Improving dependency-based models for fault localization in spreadsheets

Birgit Hofer and Franz Wotawa
1Graz University of Technology, Graz, Austria
e-mail: {bhofer, wotawa}@ist.tugraz.at

Abstract
Locating faults in spreadsheets can be difficult. Therefore, tools supporting the localization of faults are needed. Model-based software debugging (MBSD) is a promising fault localization technique. This paper presents a novel dependency-based model that can be used in MBSD. This model allows improvements of the diagnostic accuracy while keeping the computation time short. In an empirical evaluation, we show that dependency-based models of spreadsheets whose value-based models cannot be solved in an acceptable amount of time can be solved in less than one second. Furthermore, the amount of diagnoses obtained through dependency-based models is reduced by 15\% on average when using the novel model instead of the original dependency-based model. The short computation time and the improved diagnostic accuracy enable the usage of model-based debugging for spreadsheets in practice.

1 Introduction
Once a program does not behave like expected the process of fault localization and afterwards fault correction starts, which is often difficult and time consuming, independent of the used programming paradigm. Fault identification and correction is even more complicated in cases where the program under consideration is not very well known. This happens for example in software maintenance where programs have to be changed that were developed years ago. Therefore, automated support for localizing and correcting faults is needed.

In particular, Model-based Software Debugging (MBSD) is an interesting automated fault localization technique. MBSD is flexible because different types of models can be used. The chosen model influences the quality of the obtained results and the computation time. Models range from logical sentences representing different kinds of dependencies (like data or control dependencies) to constraint systems which represent the program’s original semantics. All kinds of models used can be automatically obtained from a faulty program. Manual intervention is not needed during the model construction. Since the model is constructed from a faulty program, the model also contains the fault.

The quality of the obtained diagnostic results depends on the underlying model. The closer the model to the original program behavior the better the results. Unfortunately, precise models are computationally demanding. Thus, their application is restricted to small programs. Abstract models (e.g., dependency-based models) compute more diagnoses, but they are computationally less demanding. Therefore, dependency-based models should be improved. In this paper, we show that small changes in the model improve the diagnostic accuracy while keeping the computational complexity small.

In this paper, we rely on spreadsheets, but the presented ideas can be adapted to functional and procedural languages. Spreadsheets are programs written in a particular programming environment, e.g., Microsoft Excel, iWork’s Numbers, and OpenOffice’s Calc. Spreadsheet users outnumber professional programmers many times over: the US Bureau of Labor and Statistics estimates that more than 55 million people used spreadsheets and databases at work on a daily basis in the year 2012 [1]. Spreadsheets are used in companies for example for financial reporting and forecasting. Considering that important decisions are often based on spreadsheets, it is desirable that spreadsheets are free from errors. Unfortunately, numerous studies have shown that existing spreadsheets contain errors at an alarmingly high rate [2]. Localizing the cell(s) which explain an observed error can be very demanding because spreadsheets lack support for abstraction, encapsulation, or structured programming. Furthermore, spreadsheets are often created without planning ahead of time for maintainability or scalability. Therefore, approaches supporting fault localization are needed.

Some software fault localization techniques have been adapted to spreadsheets, e.g., [3; 4; 5; 6]. MBSD is one of these techniques. So far, researchers have only focused on methods that use value-based models [7; 8]. Value-based models compute a small set of possible explanations (i.e., diagnoses) for an observed misbehavior. Unfortunately, value-based models have high computation times and they do not scale: the underlying solving mechanisms have problems when dealing with variables with large domains. In addition, spreadsheets containing Real numbers could not be handled satisfactorily. In an empirical evaluation, Außerlechner et al. [9] showed the limitations of different constraint solvers and SMT (satisfiability modulo theories) solvers when using these models.

To the best of our knowledge, dependency-based models [10] have not been used for localizing faults in spreadsheets. The reason for this may be their inaccuracy. Dependency-based models compute a significantly higher number of diagnoses than value-based models. In this pa-
per, we propose a novel type of dependency-based model which improves on the diagnostic accuracy while keeping the time for computing the diagnoses short.

