00**	**0.08**
queens-8	86	1.18	0.30	**1.30**	**0.16**	1.24	0.12	1.19	0.10
hospital payment	11,728	1.60	0.40	**1.70**	**0.21**	1.51	0.15	1.36	0.11
profit calculation	81	1.53	0.38	**1.59**	**0.20**	1.51	0.15	1.44	0.12
course planning	15,323	2.31	0.58	**2.85**	**0.36**	2.84	0.28	2.73	0.23
preservation model	127	1.34	0.34	1.41	0.18	1.41	0.14	**1.43**	**0.12**
revenue calculation	460	**2.39**	**0.60**	2.17	0.27	1.96	0.20	1.85	0.15

Table 12: Observed performance gains for the CSP benchmarks and spreadsheets on a server with 12 hardware threads.

Ontology	**Seq.(QXP)**	**FP(QXP)**							
	[ms]	$\mathbf{S}_4$	$\mathbf{E}_4$	$\mathbf{S}_8$	$\mathbf{E}_8$	$\mathbf{S}_{10}$	$\mathbf{E}_{10}$	$\mathbf{S}_{12}$	$\mathbf{E}_{12}$
Chemical	246	**1.37**	**0.34**	1.29	0.16	1.30	0.13	1.32	0.11
Koala	21	**1.07**	**0.27**	1.02	0.13	1.03	0.10	0.99	0.08
Sweet-JPL	6	1.09	0.27	**1.13**	**0.14**	1.08	0.11	1.02	0.09
miniTambis	134	1.47	0.37	**1.49**	**0.19**	1.47	0.15	1.45	0.12
University	88	1.53	0.38	**1.64**	**0.21**	1.56	0.16	1.56	0.13
Economy	352	**1.48**	**0.37**	0.90	0.11	0.76	0.08	0.71	0.06
Transportation	1,448	**1.74**	**0.43**	1.23	0.15	1.07	0.11	1.09	0.09

Table 13: Observed performance gains for the ontologies on a server with 12 hardware threads.

with the following varying parameters: number of components (#Cp), number of conflicts (#Cf), average size of conflicts ($\overline{|\text{Cf}|}$). Given these parameters, we used a problem generator which produces a set of minimal conflicts with the desired characteristics. The generator first creates the given number of components and then uses these components to generate the requested number of conflicts.

To obtain more realistic settings, not all generated conflicts were of equal size but rather varied according to a Gaussian distribution with the desired size as a mean. Similarly, not all components should be equally likely to be part of a conflict and we again used a Gaussian distribution to assign component failure probabilities. Other probability distributions could be used in the generation process as well, e.g., to reflect specifics of a certain application domain.

Since for this experiment all conflicts are known in advance, the conflict detection algorithm within the consistency check only has to return one suitable conflict upon request. Because zero computation times are unrealistic and our assumption is that the conflict detection is actually the most costly part of the diagnosis process, we varied the assumed conflict computation times to analyze their effect on the relative performance gains. These computation times were simulated by adding artificial active *waiting times* (Wt) inside the consistency check (shown in ms in Table 14). Note that the consistency check is only called if no conflict can be reused for the current node; the artificial waiting time only applies to cases in which a new conflict has to be determined.

Each experiment was repeated 100 times on different variations of each problem setting to factor out random effects. The number of diagnoses #D is thus an average as well. All algorithms had, however, to solve identical sets of problems and thus returned identical sets of diagnoses. We limited the search depth to 4 for all experiments to speed up the benchmark process. The average running times are reported in Table 14.

Results – Varying Computation Times First, we varied the assumed conflict computation times for a quite small diagnosis problem using 4 parallel threads (Table 14). The first row with assumed zero computation times shows how long the HS-tree construction alone needs. The improvements of the parallelization are smaller for this case because of the overhead of thread creation and synchronization. However, as soon as we add an average running time of 10ms for the consistency check, both parallelization approaches result in a speedup of about 3, which corresponds to a runtime reduction of 67%. Further increasing the assumed computation time does not lead to better relative improvements using the pool of 4 threads.

Results – Varying Conflict Sizes The average conflict size impacts the breadth of the HS-tree. Next, we therefore varied the average conflict size. Our hypothesis was that larger conflicts and correspondingly broader HS-trees are better suited for parallel processing. The results shown in Table 14 confirm this assumption. FP is always slightly more efficient than LWP. Average conflict sizes larger than 9 did, however, not lead to strong additional improvements when using 4 threads.

Results – Adding More Threads For larger conflicts, adding additional threads leads to further improvements. Using 8 threads results in improvements of up to 7.27 (corresponding to a running time reduction of over 85%) for these larger conflict sizes because in these cases even higher levels of parallelization can be achieved.

Results – Adding More Components Finally, we varied the problem complexity by adding more components that can potentially be faulty. Since we left the number and size of the conflicts unchanged, adding more components led to diagnoses that included more different components. As we limited the search depth to 4 for this experiment, fewer diagnoses were found up to this level and the search trees were narrower. As a result, the relative performance gains were lower than when there are fewer components (constraints).

Discussion The simulation experiments demonstrate the advantages of parallelization. For all tests, the speedups of LWP and FP are statistically significant. The results also confirm that the performance gains depend on different characteristics of the underlying problem. The additional gains of not waiting at the end of each search level for all worker threads to be finished typically led to small further improvements.

Redundant calculations can, however, still occur, in particular when the conflicts for new nodes are determined in parallel and two worker threads return the same conflict.

#Cp, #Cf, $\overline{\|Cf\|}$	#D	Wt [ms]	**Seq.** [ms]	**LWP** $\mathbf{S_4}$	**LWP** $\mathbf{E_4}$	**FP** $\mathbf{S_4}$	**FP** $\mathbf{E_4}$
Varying computation times Wt							
50, 5, 4	25	**0**	23	2.26	0.56	**2.58**	**0.64**
50, 5, 4	25	**10**	483	2.98	0.75	**3.10**	**0.77**
50, 5, 4	25	**100**	3,223	2.83	0.71	**2.83**	**0.71**
Varying conflict sizes							
50, 5, **6**	99	10	1,672	3.62	0.91	**3.68**	**0.92**
50, 5, **9**	214	10	3,531	3.80	0.95	**3.83**	**0.96**
50, 5, **12**	278	10	4,605	3.83	0.96	**3.88**	**0.97**
Varying numbers of components							
50, 10, 9	201	10	3,516	**3.79**	**0.95**	3.77	0.94
75, 10, 9	105	10	2,223	**3.52**	**0.88**	3.29	0.82
100, 10, 9	97	10	2,419	3.13	0.78	**3.45**	**0.86**
#Cp, #Cf, ∅|Cf|	#D	Wt [ms]	**Seq.** [ms]	**LWP** $\mathbf{S_8}$	**LWP** $\mathbf{E_8}$	**FP** $\mathbf{S_8}$	**FP** $\mathbf{E_8}$
Adding more threads (8 instead of 4)							
50, 5, **6**	99	10	1,672	6.40	0.80	**6.50**	**0.81**
50, 5, **9**	214	10	3,531	7.10	0.89	**7.15**	**0.89**
50, 5, **12**	278	10	4,605	7.25	0.91	**7.27**	**0.91**

Table 14: Simulation results.

