Combining Models for Improved Fault Localization in Spreadsheets

Birgit Hofer, Andrea Höfler, and Franz Wotawa

***Abstract*—Spreadsheets are the most prominent example of end-user programing, but they unfortunately are often erroneous, and thus, they compute wrong values. Localizing the true cause of such an observed misbehavior can be cumbersome and frustrating especially for large spreadsheets. Therefore, supporting techniques and tools for fault localization are highly required. Model-based software debugging (MBSD) is a well-known technique for fault localization in software written in imperative and object-oriented programing languages like C, C++, and Java. In this paper, we explain how to use MBSD for fault localization in spreadsheets and compare three types of models for MBSD, namely the value-based model (VBM), the dependency based model (DBM), and an improved version of the DBM. Whereas the VBM computes the lowest number of diagnoses, both DBMs convince by their low computational complexity. Hence, a combination of these two types of models is desired, and we present a solution that combines value-based and DBM in this paper. Moreover, we discuss a detailed evaluation of the models and the combined approach, which indicates that the combined approach computes the same number of diagnoses like the VBMs while requiring less computation time. Hence, the proposed approach is more appropriate to be used in tools for fault localization in spreadsheets.**

***Index Terms*—Fault diagnosis, fault location, model-based diagnosis (MBD), reasoning about programs, spreadsheet programs.**

Abbreviations & Acronyms list

AB	Abnormal (usually used in MBSD as predicates in formulas).
CSP	Constraint Satisfaction Problem.
DBM	Dependency Based Model.
HPD	High Priority Diagnoses.
ISSRE	The IEEE International Symposium on Software Reliability Engineering.
LPD	Low Priority Diagnoses.
MBD	Model-Based Diagnosis.
MBSD	Model-Based Software Debugging.
NDM	Novel Dependency based Model.
OBS	Observations, i.e., values of cells in a spreadsheet.
SD	System Description; formal model of a spreadsheet.
SMT	Satisfiability Modulo Theories.
VBM	Value-Based Model.

Manuscript received April 8, 2015; revised October 29, 2016, May 10, 2016, and August 17, 2016; accepted November 16, 2016. Date of publication January 4, 2017; date of current version March 1, 2017. This work was supported by the Austrian Science Fund (FWF) project *DEbugging Of Spreadsheet Programs* under contract number I2144 and the Deutsche Forschungsgemeinschaft (DFG) under contract number JA 2095/4-1. Associate Editor: T. H. Tse.

B. Hofer and F. Wotawa are with the Institute for Software Technology, Graz University Technology, Graz 8010, Austria (e-mail: bhofer@ist.tugraz.at; wotawa@ist.tugraz.at).

A. Höfler is with the Graz University of Technology, Graz 8010, Austria (e-mail: andrea.hoefler@student.tugraz.at).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2016.2632151

I. Introduction

SPREADSHEETS are by far the most successful and prominent example of end-user programing. Spreadsheet users outnumber professional programers many times over. The US Bureau of Labor and Statistics estimates that, in 2012, more than 55 million people used spreadsheets and databases at work on a daily basis [1]. Moreover, spreadsheets are used in companies for example for financial reporting and forecasting.

Given 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 an alarmingly high number of errors [2]. The list of horror stories where spreadsheets caused immense financial loses or damage of image is long[1]. For example, the economists Reinhart and Rogoff suffered from a loss of reputation, when it became public that they published erroneous data because of a spreadsheet fault [3], [4]. A recent example of a financial loss caused by a spreadsheet is the "London whale" trading debacle [5]: A loss of $400 million was not immediately detected because of a fault in JP Morgan's value at risk model spreadsheet. According to Panko [6], there is a 3% to 5% chance of making a mistake when writing a formula; hence, the probability that a spreadsheet with 1000 formulas contains at least one fault is more than 99.9%.

Localizing the cell(s) which are responsible for an observed error can be very demanding because spreadsheets lack support for abstraction, encapsulation, or structured programing. Furthermore, spreadsheets are often created without previous planning of time for maintainability or scalability. Therefore, support for fault localization is strongly needed. There have been studies on how users manually debug spreadsheets, e.g., [68]. In this paper, we focus on automated debugging, in particular model-based fault localization for spreadsheets.

Fig. 1 illustrates our running example. This spreadsheet calculates the moving behavior of an object in three subsequent phases: constant acceleration, constant velocity, and constant deceleration. These phases are represented by the columns B, C, and D, accordingly. When a domain expert examines this

[1]See European Spreadsheet Risks Interest Group (EuSpRiG), http://www.eusprig.org/horror-stories.htm.