In order to demonstrate the differences between the value-based, the original dependency-based, and our improved dependency-based model, we make use of a running example. This example is a simplified version of the “homework/budgetone” spreadsheet taken from the EUSES spreadsheet corpus [11]. We manually inserted a fault into this spreadsheet. Figure 1(a) shows a faulty variant of the spreadsheet. Figure 1(b) shows the formula view of this spreadsheet. The output cells1 of the spreadsheet are shaded in gray. The faulty cell D5 is framed. The fault manifests in the value of the output cell D7. The expected value for this cell is 78.6 %.

![Figure 1: Running example](image)

When using any of the discussed models, the faulty cell can be detected. The value-based model and our novel dependency-based model identify three cells that could explain the observed misbehavior, while the original dependency-based model identifies six cells as possible explanations. When using the constraint solver MINION [12], both dependency-based models require only one third of the computation time compared to the value-based model.

One might think that capturing the dependencies of cells is easy, but the tracking of dependencies can be demanding even for small spreadsheets. Figure 2 demonstrates the dependency graph for the running example. The input cells are shaded in dark gray, the output cells are shaded in light gray. Even for this small example, there is some time and attention required to track and understand all dependencies.

The remainder of this paper is structured as follows: In Section 2, we discuss related research. In Section 3, we introduce the preliminaries and the different types of models. In addition, we show how the models look like for the running example. The empirical evaluation (Section 4) shows that the novel dependency-based model reduces the number of diagnoses by 15 % compared to the original dependency-based model. For one third of the spreadsheets, the value-based model could not be solved within a limit of 20 minutes. The dependency-based models for the same spreadsheets were solved within a fraction of one second. In Section 5, we discuss the advantages of dependency-based models. Finally, we conclude the paper in Section 6.

1 An output cell is a cell that is not referenced by any other cell.

Figure 2: Dependency graph for the spreadsheet from Fig. 1

2 Related work

The research area of automated debugging starts with Weiser’s [13; 14] and Shapiro’s [15] seminal works. Shapiro presented an algorithmic debugging approach. Later Console et al. [16] and Bond [17] used model-based diagnosis [18] to improve debugging by automatically eliminating parts of the source code that is irrelevant for an observed error. Console et al. [16] started the research area of MBSD. Besides the application of MBSD to logic programming languages [16] there are examples of MBSD for hardware design languages [19; 20] and functional languages [21].


Mateis et al. [26; 27] applied MBSD to an object-oriented language. In their work, they describe the conversion of Java programs into simple dependency-based models. Wotawa et al. [28; 29] present an MBSD approach which relies on a value-based model. They show how to formulate a debugging problem as a constraint satisfaction problem (CSP). Wotawa et al. [30] discuss the computational costs of MBSD. Therefore, they represent the created CSPs as hypertrees. The width of a hypertree is a good indicator for the complexity of the CSP. In addition, they prove that there exists no constant upper-bound for the hypertree width of arbitrary programs. The upper-bound of the hypertree width for a particular program is given by the number of statements in the program. Problems with a hypertree width > 5 are said to be hard problems. The authors showed that the hypertree width is often greater than 5 even for small programs. Therefore, they conclude that debugging is a hard problem.

Jannach and Engler [6] used constraint solving for spreadsheet debugging. Jannach and Schmitz [8] developed the EXQUISITE debugging tool for MS Excel. This debugging tool implements a model-based debugging approach that is based on user-defined test cases and hitting sets. Abreu et al. [7] proposed a model-based debugging approach for spreadsheets. This approach converts a spreadsheet into a value-based constraint satisfaction...
problem (CSP). Their approach differs from Jannach and Schmitz in two aspects: (1) Abreu et al. rely only on a single test case. (2) Instead of using a hitting set approach, they directly encode the reasoning about the correctness of cells into the CSP.