Although without parallelization the computing resources would have been left unused anyway, redundant calculations can lead to overall longer computation times for very small problems because of the thread synchronization overheads.

A.2 Additional Experiments Using MXP for Conflict Detection

In this section we report the additional results that were obtained when using MergeXplain instead of QuickXplain as a conflict detection strategy as described in Section 4.2. The different experiments were again made using a set of CSPs and ontology debugging problems. Remember that in this set of experiments our goal is to identify a set of leading diagnoses.

A.2.1 Diagnosing Constraint Satisfaction Problems

Table 15 shows the results when searching for five diagnoses using the CSP and spreadsheet benchmarks. MXP could again help to reduce the running times for most of the tested scenarios except for some of the smaller ones. For the tiny scenario mknap-1-5, the simple sequential algorithm using QXP is the fastest alternative. For most of the other scenarios, however, parallelization pays off and is faster than when sequentially expanding the search tree. The best result could be achieved for the scenario *costasArray-13*, where FP using MXP reduced the running times by 83% compared to the sequential algorithm using QXP, which corresponds to a speedup of 6. The results again indicate that FP works well for both QXP and MXP.

Scenario	**Seq.(QXP)**	**FP(QXP)**		**Seq.(MXP)**	**FP(MXP)**	
	[ms]	**S$_4$**	**E$_4$**	**[ms]**	**S$_4$**	**E$_4$**
c8	455	1.03	0.26	251	**1.06**	**0.26**
costasArray-13	2,601	3.66	0.91	2,128	**4.92**	**1.23**
domino-100-100	53	1.26	0.32	50	**1.43**	**0.36**
graceful–K3-P2	528	2.67	0.67	419	**2.48**	**0.62**
mknap-1-5	**19**	0.99	0.25	21	1.01	0.25
queens-8	75	1.55	0.39	63	**1.67**	**0.42**
hospital payment	1,885	1.17	0.29	1,426	**1.28**	**0.32**
profit calculation	33	**1.92**	**0.48**	40	1.86	0.46
course planning	1,522	0.99	0.25	1,188	**1.42**	**0.35**
preservation model	411	**1.50**	**0.37**	430	1.50	0.37
revenue calculation	48	1.21	0.30	42	**1.48**	**0.37**

Table 15: Results for CSP benchmarks and spreadsheets (QXP vs MXP).

Note that in one case (costasArray-13) we see an efficiency value larger than one, which means that the obtained speedup is super-linear. This can happen in special situations in which we search for a limited number of diagnoses and use the FP method (see also Section A.3.1). Assume that generating one specific node takes particularly long, i.e., the computation of a conflict set requires a considerable amount of time. In that case, a sequential algorithm will be "stuck" at this node for some time, while the FP method will continue generating other nodes. If these other nodes are then sufficient to find the (limited) required number of diagnoses, this can lead to an efficiency value that is greater than the theoretical optimum.

A.2.2 Diagnosing Ontologies

The results are shown in Table 16. Similar to the previous experiment, using MXP in combination with FP pays off in all cases except for the very simple benchmark problems.

A.3 Additional Experiments – Parallel Depth-First Search

In this section, we report the results of additional experiments that were made to assess the effects of parallelizing a depth-first search strategy as described in Section 5.3. In this set of experiments the goal was to find one single minimal diagnosis. We again report the results obtained for the constraint problems and the ontology debugging problems and discuss the findings of a simulation experiment in which we systematically varied the problem characteristics.

A.3.1 Diagnosing Constraint Satisfaction Problems

The results of searching for a single diagnosis for the CSPs and spreadsheets are shown in Table 17. Again, parallelization generally shows to be a good strategy to speed up the

Ontology	**Seq.(QXP)**	**FP(QXP)**		**Seq.(MXP)**	**FP(MXP)**	
	[ms]	$\mathbf{S}_4$	$\mathbf{E}_4$	[ms]	$\mathbf{S}_4$	$\mathbf{E}_4$
Chemical	187	2.10	0.53	144	**1.94**	**0.48**
Koala	15	**1.49**	**0.37**	13	1.27	0.32
Sweet-JPL	5	**1.27**	**0.32**	4	1.05	0.26
miniTambis	68	1.04	0.26	56	**1.08**	**0.27**
University	33	1.05	0.26	26	**1.02**	**0.26**
Economy	19	1.10	0.27	14	**1.00**	**0.25**
Transportation	71	1.08	0.27	53	**1.10**	**0.27**
Cton	174	1.36	0.34	154	**1.33**	**0.33**
Opengalen-no-propchains	2,145	1.22	0.30	1,748	**1.35**	**0.34**

Table 16: Results for Ontologies (QXP vs MXP).

diagnosis process. All measured speedups except the speedup of RDFS for the first scenario *c8* are statistically significant. In this specific problem setting, only the FP strategy had a measurable effect and for some strategies even a modest performance deterioration was observed when compared to Reiter's sequential algorithm. The reason lies in the resulting structure of the HS-tree which is very narrow as most conflicts are of size one.

The following detailed observations can be made when comparing the algorithms.

- In most of the tested CSPs, FP is advantageous when compared to RDFS and PRDFS.
- For the spreadsheets, in contrast, RDFS or PRDFS were better than the breadth-first approach of FP in three of five cases.
- When comparing RDFS and PRDFS, we can again observe that parallelization can be advantageous also for these depth-first strategies.
- Again, however, the improvements seem to depend on the underlying problem structure. In the case of the *hospital payment* scenario, the speedup of PRDFS is as high as 3.1 compared to the sequential algorithm, which corresponds to a runtime reduction of more than 67%. The parallel strategy is, however, not consistently better for all test cases.
- The performance of the Hybrid method again lies in between the performances of its two components for many, but not all, of the tested scenarios.

A.3.2 Diagnosing Ontologies

Next, we evaluated the search for one diagnosis on the real-world ontologies (Table 18). In the tested scenarios, applying the depth-first strategy did often not pay off when compared to the breadth-first methods. The reason is that in the tested examples from the ontology debugging domain in many cases single-element diagnoses exist, which can be quickly detected by a breadth-first strategy. Furthermore the absolute running times are often comparably small. Parallelizing the depth-first strategy leads to significant speedups in some but not all cases.