	A	B	C	D	E
1		**Constant Acceleration**	**Constant Velocity**	**Constant Deceleration**	**Final State**
2	**Initial Velocity [m/s]**	0,0	20,0	20,0	**0,0**
3	**Acceleration [m/s²]**	2,0	0,0	-4,0	
4	**Duration [s]**	10,0	10000,0	5,0	
5	**Distance [m]**	100,0	0,0	50,0	
6	**Accumlated Distance [m]**	**100,0**	**100,0**	**150,0**	

(a)

	A	B	C	D	E
1		**Constant Acceleration**	**Constant Velocity**	**Constant Deceleration**	**Final State**
2	**Initial Velocity [m/s]**	0	=B2+B3*B4	=C2+C3*C4	=D2+D3*D4
3	**Acceleration [m/s²]**	2	0	-4	
4	**Duration [s]**	10	10000	5	
5	**Distance [m]**	=B2*B4+B3*B4*B4/2	=B2*C4+C3*C4*C4/2	=D2*D4+D3*D4*D4/2	
6	**Accumlated Distance [m]**	**=B5**	**=B5+C5**	**=B5+C5+D5**	

(b)

Fig. 1. Running example. Input cells are shaded in blue. Output vales are given in bold font. Correct output values are given in green color, erroneous output values are given in red color. The faulty cell is framed in red. (a) Value view of the faulty spreadsheet and (b) formula view of the faulty spreadsheet.

spreadsheet, he/she first identifies the cells that are important in the expert's opinion. For example, the expert decides that the cells B6, C6, D6, and E2 are important. Afterward, he/she makes back-of-an-envelope calculations for these cells to get a feeling for the order of magnitude the values of these cells should have. Assume that the domain expert calculates that the value for cell B6 must be around 100, the values for the cells C6 and D6 must be more than 100 000, and the value for E2 must be 0. These back-of-an-envelope calculations help to detect that the values for the cells C6 and D6 must be wrong because the order of magnitude of the computed values differs from that of the estimation (100 versus 100 000). The correct values for C6 and D6 are 200 100 and 200 150, respectively. Please note that the user might have also used spreadsheet testing tools, e.g., WYSIWYT [51], Expector [65], or AUTOTEST [53] to detect these errors.

Why do the computed values for these cells deviate from the expected values (while the values for B6 and E2 are correct)? For this small example, we identify the root cause of the observed misbehavior, i.e., the faulty formula, by investigating all formulas step by step. However, for larger spreadsheets comprising several hundred or even thousands of formulas, identifying the root cause of an observed misbehavior is challenging, difficult, and time consuming.

One possibility to deal with the fault localization problem is model-based software debugging (MBSD) [7], where we make assumptions about the correctness of cells. For example, when assuming that the cell C5 of the running example contains a wrong formula, we ignore this formula and we investigate whether the remaining formulas do not contradict the expected values. This can be automatically done when considering formulas as equations and using a constraint or satisfiability modulo theories (SMT) solver for consistency checking. When ignoring the formula given in C5, a solver would not detect an inconsistency with the expected output values (B6 = 100, C6 = 200,100, D6 = 200,150, and E2 = 0). Therefore, C5 is a root cause explanation for the observed misbehavior; changing its formula to C2 * C4 + C3 * C4 * C4/2 is a repair.

MBSD requires a model of the spreadsheet for the consistency checks. This model contains either concrete or abstract information. In the first case, we speak of a value-based model (VBM); in the second case, we speak of a dependency based model (DBM). Both model types can be automatically generated from a faulty spreadsheet without any user interaction.

In the VBM [8]–[11], the formulas are directly converted into constraints. Concrete values are propagated within the individual constraints of the model. The main advantage of VBMs is their precision: they allow for computing a small set of diagnoses, i.e., explanations for an observed misbehavior. Unfortunately, they have high computation times and they do not scale. In addition, real numbers cannot be handled satisfactorily [12].

In the DBM, only the data and control dependencies are captured. Instead of propagating concrete values, only correctness information is propagated: the computed value of the cell is correct, if the cell is assumed to behave as expected, and all of its input values are also correct. This type of model is well-known for debugging programs written in traditional programing languages such as C, C++, and Java [18], [19]. In our previous work presented at ISSRE 2014 [20], we introduced this type of model for spreadsheet debugging. Unfortunately, DBMs are less accurate than VBMs: they compute significantly more diagnoses compared to VBMs. For our running example, we obtain two single fault diagnoses, B5 and C5, when using the DBM. Even though DBMs have a lower diagnostic accuracy than VBMs, they have a convincing advantage: they require significantly less computation time. VBMs often cannot be solved by state-of-the-art constraint and SMT solvers, but DBMs for the same underlying spreadsheets are solved in less than 1 s.

Small modifications in the DBM improve its diagnostic accuracy [20]. However, this improved DBM still has, on average, a higher number of diagnosis candidates than the VBM. MBSD

will only be accepted in practice if it 1) provides a small number of diagnosis candidates without missing the true fault, and 2) computes the diagnoses in real time.