3 Model-based Software Debugging

MBSD originates from model-based diagnosis (MBD) where the objective is to formalize a theory of diagnosis that allows for localizing faulty components in a system based on observations. According to Reiter [18], a diagnosis problem comprises a system with its corresponding model SD (system description), a set of components COMP and a set of observations OBS. The system description SD comprises a formal description of the structure of the system and the behavior of its components. The behavior of a component \( C \in COMP \) is given in the form \( \neg AB(C) \rightarrow Behav(C) \) where AB stands for abnormal: either a component is abnormal \((AB(C))\) or C behaves as expected. The real behavior of C has to be stated in \( Behav(C) \).

Diagnoses can be made if the model contradicts the observations when assuming that all components in COMP are correct, i.e., for all \( C \in COMP : \neg AB(C) \) holds. The idea behind MBD is to find settings for the AB predicates that remove the contradiction. Formally, a diagnosis is defined as follows: \( \Delta \subseteq COMP \) is a diagnosis, if and only if \( SD \cup OBS \cup \{ \neg AB(C)[C \in COMP \backslash \Delta] \} \cup \{ AB(C)[C \in \Delta] \} \) is satisfiable.

Reiter assumed that SD and OBS comprise logical sentences. He also specified an algorithm for computing all minimal diagnoses, i.e., diagnoses where no element can be eliminated furthermore. Constraint representations of models and observations as well as algorithms for diagnosis based on constraint solving have been introduced. The algorithms rely on the idea to use a constraint solver for setting values of the AB predicates. In this paper, we follow the idea of converting the debugging problem into a constraint satisfaction problem (CSP). A CSP \([31]\) is a tuple \((V, D, C)\) where \( V \) is a set of variables with a corresponding domain from \( D \), and \( C \) is a set of constraints. Each constraint has a set of variables, i.e., its scope, and specifies the relation between the variables. A solution to a CSP is an assignment of values to variables such that all constraints are fulfilled. For more information regarding constraint solving in case of debugging we refer the interested reader to Wotawa et al. [29]. In their paper, the mapping of programs into CSPs and an algorithm for debugging are given.

In this paper, relying on spreadsheet programs, the spreadsheet’s cells and the given observations (i.e., the test case\(^2\)) are converted into constraints. As the given test case is a failing test case, this constraint system results in a contradiction. In order to determine which cells could resolve this contradiction, we use the ideas of MBD and introduce abnormal variables (AB) for all cells. These abnormal variables represent the “health” state of the cells. If a cell \( c \) is not abnormal, the formula of the cell must be correct: \( \neg AB(c) \rightarrow \text{constraint}(c) \). This logic expression can be transformed to \( AB(c) \lor \text{constraint}(c) \). Having such a constraint system, we are able to use a constraint or SMT solver to determine which abnormal variables have to be set to true to eliminate the contradiction.

We discuss three different types of models in the following subsections. These models can be automatically generated from a faulty spreadsheet. Since the models are derived from a faulty spreadsheet, they also contain the fault(s).

3.1 Value-based models

When using value-based models, the values of the cells are propagated. A value-based constraint system contains (i) the input cells and their values, (ii) the output cells and their expected values, and (iii) all formulas concatenated with their abnormal variable. The constraint representation handles the formulas as equations instead of assignments. This allows to draw conclusions on the input from the output of a formula. Such a value-based model for spreadsheets is proposed by Abreu et al. [7]. The running example from Figure 1(b) is converted into the following constraints:

<table>
<thead>
<tr>
<th>Input:</th>
<th>Output:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B2 ) == 1000</td>
<td>( D3 ) == 20.6</td>
</tr>
<tr>
<td>( C2 ) == 1500</td>
<td>( B7 ) == 0.75</td>
</tr>
<tr>
<td>( B3 ) == 20</td>
<td>( C7 ) == 0.81</td>
</tr>
<tr>
<td>( C3 ) == 21</td>
<td>( D7 ) == 0.786</td>
</tr>
<tr>
<td>( B5 ) == 5000</td>
<td></td>
</tr>
<tr>
<td>( C5 ) == 6000</td>
<td></td>
</tr>
</tbody>
</table>