Scenario	Seq.	FP		RDFS	PRDFS		Hybrid	
	[ms]	S_4	E_4	[ms]	S_4	E_4	S_4	E_4
c8	462	**1.09**	**0.27**	454	0.89	0.22	0.92	0.23
costasArray-13	1,996	**4.78**	**1.19**	3,729	3.42	0.85	5.90	1.47
domino-100-100	57	1.22	0.30	45	**1.17**	**0.29**	1.05	0.26
graceful–K3-P2	372	**2.86**	**0.71**	305	2.01	0.50	1.89	0.47
mknap-1-5	166	**2.18**	**0.55**	114	1.02	0.26	1.35	0.33
queens-8	72	**1.38**	**0.34**	55	1.02	0.26	0.95	0.24
hospital payment	263	1.83	0.46	182	**2.14**	**0.54**	1.72	0.43
profit calculation	99	**1.67**	**0.42**	70	1.15	0.29	1.10	0.28
course planning	3,072	1.11	0.28	**2,496**	0.90	0.23	0.87	0.22
preservation model	182	**1.78**	**0.44**	104	0.99	0.25	0.95	0.24
revenue calculation	152	1.11	0.28	**121**	0.92	0.23	0.90	0.22

Table 17: Results for CSP benchmarks and spreadsheets for finding one diagnosis.

Ontology	Seq.	FP		RDFS	PRDFS		Hybrid	
	[ms]	S_4	E_4	[ms]	S_4	E_4	S_4	E_4
Chemical	73	**2.18**	**0.54**	57	1.62	0.41	1.47	0.37
Koala	10	**2.20**	**0.55**	9	1.93	0.48	1.39	0.35
Sweet-JPL	**3**	0.92	0.23	4	0.97	0.24	0.92	0.23
miniTambis	**58**	0.95	0.24	62	0.92	0.23	0.93	0.23
University	29	**1.06**	**0.27**	30	1.03	0.26	1.03	0.26
Economy	17	**1.10**	**0.27**	18	1.16	0.29	1.10	0.27
Transportation	65	1.03	0.26	61	**1.03**	**0.26**	0.98	0.24

Table 18: Observed performance gains for ontologies for finding one diagnosis.

A.3.3 Systematic Variation of Problem Characteristics

Table 19 finally shows the simulation results when searching for one single diagnosis. In the experiment we used a uniform probability distribution when selecting the components of the conflicts to obtain more complex diagnosis problems. The results can be summarized as follows.

- FP is as expected better than the sequential version of the HS-tree algorithm for all tested configurations.
- For the very small problems that contain only a few and comparably small conflicts, the depth-first strategy does not work well. Both the parallel and sequential versions are even slower than Reiter's original proposal, except for cases where zero conflict computation times are assumed. This indicates that the costs for hitting set minimization are too high.
- For the larger problem instances, relying on a depth-first strategy to find one single diagnosis is advantageous and also better than FP. An additional test with an even

#Cp, #Cf, ∅|Cf|	∅|D|	Wt [ms]	**Seq. [ms]**	**FP**		**RDFS [ms]**	**PRDFS**		**Hybrid**	
				S$_4$	**E**$_4$		**S**$_4$	**E**$_4$	**S**$_4$	**E**$_4$
Varying computation times Wt										
50, 5, 4	3.40	**0**	11	2.61	0.65	2	**1.01**	**0.25**	0.85	0.21
50, 5, 4	3.40	**10**	89	**1.50**	**0.37**	155	1.28	0.32	2.24	0.56
50, 5, 4	3.40	**100**	572	**1.50**	**0.37**	1,052	1.30	0.33	2.26	0.56
Varying conflict sizes										
50, 5, **6**	2.86	10	90	**1.57**	**0.39**	143	1.26	0.31	2.12	0.53
50, 5, **9**	2.36	10	86	**1.55**	**0.39**	138	1.34	0.33	2.04	0.51
50, 5, **12**	2.11	10	83	**1.61**	**0.40**	124	1.23	0.31	1.95	0.49
Varying numbers of components										
50, 10, 9	3.47	10	229	**2.36**	**0.59**	202	1.35	0.34	1.65	0.41
75, 10, 9	3.97	10	570	3.09	0.77	228	1.37	0.34	**1.42**	**0.36**
100, 10, 9	4.34	10	1,467	2.37	0.59	240	**1.34**	**0.33**	1.26	0.31
More conflicts										
100, **12**, 9	5.00	10	26,870	1.28	0.32	280	**1.39**	**0.35**	1.24	0.31

Table 19: Simulation results for finding one diagnosis.

larger problem shown in the last line of Table 19 reveals the potential of a depth-first search approach.

- When the problems are larger, PRDFS can again help to obtain further runtime improvements compared to RDFS.
- The Hybrid method works well for all but the single case with zero computation times. Again, it represents a good choice when the problem structure is not known.

Overall, the simulation experiments show that the speedups that can be achieved with the different methods depend on the underlying problem structure also when we search for one single diagnosis.

References

Abreu, R., & van Gemund, A. J. C. (2009). A Low-Cost Approximate Minimal Hitting Set Algorithm and its Application to Model-Based Diagnosis. In *SARA'09*, pp. 2–9.

Anglano, C., & Portinale, L. (1996). Parallel model-based diagnosis using PVM. In *EuroPVM'96*, pp. 331–334.

Autio, K., & Reiter, R. (1998). Structural Abstraction in Model-Based Diagnosis. In *ECAI'98*, pp. 269–273.

Baader, F., Calvanese, D., McGuinness, D., Nardi, D., & Patel-Schneider, P. (2010). *The Description Logic Handbook: Theory, Implementation and Applications*, Vol. 32.

Bolosky, W. J., & Scott, M. L. (1993). False Sharing and Its Effect on Shared Memory Performance. In *SEDMS'93*, pp. 57–71.

Brüngger, A., Marzetta, A., Fukuda, K., & Nievergelt, J. (1999). The parallel search bench ZRAM and its applications. *Annals of Operations Research*, *90*(0), 45–63.

Buchanan, B., & Shortliffe, E. (Eds.). (1984). *Rule-based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project.* Addison-Wesley, Reading, MA.

Burns, E., Lemons, S., Ruml, W., & Zhou, R. (2010). Best-First Heuristic Search for Multicore Machines. *Journal of Artificial Intelligence Research*, *39*, 689–743.

Campeotto, F., Palù, A. D., Dovier, A., Fioretto, F., & Pontelli, E. (2014). Exploring the Use of GPUs in Constraint Solving. In *PADL'14*, pp. 152–167.

Cardoso, N., & Abreu, R. (2013). A Distributed Approach to Diagnosis Candidate Generation. In *EPIA'13*, pp. 175–186.

Chandra, D., Guo, F., Kim, S., & Solihin, Y. (2005). Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *HPCA'11*, pp. 340–351.

Chu, G., Schulte, C., & Stuckey, P. J. (2009). Confidence-Based Work Stealing in Parallel Constraint Programming. In *CP'09*, pp. 226–241.

Console, L., Friedrich, G., & Dupré, D. T. (1993). Model-Based Diagnosis Meets Error Diagnosis in Logic Programs. In *IJCAI'93*, pp. 1494–1501.

de Kleer, J. (2011). Hitting set algorithms for model-based diagnosis. In *DX'11*, pp. 100–105.

Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, *51*(1), 107–113.

Dijkstra, E. W. (1968). The Structure of the "THE"-Multiprogramming System. *Communications of the ACM*, *11*(5), 341–346.

Eiter, T., & Gottlob, G. (1995). The Complexity of Logic-Based Abduction. *Journal of the ACM*, *42*(1), 3–42.