**Formula constraints:**

\[
AB(\text{cell}_{D2}) \lor D2 == B2 + C2 \\
AB(\text{cell}_{D3}) \lor D3 == D4 / D2 \\
AB(\text{cell}_{B4}) \lor B4 == B3 \times B2 \\
AB(\text{cell}_{C4}) \lor C4 == C3 \times C2 \\
AB(\text{cell}_{D4}) \lor D4 == B4 + C4 \\
AB(\text{cell}_{D5}) \lor D5 == B5 \\
AB(\text{cell}_{B6}) \lor B6 == B4 - B5 \\
AB(\text{cell}_{C6}) \lor C6 == C4 - C5 \\
AB(\text{cell}_{D6}) \lor D6 == D4 - D5 \\
AB(\text{cell}_{B7}) \lor B7 == B6 / B4 \\
AB(\text{cell}_{C7}) \lor C7 == C6 / C4 \\
AB(\text{cell}_{D7}) \lor D7 == D6 / D4
\]

Solving this constraint system leads to three possible solutions: Either cell D5, D6 or D7 must contain the fault.

3.2 Original dependency-based models

Dependency-based models have already been used for debugging software written in traditional programming languages [10]. However, to the best of our knowledge, they have not been used so far for debugging spreadsheets.

When using dependency-based models, only the information about whether the computed values are correct is propagated. For example, if we have the formula \( B2 + C2 \) stored in a cell \( D2 \), then the value of \( D2 \) can only be correctly computed if (1) the values of \( B2 \) and \( C2 \) are correct, and (2) the equation is also correct.

We present the correctness of values for cells as Booleans instead of Integer or Real values. All variables representing input cells are initialized with true. All variables repre-
senting correct output cells\(^1\) are also initialized with true. The variables representing erroneous output cells are initialized with false.

Instead of using the concrete formulas in the constraints, only the correctness relation is modeled. If the formula of cell \(c\) is correct and the input values of a formula are correct then cell \(c\) must compute a correct value:

\[
AB(\text{cell } c) \lor \bigwedge_{c' \in \rho(c)} c' \rightarrow c
\]

(1)

where \(\rho(c)\) is the set of all cells that are referenced in \(c\). Details about this modeling for software written in an imperative language can be found in [10]. The dependency-based constraints for our running example are as follows:

<table>
<thead>
<tr>
<th>Input:</th>
<th>Output:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(B2) == true</td>
<td>(D3) == true</td>
</tr>
<tr>
<td>(C2) == true</td>
<td>(B7) == true</td>
</tr>
<tr>
<td>(B3) == true</td>
<td>(C7) == false</td>
</tr>
<tr>
<td>(C3) == true</td>
<td>(D7) == false</td>
</tr>
<tr>
<td>(B5) == true</td>
<td>(C5) == true</td>
</tr>
<tr>
<td>(C5) == true</td>
<td>(D5) == false</td>
</tr>
</tbody>
</table>

**Formula constraints:**

\[
AB(\text{cell } D2) \lor (B2 \land C2 \rightarrow D2)
\]

\[
AB(\text{cell } D3) \lor (B2 \land D4 \rightarrow D3)
\]

\[
AB(\text{cell } B4) \lor (B2 \land B3 \rightarrow B4)
\]

\[
AB(\text{cell } C4) \lor (C2 \land C3 \rightarrow C4)
\]

\[
AB(\text{cell } D4) \lor (B4 \land C4 \rightarrow D4)
\]

\[
AB(\text{cell } D5) \lor (B5 \rightarrow D5)
\]

\[
AB(\text{cell } B6) \lor (B4 \land B5 \rightarrow B6)
\]

\[
AB(\text{cell } C6) \lor (C4 \land C5 \rightarrow C6)
\]

\[
AB(\text{cell } D6) \lor (D4 \land D5 \rightarrow D6)
\]

\[
AB(\text{cell } B7) \lor (B4 \land B6 \rightarrow B7)
\]

\[
AB(\text{cell } C7) \lor (C4 \land C6 \rightarrow C7)
\]

\[
AB(\text{cell } D7) \lor (D4 \land D6 \rightarrow D7)
\]

Solving this constraint system leads to six possible solutions: Either cell \(B4\), \(C4\), \(D4\), \(D5\), \(D6\) or \(D7\) must contain the fault. This dependency-based model computes more diagnoses because of the implication. In the value-based model, the cells \(B4\), \(C4\), and \(D4\) can be excluded from the set of possible diagnoses because \(B4\) and \(C4\) are used to compute \(D4\), and \(D4\) is used to compute \(D3\), which is known to compute the correct result. Unfortunately, this information gets lost when using the implication because the implication allows conclusions only from the input to the output but not vice versa. This issue will be solved with the novel dependency-based model that is explained in the following subsection.

**3.3 Novel dependency-based models**

The novel dependency-based model uses the bi-implication (equivalence) instead of the implication in order to eliminate the previously described weakness. The rationale here is that if a cell value is correct also the contributing parts have to be correct. The formula constraints for our running example from Figure 1(b) are as follows:

\[
AB(\text{cell } D2) \lor (B2 \land C2 \leftrightarrow D2)
\]

\[
AB(\text{cell } D3) \lor (D2 \land D4 \leftrightarrow D3)
\]

\[
AB(\text{cell } B4) \lor (B2 \land B3 \leftrightarrow B4)
\]

\[
AB(\text{cell } C4) \lor (C2 \land C3 \leftrightarrow C4)
\]

\[
AB(\text{cell } D4) \lor (B4 \land C4 \leftrightarrow D4)
\]

\[
AB(\text{cell } D5) \lor (B5 \leftrightarrow D5)
\]

\[
AB(\text{cell } B6) \lor (B4 \land B5 \leftrightarrow B6)
\]

\[
AB(\text{cell } C6) \lor (C4 \land C5 \leftrightarrow C6)
\]

\[
AB(\text{cell } D6) \lor (D4 \land D5 \leftrightarrow D6)
\]

\[
AB(\text{cell } B7) \lor (B4 \land B6 \leftrightarrow B7)
\]

\[
AB(\text{cell } C7) \lor (C4 \land C6 \leftrightarrow C7)
\]

\[
AB(\text{cell } D7) \lor (D4 \land D6 \leftrightarrow D7)
\]

Solving this constraint system leads to the same three diagnoses as when using a value-based model.

The underlying rationale that leads to the use of bi-implications instead of ordinary implications is unfortunately not always correct. The bi-implication cannot be used in case of coincidental correctness [32]. Coincidental correctness has to do with fault masking where an output value is correct but not all of its input values are. For example, consider the situation of a conditional statement \(IF(A1, A2, A3)\) where \(A2\) is returned if \(A1\) is true, and \(A3\) otherwise. In this case it might happen that \(A1\) is true, \(A2\) is returned but \(A3\) is not correctly computed. Hence, when using a bi-implication we would misleadingly assume that \(A3\) is correct too. In order to handle coincidental correctness, we have analyzed the cases where coincidental correctness happen and treated them differently. Coincidental correctness might occur for example in the following situations:

- conditional functions (e.g., the IF-function),
- abstraction functions (e.g., MIN, MAX, COUNT),
- Boolean,
- multiplication with a 0-value, and
- power with 0 or 1 as base number or 0 as exponent.

Please note, that this list gives only examples. It is not a complete list, because the size of the list depends on the functions that are supported by the concrete spreadsheet environment (e.g. Microsoft Excel, iWorks’ Number, OpenOffice’s Calc). All formulas where coincidental correctness might happen still have to be modeled with the implication instead of the bi-implication.

**4 Empirical evaluation**

This section consists of two major parts: the empirical setup (discussing the prototype implementation, the used platform, and the evaluated spreadsheet corpora) and the results showing that dependency-based models are able to compute diagnoses within a fraction of one second even for spreadsheets whose value-based models require more than 20 minutes of solving time. In addition, this empirical evaluation shows that the number of diagnoses obtained by the novel dependency-based model is reduced by 15% on average compared to the original dependency-based model.