Feldman, A., Provan, G., de Kleer, J., Robert, S., & van Gemund, A. (2010a). Solving model-based diagnosis problems with max-sat solvers and vice versa. In *DX'10*, pp. 185–192.

Feldman, A., Provan, G., & van Gemund, A. (2010b). Approximate Model-Based Diagnosis Using Greedy Stochastic Search. *Journal of Artifcial Intelligence Research*, *38*, 371–413.

Felfernig, A., Friedrich, G., Isak, K., Shchekotykhin, K. M., Teppan, E., & Jannach, D. (2009). Automated debugging of recommender user interface descriptions. *Applied Intelligence*, *31*(1), 1–14.

Felfernig, A., Friedrich, G., Jannach, D., & Stumptner, M. (2004). Consistency-based diagnosis of configuration knowledge bases. *Artificial Intelligence*, *152*(2), 213–234.

Felfernig, A., Friedrich, G., Jannach, D., Stumptner, M., & Zanker, M. (2001). Hierarchical diagnosis of large configurator knowledge bases. In *KI'01*, pp. 185–197.

Felfernig, A., Schubert, M., & Zehentner, C. (2012). An efficient diagnosis algorithm for inconsistent constraint sets. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, *26*(1), 53–62.

Felfernig, A., Friedrich, G., Jannach, D., Stumptner, M., et al. (2000). Consistency-based diagnosis of configuration knowledge bases. In *ECAI'00*, pp. 146–150.

Ferguson, C., & Korf, R. E. (1988). Distributed tree search and its application to alpha-beta pruning. In *AAAI'88*, pp. 128–132.

Friedrich, G., & Shchekotykhin, K. M. (2005). A General Diagnosis Method for Ontologies. In *ISWC'05*, pp. 232–246.

Friedrich, G., Stumptner, M., & Wotawa, F. (1999). Model-Based Diagnosis of Hardware Designs. *Artificial Intelligence*, *111*(1-2), 3–39.

Friedrich, G., Fugini, M., Mussi, E., Pernici, B., & Tagni, G. (2010). Exception handling for repair in service-based processes. *IEEE Transactions on Software Engineering*, *36*(2), 198–215.

Friedrich, G., & Shchekotykhin, K. (2005). A General Diagnosis Method for Ontologies. In *ISWC'05*, pp. 232–246.

Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co.

Grau, B. C., Horrocks, I., Motik, B., Parsia, B., Patel-Schneider, P., & Sattler, U. (2008). OWL 2: The next step for OWL. *Web Semantics: Science, Services and Agents on the World Wide Web*, *6*(4), 309–322.

Greiner, R., Smith, B. A., & Wilkerson, R. W. (1989). A Correction to the Algorithm in Reiter's Theory of Diagnosis. *Artificial Intelligence*, *41*(1), 79–88.

Horridge, M., & Bechhofer, S. (2011). The OWL API: A Java API for OWL Ontologies. *Semantic Web Journal*, *2*(1), 11–21.

Jannach, D., & Schmitz, T. (2014). Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach. *Automated Software Engineering*, *February 2014* (published online).

Jannach, D., Schmitz, T., & Shchekotykhin, K. (2015). Parallelized Hitting Set Computation for Model-Based Diagnosis. In *AAAI'15*, pp. 1503–1510.

Junker, U. (2004). QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems. In *AAAI'04*, pp. 167–172.

Kalyanpur, A., Parsia, B., Horridge, M., & Sirin, E. (2007). Finding all justifications of owl dl entailments. In *The Semantic Web*, Vol. 4825 of *Lecture Notes in Computer Science*, pp. 267–280.

Korf, R. E., & Schultze, P. (2005). Large-scale parallel breadth-first search. In *AAAI'05*, pp. 1380–1385.

Kurtoglu, T., & Feldman, A. (2011). Third International Diagnostic Competition (DXC 11). `https://sites.google.com/site/dxcompetition2011`. Accessed: 2016-03-15.

Lecoutre, C., Roussel, O., & van Dongen, M. R. C. (2008). CPAI08 competition. `http://www.cril.univ-artois.fr/CPAI08/`. Accessed: 2016-03-15.

Li, L., & Yunfei, J. (2002). Computing Minimal Hitting Sets with Genetic Algorithm. In *DX'02*, pp. 1–4.

Marques-Silva, J., Janota, M., Ignatiev, A., & Morgado, A. (2015). Efficient Model Based Diagnosis with Maximum Satisfiability. In *IJCAI'15*, pp. 1966–1972.

Marques-Silva, J., Janota, M., & Belov, A. (2013). Minimal Sets over Monotone Predicates in Boolean Formulae. In *Computer Aided Verification*, pp. 592–607.

Mateis, C., Stumptner, M., Wieland, D., & Wotawa, F. (2000). Model-Based Debugging of Java Programs. In *AADEBUG'00*.

Mencia, C., & Marques-Silva, J. (2014). Efficient Relaxations of Over-constrained CSPs. In *ICTAI'14*, pp. 725–732.

Mencía, C., Previti, A., & Marques-Silva, J. (2015). Literal-based MCS extraction. In *IJCAI'15*, pp. 1973–1979.

Metodi, A., Stern, R., Kalech, M., & Codish, M. (2014). A novel sat-based approach to model based diagnosis. *Journal of Artificial Intelligence Research*, *51*, 377–411.

Michel, L., See, A., & Van Hentenryck, P. (2007). Parallelizing constraint programs transparently. In *CP'07*, pp. 514–528.

Nica, I., Pill, I., Quaritsch, T., & Wotawa, F. (2013). The route to success: a performance comparison of diagnosis algorithms. In *IJCAI'13*, pp. 1039–1045.

Nica, I., & Wotawa, F. (2012). ConDiag - computing minimal diagnoses using a constraint solver. In *DX'12*, pp. 185–191.

Phillips, M., Likhachev, M., & Koenig, S. (2014). PA*SE: Parallel A* for Slow Expansions. In *ICAPS'14*.

Pill, I., Quaritsch, T., & Wotawa, F. (2011). From conflicts to diagnoses: An empirical evaluation of minimal hitting set algorithms. In *DX'11*, pp. 203–211.

Pill, I., & Quaritsch, T. (2012). Optimizations for the Boolean Approach to Computing Minimal Hitting Sets. In *ECAI'12*, pp. 648–653.

Powley, C., & Korf, R. E. (1991). Single-agent parallel window search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *13*(5), 466–477.

Previti, A., Ignatiev, A., Morgado, A., & Marques-Silva, J. (2015). Prime Compilation of Non-Clausal Formulae. In *IJCAI'15*, pp. 1980–1987.

Prud'homme, C., Fages, J.-G., & Lorca, X. (2015). *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. `http://www.choco-solver.org`.

Reiter, R. (1987). A Theory of Diagnosis from First Principles. *Artificial Intelligence*, *32*(1), 57–95.

Rymon, R. (1994). An SE-tree-based prime implicant generation algorithm. *Annals of Mathematics and Artificial Intelligence*, *11*(1-4), 351–365.