\(^1\)We defined output cells as those cells that are not referenced by other cells. In practice, any formula cell whose expected value is known can be taken as output cell. For correct output cells, the computed values and the expected values are equal. For erroneous output cells, the computed values differ from the expected values.
We developed a prototype in Java for performing the empirical evaluation. This prototype uses MINION [12] as constraint solver. MINION is an out-of-the-box, open source constraint solver and offers support for almost all arithmetic, relational, and logic operators such as multiplication, division, and equality over Boolean and Integers. The evaluation was performed on an Intel Core2 Duo processor (2.67 GHz) with 4 GB RAM and Windows 7 as operating system. We used the MINION version 0.15. The computation time is the average time over 10 runs.

We evaluated the models by means of spreadsheets from the publicly available Integer spreadsheet corpus⁴ [9]. This corpus contains 33 different spreadsheets (12 artificially created spreadsheets and 21 real-life spreadsheets), e.g., a spreadsheet that calculates the lowest price combination on a shopping list. These spreadsheets contain both arithmetical and logical operators as well as the functions SUM and IF. Faulty versions of the spreadsheets (containing single, double and triple faults) were created by randomly selecting formulas and applying Abraham and Erwig’s mutation operators [33] on them. Abraham and Erwig have chosen their mutation operators in a way that they reflect faults known from literature, e.g. the replacement of a formula by a constant or the replacement of a range with another range. The corpus contains in total 220 mutants. We refer the interested reader to [9] for more information about this corpus. In the empirical evaluation, we used only the spreadsheets which contain a single fault, i.e. 94 spreadsheets.

Table 1 gives a short summary of the characteristics of the used spreadsheets. The spreadsheets are divided into two sub-groups: spreadsheets whose value-based models are solved by MINION in less than 20 minutes (low-complexity spreadsheets) and spreadsheets whose value-based models could not be solved within 20 minutes (i.e. 31 from 94 spreadsheets, called high-complexity spreadsheets). On average, the spreadsheets contained 32.7 respectively 64.3 formula cells. This table also lists the number of formula cells where the implication can be replaced by the bi-implication. For the low-complexity spreadsheets, the bi-implication can be used for 40 % of the formulas. For the high-complexity spreadsheets, the bi-implication can be used for 53 % of the formulas. The average number of referenced cells and operators per formula indicates that the formulas have a low complexity. For all spreadsheets, the fault manifests only in a single (erroneous) output cell. For all other output cells, the computed value is correct. In practice, there could be more erroneous output cells or the user can also indicate that an intermediate cell is erroneous.

![Table 1: Information about the used spreadsheets](https://dl.dropbox.com/u/38372651/Spreadsheets/Integer_Spreadsheets.zip)

<table>
<thead>
<tr>
<th>Avg. numbers per collection</th>
<th>63 sp.</th>
<th>31 sp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formula cells</td>
<td>32.7</td>
<td>64.3</td>
</tr>
<tr>
<td>Formula cells with equivalence</td>
<td>13.0</td>
<td>34.2</td>
</tr>
<tr>
<td>Formula cells with implication</td>
<td>19.7</td>
<td>30.1</td>
</tr>
<tr>
<td>Referenced cells per formula</td>
<td>3.0</td>
<td>2.9</td>
</tr>
<tr>
<td>Operators per formula</td>
<td>2.4</td>
<td>2.0</td>
</tr>
<tr>
<td>Input cells</td>
<td>23.3</td>
<td>29.5</td>
</tr>
<tr>
<td>Erroneous output cells</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Correct output cells</td>
<td>3.0</td>
<td>5.2</td>
</tr>
</tbody>
</table>