Satoh, K., & Uno, T. (2005). Enumerating Minimally Revised Specifications Using Dualization. In *JSAI'05*, pp. 182–189.

Schulte, C., Lagerkvist, M., & Tack, G. (2016). GECODE - An open, free, efficient constraint solving toolkit. `http://www.gecode.org`. Accessed: 2016-03-15.

Shchekotykhin, K., Friedrich, G., Fleiss, P., & Rodler, P. (2012). Interactive ontology debugging: Two query strategies for efficient fault localization. *Journal of Web Semantics*, *1213*, 88–103.

Shchekotykhin, K. M., & Friedrich, G. (2010). Query strategy for sequential ontology debugging. In *ISWC'10*, pp. 696–712.

Shchekotykhin, K., Jannach, D., & Schmitz, T. (2015). MergeXplain: Fast Computation of Multiple Conflicts for Diagnosis. In *IJCAI'15*, pp. 3221–3228.

Shchekotykhin, K. M., Friedrich, G., Rodler, P., & Fleiss, P. (2014). Sequential diagnosis of high cardinality faults in knowledge-bases by direct diagnosis generation. In *ECAI'14*, pp. 813–818.

Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., & Katz, Y. (2007). Pellet: A Practical OWL-DL Reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, *5*(2), 51 – 53.

Stern, R., Kalech, M., Feldman, A., & Provan, G. (2012). Exploring the Duality in Conflict-Directed Model-Based Diagnosis. In *AAAI'12*, pp. 828–834.

Stuckey, P. J., Feydy, T., Schutt, A., Tack, G., & Fischer, J. (2014). The MiniZinc Challenge 2008-2013. *AI Magazine*, *35*(2), 55–60.

Stumptner, M., & Wotawa, F. (1999). Debugging functional programs. In *IJCAI'99*, pp. 1074–1079.

Stumptner, M., & Wotawa, F. (2001). Diagnosing Tree-Structured Systems. *Artificial Intelligence*, *127*(1), 1–29.

White, J., Benavides, D., Schmidt, D. C., Trinidad, P., Dougherty, B., & Cortés, A. R. (2010). Automated diagnosis of feature model configurations. *Journal of Systems and Software*, *83*(7), 1094–1107.

Williams, B. C., & Ragno, R. J. (2007). Conflict-directed A* and its role in model-based embedded systems. *Discrete Applied Mathematics*, *155*(12), 1562–1595.

Wotawa, F. (2001a). A variant of Reiter's hitting-set algorithm. *Information Processing Letters*, *79*(1), 45–51.

Wotawa, F. (2001b). Debugging Hardware Designs Using a Value-Based Model. *Applied Intelligence*, *16*(1), 71–92.

Wotawa, F., & Pill, I. (2013). On classification and modeling issues in distributed model-based diagnosis. *AI Communications*, *26*(1), 133–143.

Efficient Sequential Model-Based Fault-Localization with Partial Diagnoses

[Placeholder]

Kostyantyn Shchekotykhin
Alpen-Adria University Klagenfurt, Austria
kostyantyn.shchekotykhin@aau.at

Thomas Schmitz
TU Dortmund, Germany
thomas.schmitz@tu-dortmund.de

Dietmar Jannach
TU Dortmund, Germany
dietmar.jannach@tu-dortmund.de

IJCAI '16, July 9–15, 2016, New York, NY, USA

ISBN: 978-1-57735-770-4

Finding Errors in the Enron Spreadsheet Corpus

Thomas Schmitz
TU Dortmund
44221 Dortmund, Germany
thomas.schmitz@udo.edu

Dietmar Jannach
TU Dortmund
44221 Dortmund, Germany
dietmar.jannach@udo.edu

***Abstract*—Spreadsheet environments like MS Excel are the most widespread type of end-user software development tools and spreadsheet-based applications can be found almost everywhere in organizations. Since spreadsheets are prone to error, several approaches were proposed in the research literature to help users locate formula errors. However, the proposed methods were often designed based on assumptions about the nature of errors and were evaluated with mutations of correct spreadsheets.**

In this work we propose a method and tool to identify real-world formula errors within the Enron spreadsheet corpus. Our approach is based on heuristics that help us identify versions of the same spreadsheet and our software helps the user identify spreadsheets of which we assume that they contain error corrections. An initial manual inspection of a subset of such candidates led to the identification of more than two dozen formula errors. We publicly share the new collection of real-world spreadsheet errors.

I. Introduction

Spreadsheets are used almost everywhere and at all levels of organizations [1]. They are often used for financial calculations and planning purposes so that errors in the calculations can have severe impacts for organizations [2]. Errors in spreadsheets are unfortunately not uncommon, in particular because spreadsheets are often developed by end-users with no education in software development. Already in the late 1990s a survey showed that in many studies on spreadsheet errors at least one fault[1] was found in every analyzed spreadsheet [4].

Different approaches to avoid spreadsheet errors are possible, starting with better training for end-users or defined quality procedures for spreadsheets. Over the years, also a variety of proposals for better *tool support* were made in the literature [3], ranging from visualization approaches [5], over environments that support systematic tests [6], to interactive debugging aids [7]. Many of the proposed error detection and correction tools focus on errors in individual *formulas*.

A common challenge when designing and evaluating such approaches is that not many real-world spreadsheets with known formula errors are available. Although larger collections of real-world spreadsheets exist, usually no information about the contained errors is given [8], [9]. To evaluate novel test and debugging techniques researchers therefore often inject errors into real-world or artificial spreadsheets using, e.g., the set of mutation operators for spreadsheets proposed in [10]. Such mutations can represent a useful approximation of the true errors that are made by users. Nonetheless, these mutation-based evaluations are based on certain assumptions about the types and frequency of different types of errors.

In 2015, Hermans and Murphy-Hill [11] published a new corpus of spreadsheets extracted from the publicly available emails of Enron, a huge US-company that went bankrupt in 2001 ("Enron scandal"). The new corpus comprises 15,770 spreadsheets that were created for productive use and of which 9,120 contain formulas. Again, however, no information is available about the errors that these spreadsheets contain.

In this paper, we therefore propose a method and publish a tool [12] to locate formula errors in spreadsheets of the Enron corpus. To find such errors, we first try to identify different *versions* of the same spreadsheet in the corpus, where one version contains a fix to a bug that existed in the previous version. We use different heuristics to detect such spreadsheet versions. In one strategy we reconstruct parts of the email conversations in which spreadsheets were exchanged and look for indicators in the email texts which suggest that the enclosed spreadsheet contains a bug fix. All spreadsheets that are attached in this conversation are then automatically checked for differences. In another approach we look for spreadsheets whose names are similar or slightly different and, e.g., contain a suffix like "_v2" or "_fixed". We then again compute the differences between these files. If only one or a few formulas were changed, these files represent *candidate* spreadsheets, which can then be manually inspected for errors.

Determining if a change of a formula represents a bug fix or rather implements an updated business logic is hard to automate as one has to understand the intended semantics of each formula. We therefore implemented a visual tool that automatically retrieves the different versions of a spreadsheet and supports the user in inspecting them. With the help of this tool we identified several spreadsheet errors of different types using only a limited set of heuristics. We publicly share our collection of errors to foster future research in the field [13].

II. Technical Approach

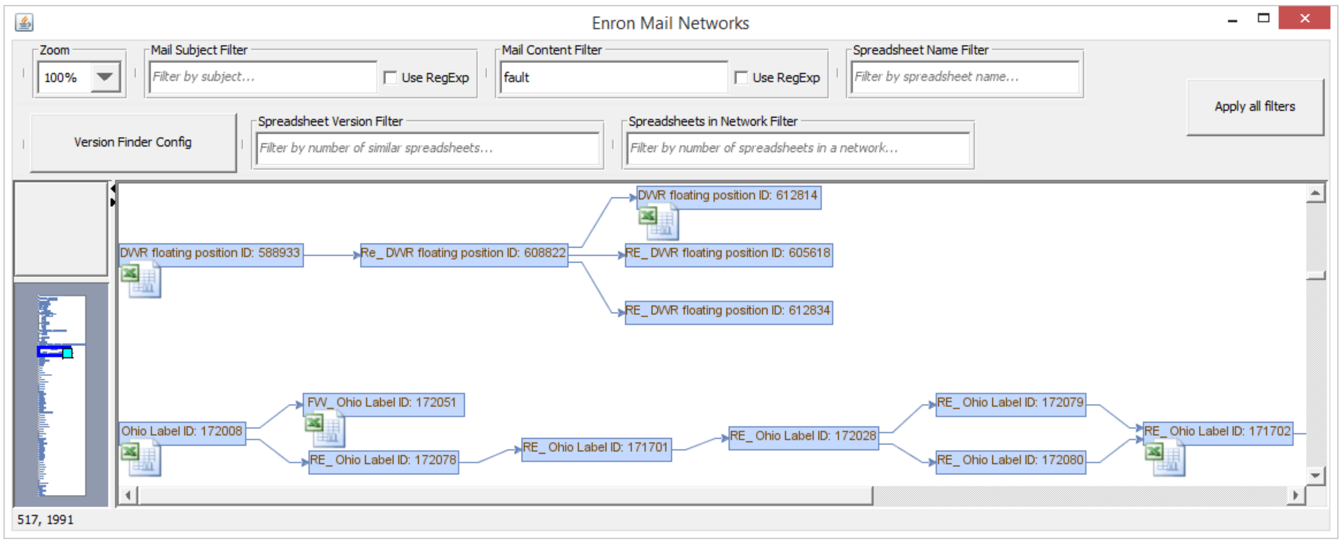
In this section we present how we reconstruct the email conversations and how we analyze differences in spreadsheets.

A. Reconstruction of Email Conversations

To identify emails that discuss errors in the attached spreadsheets, we propose to reconstruct the email conversations.

[1]In this paper we use the terms error and fault in an interchangeable manner. A discussion of the usage of the different terms can be found in [3].

978-1-5090-0252-8/16/$31.00 ©2016 IEEE


Zoom: 100%

Mail Content Filter: fault

517, 1991
Enron Mail Networks
Mail Subject Filter
Filter by subject...
Use RegExp
Use RegExp
Spreadsheet Name Filter
Filter by spreadsheet name...
Apply all filters
Version Finder Config
Spreadsheet Version Filter
Filter by number of similar spreadsheets...
Spreadsheets in Network Filter
Filter by number of spreadsheets in a network...
DWR floating position ID: 612814
DWR floating position ID: 588933
Re_ DWR floating position ID: 608822
RE_ DWR floating position ID: 605618
RE_ DWR floating position ID: 612834
FW_ Ohio Label ID: 172051
RE_ Ohio Label ID: 172079
Ohio Label ID: 172008
RE_ Ohio Label ID: 172028
RE_ Ohio Label ID: 171702
RE_ Ohio Label ID: 171701
RE_ Ohio Label ID: 172078
RE_ Ohio Label ID: 172080

Fig. 1. A screenshot of our interactive tool for finding errors in the Enron corpus.

1) General Idea: Figure 1 shows the interactive visualization of such a conversation in our tool. Nodes in the graph correspond to emails and the edges represent that, for example, one email was sent in reply to another.

Our tool reconstructs such conversations using different heuristics. With the implemented heuristics we created 13,440 conversation graphs that had at least one spreadsheet attached. 1,100 of them consisted of two or more nodes. In our tool, individual keywords like “fix” or “error” as well as complex regular expressions can be used to filter those conversations that contain these keywords in the subject line, email text, or as part of a spreadsheet name.

The conversation and the attached spreadsheets can then be manually inspected one by one. To support the user in this manual process, the tool automatically determines and displays the exact differences between each spreadsheet of the conversation. If the number of differences between two files is very small and, e.g., only one single formula was changed, this might be an indicator of a possible bug fix.

Our approach of searching for certain terms in email conversations is inspired by [11], who found over 4,000 emails in the Enron corpus which had a spreadsheet attached and contained one of several keywords like *error* or *mistake*. Retrieving emails with certain keywords is however not sufficient for our purpose, as our goal is to find different versions of one spreadsheet to be able to identify possible errors.

2) Reconstruction Heuristics: Reconstructing the email conversations is not a straightforward process with the given data. The emails of the corpus unfortunately do not contain the two header fields called *references* and *in-reply-to* of the *Internet Message Standard*, which should contain unique message identifiers of previous messages.

Therefore, we used the email header information about the subject, sender, recipients and the timestamp of the message, as well as the message text itself to approximately reconstruct the conversations. Specifically, we inserted a link in a conversation graph – indicating that a message a is followed by a message b – whenever the following conditions were fulfilled.

(i) One of the recipients of a is the sender of b, i.e., the sender of b replied to a or forwarded a.

(ii) The subject lines of message a and b match (after removing prefixes like “Re:”) **or** the message text of b contains the entire text of a.

(iii) The timestamp of b is later than the one of a **and** there is no other email c with a timestamp that lies between a and b and for which conditions 1 and 2 are fulfilled.

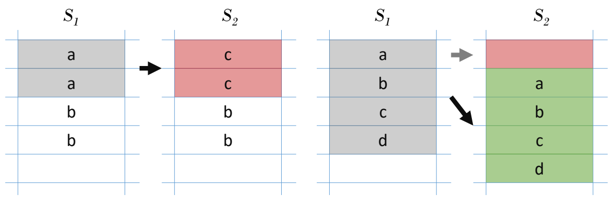
Checking these conditions again requires some heuristics-based approximations due to the noisiness of the data. The sender and recipient names, for example, are often set by the email client based on an integrated address book and do not contain email addresses but real names with no consistent ordering of first and last names. Therefore, we implemented a name matching technique that tries different orderings and uses the Jaro-Winkler distance to assess the similarity of different entries. We assumed the names to be identical if a certain threshold was surpassed.