Table 2 compares the three types of models with respect to fault localization capabilities and runtimes. The fault localization capabilities are expressed by means of the number of cells that are single fault diagnoses. The runtime is measured by means of MINION’s average solving time (over 10 runs). For the 31 spreadsheets whose value-based models could not be solved within 20 minutes, the dependency-based models are solved in less than one second. These runtime results indicate that dependency-based models are better suited for debugging large spreadsheets than value-based models. Considering the diagnostic accuracy, the value-based model yields better results. It computes only one third of the diagnoses of the original dependency-based model. Table 2 in addition shows the average achieved REDUCTION:

\[
\text{REDUCTION} = 1 - \frac{\text{Diagnoses in model}}{\text{Formula cells}}
\]

<table>
<thead>
<tr>
<th>Model</th>
<th>Single fault diagnoses</th>
<th>Reduction in %</th>
<th>Solving time (in ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBM</td>
<td>4.0</td>
<td>53.5</td>
<td>56818.8</td>
</tr>
<tr>
<td>DBM</td>
<td>13.2</td>
<td>40.0</td>
<td>32.0</td>
</tr>
<tr>
<td>NDM</td>
<td>11.0</td>
<td>46.0</td>
<td>31.6</td>
</tr>
</tbody>
</table>

Table 2: Evaluation results for the value-based model (VBM), the original dependency-based model (DBM) and the novel dependency-based model (NDM)

The improved dependency-based model decreases the number of computed diagnoses by 15 % compared to the original dependency-based model. Table 3 gives an overview of the achieved IMPROVEMENT:

\[
\text{IMPROVEMENT} = 1 - \frac{\text{Diagnoses in novel model}}{\text{Diagnoses in original model}}
\]

For 64 spreadsheets, no reduction in the number of diagnoses was achieved when using the novel dependency-based model instead of the original model. However, for 15 spreadsheets, a reduction of more than 80 % was achieved.

<table>
<thead>
<tr>
<th>IMPROVEMENT</th>
<th>Number of spreadsheets</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 %</td>
<td>64</td>
</tr>
<tr>
<td>[0 %;10 %]</td>
<td>0</td>
</tr>
<tr>
<td>[10 %;20 %]</td>
<td>1</td>
</tr>
<tr>
<td>[20 %;30 %]</td>
<td>1</td>
</tr>
<tr>
<td>[30 %;40 %]</td>
<td>2</td>
</tr>
<tr>
<td>[40 %;50 %]</td>
<td>5</td>
</tr>
<tr>
<td>[50 %;60 %]</td>
<td>0</td>
</tr>
<tr>
<td>[60 %;70 %]</td>
<td>4</td>
</tr>
<tr>
<td>[70 %;80 %]</td>
<td>2</td>
</tr>
<tr>
<td>[80 %;90 %]</td>
<td>7</td>
</tr>
<tr>
<td>[90 %;100 %]</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 3: Summary of the achieved reduction when using the novel dependency-based model instead of the original dependency-based model

⁴https://dl.dropbox.com/u/38372651/Spreadsheets/Integer_Spreadsheets.zip
Both, the value-based and the dependency-based models can be used to debug spreadsheets which contain more than one fault. However, we refrain from comparing the different models for multiple-fault spreadsheets for the following reason: In our evaluation, we only considered minimal diagnoses: a diagnosis is minimal if no proper subset of it is a valid diagnosis. By definition, all supersets of a diagnosis are also diagnoses. When evaluating spreadsheets containing multiple faults we have two options: (1) Either we only count the number of minimal diagnoses, (2) or we also count the number of all supersets of the minimal diagnoses. Our experiments showed that the dependency-based models compute more single fault diagnoses than the value-based model. Option 1 would result in less diagnoses for the dependency-based models than the value-based model, because the value-based model could compute combinations of pairs that are already reported as single fault diagnoses in the dependency-based models. Option 2 would result in a huge amount of diagnoses for all types of models (the value-based model would compute the smallest number of diagnoses, followed by the novel dependency-based model). For both options, the presented data would not be meaningful. Therefore, further research is required in order to build a robust metric for comparing higher order diagnoses for different types of models.

5 Discussion

Dependency-based models could be faster solved than value-based models because the domain is restricted. Instead of using Integer or Real numbers, only Boolean values are used. This reduces the complexity to two possible instantiations per cell: the output value could be either correct or incorrect. When using value-based models, the output values of cells have an infinite number of possible instantiations. Value-propagation and heuristics are some of the techniques used by constraint and SMT solvers to restrict the search space. However, the search space when using Integer and Real numbers is still immense. This explains the runtime results of the empirical evaluation.

The reduction of the domain comes with an additional advantage: Users only have to indicate which output cells have a wrong value instead of indicating the expected values for those cells. Users often do not know the exact expected values. It is easier for them to only determine if a value is incorrect. When using Real numbers, indicating correct expected values is even more difficult because the user has to additionally indicate the required precision. For example, consider that the user indicates 3.24 as expected value. For one scenario, the value 3.2412 could be a correct computed value, but for another scenario, only 3.2400 could be the correct computed value. When using only Boolean variables, this problem could never happen.

An additional advantage of the reduction of the domain is the larger range of solvers that can be used. There are many solvers which are specialized on solving problems which only contain Boolean variables. Since all solvers support at least Boolean variables, an arbitrary solver can be used.

Since the beginnings of dependency-based models (see e.g., [34]) always implications instead of bi-implication has been used. The reason for this is that faults might be masked (coincidental correctness). In case of fault masking, MBSD could not detect the fault when the bi-implication is used instead of an implication. In this paper, we consider this effect and we therefore use the bi-implication only for modeling the behavior of those cells where fault masking could not happen. When fault masking might occur, we still use the implication in order to avoid missing a faulty cell.

Model-based software debugging (MBSD) has its origins in model-based diagnosis (MBD). Therefore, we discuss the relation of dependency- and value-based models in MBSD to structural, weak-fault and strong fault-models in MBD. The dependency-based models are related to the structural model, as both models propagate only correctness information. The value-based model is related to the weak-fault model, because both models compute of the values of the components. In this paper, we did not discuss a model that is equivalent to the strong-fault model. Such a model would also include explanations why a component fails. For the spreadsheet domain, such a strong-fault model would encode what part of a formula has to be changed in order to eliminate the observed misbehavior (e.g. replace a ‘+‘ with a ‘-‘ in the formula of cell A1).

Dependency-based models are related to slicing [10]. Lyle and Weiser proposed to use program dicing [35] for narrowing down the search space when locating faults. In program dicing, the slice of a correct variable (i.e. a variable where the computed value equals the expected value) is subtracted from the slice of an incorrect variable. The programmer focuses only on those statements that are in this difference set. The problem of program dicing is that the slice of the correct variable could also contain the fault. In such a case, the difference set would not contain the faulty statement. When restricting the dicing approach to allow only slices of variables where no fault masking could happen, this problem could be eliminated.

6 Conclusions

Locating faulty formulas in spreadsheets can be time consuming. This paper addresses the fault localization problem by means of model-based diagnosis. Our most important contribution is the introduction of a novel dependency-based model. This model improves previous work in two ways: (1) Compared to the original dependency-based model, it reduces the amount of diagnoses that have to be manually investigated by 15 %. (2) Compared to the value-based model, it reduces the required solving time and allows the computation of diagnoses in real-time where the value-based model cannot compute solutions within 20 minutes. The savings in computation time can be explained by the reduction of the domain: The dependency-based model requires only Boolean variables instead of Integers and Real numbers.

The reduction of the domain comes with additional advantages: (1) An arbitrary solver can be used, because all solvers support at least Boolean variables. (2) Even spreadsheets containing Real numbers can be debugged with any solver when using dependency-based models. (3) The user does not need to indicate concrete values for the erroneous output variables. The information that an output cell computes the wrong value is sufficient.

We are convinced that the model presented improves the state of the art in model-based diagnosis. Further work includes a user study and the adaptation to other types of programs, e.g. programs written in imperative or object-oriented languages.
References