B. Analyzing the Differences in Spreadsheets

Once we have determined a subset of spreadsheets that are presumably related, e.g., because they are in the same conversation graph or because they have similar names, our tool supports the user with an automated analysis of the differences between the files.

1) Detecting Modifications: Our analysis of differences focuses on changes in formulas. Changes only in number and text constants between two versions are not considered. We consider formula updates, insertions and deletions as changes between spreadsheet versions.

As mentioned above, spreadsheet versions that only have a limited number of differences are particularly relevant for us as



(a) Formulas were changed. (b) Formulas were moved.

Fig. 2. Analyzing differences of a spreadsheet.

it makes it easier to understand the modifications. A commonly used functionality in spreadsheet systems is to copy formulas to apply the same calculations on different rows or columns. In the Enron corpus a spreadsheet with formulas on average contains 2,223 formulas of which only 100 are unique [11]. If a bug fix concerns such a copied formula, we would therefore detect multiple formula changes.

In our calculation scheme for differences we account for such situations where so-called "copy-equivalent" formulas are changed. We achieve this through the use of the R1C1 notation in which copy-equivalent formulas have the same cell content. Figure 2a shows an example where in two copy-equivalent cells the formula was changed from a to c. According to our heuristic, this would only count as one difference.

2) Detecting Moved Cells: Another situation in which a naive approach to spot differences would lead to too many suspected changes is when new rows or columns are inserted as part of a change. Figure 2b shows such a situation where an empty row was inserted. The goal of the subsequently described heuristic is to detect when (blocks of) cells are moved. In the example in Figure 2b, our method should therefore report "no change" instead of a formula deletion in the topmost cell and a formula addition at the bottom.

To detect such movements we use heuristics regarding the surrounding of the changed cells. If we find the formula of the changed cell and an identical surrounding area of a specified size at a different location in the changed spreadsheet, we assume that the whole area was moved to this location.

Algorithm 1 sketches the idea of our corresponding spreadsheet difference analysis. The algorithm takes two spreadsheets S_1 and S_2 to be compared as input and maintains a list called *diffs* in which the found differences are stored. The main function examines all cells which contain a formula in at least one of the spreadsheets. For these cells, the function ISDIFFERENT is called, which checks if the content of the cell differs in the two spreadsheets. Internally, this method also checks if the same difference was already observed before for a copy-equivalent cell as we only want to count each difference once. In case a difference was found, i.e., one of the cells contains no formula or the formulas differ, the function WASMOVED is called, which returns true if we assume that a formula and its surroundings were moved. If the observed difference is not the result of a move, the cell c is stored as a difference in the set *diffs*.

The function WASMOVED checks if the formula in the given

Algorithm 1: FINDDIFFERENCES

```
Input: Two spreadsheets $S_1$, $S_2$; A minimum area size
       $minSize$ to recognize moved areas
Output: A set of cell positions $diffs$ for which
        differences were found between $S_1$ and $S_2$
1 foreach $c \in$ FORMULACELLS($S_1$)
    $\cup$ FORMULACELLS($S_2$) do
2     if ISDIFFERENT($c, S_1, S_2, diffs$) $\wedge$
        $\neg$WASMOVED($c, S_1, S_2, minSize$) then
3         $diffs \leftarrow diffs \cup \{c\}$;
4 return $diffs$;
  function WASMOVED($c, S_1, S_2, minSize$)
5     $candidates \leftarrow$ FINDSAMEFORMULAS($c, S_1, S_2$);
6     foreach $candidate \in candidates$ do
7         $areas \leftarrow areas \cup \{$FINDEQUIVALENTAREA($c$,
              $S_1, candidate, S_2$)$\}$;
8     return $minSize <$ MAXSIZE($areas$);
```

cell with the same surrounding area can be found elsewhere in the spreadsheet. The function first searches for all cells in S_2 that have the same formula as cell c in S_1. Then it iterates over all elements of this list called *candidates* and calculates the size of the area in S_2 that is equal to the area surrounding c in S_1. If a sufficiently large identical block – as specified by the *minSize* parameter – is found for at least one of the *candidates*, the algorithm assumes that the corresponding area was moved.

More complex heuristics or even exact pattern matching methods could of course be used but can come at the cost of higher computational complexity. We chose a simple heuristic as our goal is to support the parameterizable "on-demand" calculation of differences, e.g., in the context of email conversation graphs.

III. VALIDATION – DETECTING ERRORS IN THE CORPUS

To validate our general approach and the designed heuristics, we used the developed software tool to locate an initial set of real-world errors in the Enron corpus.

Our method supports two modes of operation to find spreadsheet versions: (a) based on the inspection of email conversations, (b) based on the similarity of file names.

A. Classifying Changes as Error Corrections

Determining whether a change from one spreadsheet version to another led to the correction or introduction of an error can in most cases only be done through a manual process[2]. Each identified error that we report here was therefore classified as such by at least two independent spreadsheet experts in a manual process. We adopted a conservative strategy and classified changes only as errors if the intended semantics of

[2]In our view, only very simple cases like the removal of a #DIV/0 error can probably be automatically detected with some confidence.

the calculations in the spreadsheet were understandable and the bug was obvious or even mentioned in the email text.

1) Example 1: We searched for email conversations that contained the words "error" and "spreadsheet" in the message text.[3] One filtered email contained the text "*Ron pointed out an error to me in my spreadsheet. The revised one is attached*". The sender pointed out that one calculation outcome was "*too low*". An automated comparison of the attached spreadsheet with other versions of it quickly led us to the change. In cell D6, the formula "=D4*1500" was changed by the sender of the email to "=D10*1500", i.e., a cell reference error was made in the original file, which led to the faulty (too low) outcome.

In that particular case the file names of the different versions of the spreadsheet attached to the emails were identical. This file and its different versions would therefore also be found by our tool when we only look for file versions without reconstructing the email conversations. The text of the email message however assures us that the change was actually an error and not a change of the business rules.

2) Example 2: When searching for files with similar names, our tool returned two versions of a multi-worksheet spreadsheet named CrackSpreadOptions.xls. The files contained six formula differences, which were however detected as changes to copy-equivalent formulas and counted as one. Specifically, the formulas in column M were changed from "=HEAT($B9;...;M$7)" to "=C9*HEAT($B9;...;M$7)" etc., i.e., the computation was extended with a multiplication factor that was forgotten in the previous version.[4] We were confident that this was truly a hard-to-detect omission error [14] because the updated spreadsheet also contained the comment "*Had to scale column M by the gas price!!!*".

B. An Initial Corpus of Errors in the Enron Corpus

So far, we have only conducted a few first sessions to build a corpus of spreadsheet errors with the help of our tool. We have inspected a few dozen of the email conversations with the above mentioned keywords manually to locate obvious errors as those reported above. Furthermore, we made a search based on identical filenames and limited the search to files which differed from each other in at most three formulas. From the returned spreadsheets we inspected about 200 files manually.

Overall, already through our initial search we could identify 28 occurrences which we classified as quantitative errors with high confidence. According to the classification of [15], we found 14 *mechanical* errors, 9 *logical* errors, and 5 *omission* errors. In addition to these errors, we found 8 *qualitative errors* [15], i.e., errors which do not directly lead to immediate failures but degrade the quality of the spreadsheet. Such qualitative errors for example include wrong labels for formulas. We are continuing to extend the corpus and provide all details on a public web site [13]. Our results so far confirm that all error types mentioned in the literature actually appear in real-world spreadsheets.

In the current corpus the majority of the problems was identified based on matching file names as this was the first technique that we explored. More than half of the errors could however have been found using either of our identification techniques (name-based or conversation-based). Specifically, for 19 of the 36 errors the email conversations included information about a corrected error or even its exact location.

IV. Related Work

Besides the Enron document corpus [11] used in this work, other collections of spreadsheets were published over the years to support error research for spreadsheets. Both the often-used EUSES corpus [8] (4,498 documents) and the more recent FUSE corpus [9] (249,376 documents) contain spreadsheets that were retrieved with the help of search engines. Many of the documents, however, contain no formulas at all. Furthermore, no additional information is available about potential errors in the spreadsheets or if they were in practical use.

Other spreadsheet collections were designed to include information about errors. The Hawaii Kooker Corpus for example comprises 75 spreadsheets (with 97 faults) that were created by undergraduate students [16]. A comparable corpus of spreadsheet documents created by students was presented in [17]. While these corpora obviously contain real errors made by humans, it is not fully clear if the spreadsheets and example calculations are representative for spreadsheets that are found in industry. Furthermore, spreadsheets that are created in exercises can be structurally quite diverse, hard to comprehend, or incomplete. Comparing a submitted solution with a reference solution can therefore be tedious.

Using email conversations as an additional source to detect errors in real-world spreadsheets has to our knowledge not been done before. Some works, however, exist that aim at automatically detecting differences in spreadsheets. *SheetDiff* [18], for example, uses a greedy technique to search for several types of differences which are then visually presented to the user. Later on, an approach called *RowColAlign* was proposed that uses a dynamic programming technique to address some shortcomings of *SheetDiff* [19]. In the current version of our tool the differences between spreadsheets are presented in a structured and compact text-based form. We see the integration of the ideas proposed in [18] or [19] to visualize the differences as a promising direction for our future work.

V. Conclusion

Research on error detection techniques for spreadsheets requires a solid understanding of the types of errors that users make when creating spreadsheets. In this work we have presented a method and tool to locate errors in the Enron spreadsheet corpus based on the identification of versions of the same spreadsheet. One particular novelty of our approach lies in the utilization of information from the email conversations in the company. Through a first manual inspection of a number of version candidates with our tool, we could develop an initial set of real-world spreadsheet errors which we plan to continuously extend in the future.

[3]The search with the two terms returned quite a number of irrelevant conversations as the word "error" was often part of email disclaimers.

[4]The function HEAT is part of an external library.

Acknowledgment

The work was funded by the Austrian Science Fund (FWF, contract I2144) and the German Research Foundation (DFG, contract JA 2095/4-1). Thanks to Tom-Philipp Seifert for implementation works.

References

[1] R. R. Panko and D. N. Port, "End User Computing: The Dark Matter (and Dark Energy) of Corporate IT," in *Proceedings of the 45th Hawaii International Conference on System Sciences (HICSS 2012)*, Wailea, HI, USA, 2012, pp. 4603–4612.

[2] EuSpRIG, "Spreadsheet horror stories," Published online at http://www.eusprig.org/horror-stories.htm, Last accessed 2016.

[3] D. Jannach, T. Schmitz, B. Hofer, and F. Wotawa, "Avoiding, finding and fixing spreadsheet errors - a survey of automated approaches for spreadsheet QA," *Journal of Systems and Software*, vol. 94, pp. 129–150, 2014.

[4] R. R. Panko, "What We Know About Spreadsheet Errors," *Journal of End User Computing*, vol. 10, no. 2, pp. 15–21, 1998.

[5] F. Hermans, M. Pinzger, and A. van Deursen, "Supporting Professional Spreadsheet Users by Generating Leveled Dataflow Diagrams," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*, 2011, pp. 451–460.

[6] R. Abraham and M. Erwig, "AutoTest: A Tool for Automatic Test Case Generation in Spreadsheets," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2006)*, 2006, pp. 43–50.

[7] D. Jannach and T. Schmitz, "Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach," *Automated Software Engineering*, vol. 23, no. 1, pp. 105–144, 2016.

[8] M. Fisher and G. Rothermel, "The EUSES Spreadsheet Corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms," *SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[9] T. Barik, K. Lubick, J. Smith, J. Slankas, and E. Murphy-Hill, "FUSE: A Reproducible, Extendable, Internet-scale Corpus of Spreadsheets," in *Proceedings of the 12th Working Conference on Mining Software Repositories, Data Challenge*, 2015.

[10] R. Abraham and M. Erwig, "Mutation Operators for Spreadsheets," *IEEE Transactions on Software Engineering*, vol. 35, no. 1, pp. 94–108, 2009.

[11] F. Hermans and E. R. Murphy-Hill, "Enron's Spreadsheets and Related Emails: A Dataset and Analysis," in *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, Florence, Italy, 2015, pp. 7–16.

[12] T. Schmitz and D. Jannach, "Enron Spreadsheet Error Finder," Published online at http://ls13-www.cs.tu-dortmund.de/homepage/spreadsheets/enron-spreadsheet-tool.shtml, last accessed 2016.

[13] ——, "The Enron Errors Corpus," Published online at http://ls13-www.cs.tu-dortmund.de/homepage/spreadsheets/enron-errors.htm, last accessed 2016.

[14] R. R. Panko and R. P. Halverson, "Are two heads better than one? (at reducing spreadsheet errors in spreadsheet modeling?)," *Office Systems Research Journal*, vol. 15, no. 1, pp. 21–32, 1997.

[15] ——, "Spreadsheets on Trial: A Survey of Research on Spreadsheet Risks," in *Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS 1996)*, Wailea, HI, USA, 1996, pp. 326–335.

[16] S. Aurigemma and R. R. Panko, "The Detection of Human Spreadsheet Errors by Humans versus Inspection (Auditing) Software," in *Proceedings of EuSpRIG 2010 Conference*, London, United Kingdom, 2010.

[17] E. Getzner, "Improvements for Spectrum-based Fault Localization in Spreadsheets," Master's thesis, Graz University of Technology, http://spreadsheets.ist.tugraz.at/index.php/corpora-for-benchmarking/info1/, 2015.

[18] C. Chambers, M. Erwig, and M. Luckey, "SheetDiff: A Tool for Identifying Changes in Spreadsheets," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2010)*, Madrid, Spain, 2010, pp. 85–92.

[19] A. Harutyunyan, G. Borradaile, C. Chambers, and C. Scaffidi, "Planted-model evaluation of algorithms for identifying differences between spreadsheets," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2012)*, Innsbruck, Austria, 2012, pp. 7–14.