This paper directly builds upon our previous work [20] and improves it in several ways as follows:

1) We present an approach that combines the value-based and the DBM. This approach allows for a small number of diagnoses while having a moderate computation time.
2) We refine our list of operations and functions where coincidental correctness might occur. Knowledge about coincidental correctness is required for the improved DBM.
3) We provide an in-depth evaluation of the basic models as well as their combination.

The remainder of this paper is organized as follows: In Section II, we discuss related research both in the field of MBSD and in the field of spreadsheet debugging. In Section III, we briefly review the basic definitions of the spreadsheet language. In Section IV, we explain MBSD for spreadsheets and the different types of models that we have introduced for spreadsheets in our previous work [20]. In addition, we illustrate the three different types of models by means of the running example. Afterward, we present in Section V the diagnosis partitioning approach, which combines the dependency based and the VBMs. In Section VI, we present the setup and the results of the empirical evaluation. The results of this evaluation are in line with the results obtained in our previous work [20] where we stated that the basic DBMs compute more diagnoses, but in a fraction of the time required for the VBM. In addition, we show, in this new evaluation, that the DBM solves more spreadsheet diagnosis problems than the VBM. Furthermore, the empirical evaluation shows that the DBMs with diagnosis partitioning have a 30% lower median runtime than the median runtime of the VBM. The search space is reduced by more than 90% for the majority of the spreadsheets when using MBSD. We conclude the paper in Section VII.

II. Related Work

Weiser [30], [31], Shapiro [32], and Reiter [21] were among the first who focused on automated debugging. Reiter [21] has laid the foundations for modern MBSD with his theory about "Reasoning from first principles." He used a system description and a set of observations as logical sentences to identify the faulty parts of circuits. Reiter's diagnosis algorithm, namely model-based diagnosis (MBD), computes all minimal diagnoses; a diagnosis is minimal if no proper subset of the diagnosis is a diagnosis. Greiner *et al.* [29] presented a corrected version of Reiter's diagnosis algorithm.

While Reiter has focused on hardware debugging, Console *et al.* [7] and Bond [33] used the ideas of MBD [21] to improve debugging for software: They used Reiter's algorithm to eliminate parts of the source code that does not explain an observed error. With their paper [7], Console *et al.* started the research area called MBSD.

Nowadays, MBSD has a wide area of application in the software domain: Besides the application of MBSD to logic programing languages [7], MBSD has successfully been used in hardware design languages [18], [34] and functional languages [35]. Wotawa [19] has investigated the relationship of DBMs and program slicing. Mayer and Stumptner [36], [37] focus on the different types of MBSD models. In addition to the value-based and DBMs, they propose abstraction-based models. Mayer's Ph.D. thesis [38] summarizes and compares different types of dependency based and VBMs. Another work of Mayer *et al.* that is worth mentioning is their experience report about using VBMs in the debugging process [39].

Mateis *et al.* [40], [41] were among the first who applied MBSD to an object-oriented language, namely Java. While Mateis *et al.* use DBMs, Wotawa *et al.* [23], [42] build upon VBMs. They explain how to convert a debugging problem into a constraint satisfaction problem (CSP). Another work of Wotawa *et al.* [43] deals with the computational costs of MBSD: they illustrate CSPs as hypertrees, because the width of hypertrees indicates the complexity of the CSP: Hypertrees with a width > 5 are hard problems. Wotawa *et al.* showed that the hypertree width is often greater than 5, and therefore, debugging is a hard problem.

Jannach and Engler [44] used constraint solving for spreadsheet debugging. In a follow-up work, Jannach and Schmitz [8] presented the EXQUISITE debugging tool as an add-on for MS Excel. Abreu *et al.* [9]–[11] developed a model-based debugging approach for spreadsheets which converts a spreadsheet into a value-based CSP. In contrast to Jannach and Schmitz's approach, Abreu's approach relies on a single test case, and Abreu *et al.* directly encode the reasoning about the correctness of cells into the CSP. The VBM used in this paper directly builds upon the model presented in [9], [10], and [11]. Ayalew and Mittermeir [45] propose a slicing approach for spreadsheet fault localization. In this approach, the number of incorrect successor and predecessor cells influences the ranking of a cell in the fault localization. Ruthruff *et al.* [46] adapted spectrum-based fault localization to the spreadsheet domain.

Abraham and Erwig presented GoalDebug [47], [48]: a tool supporting end users in the debugging process by generating repair candidates for faulty cells. In addition to GoalGebug, the authors presented a list of mutation operators for spreadsheets [28] and the UCheck system [49] which detects errors that are caused by unit faults. Coblenz *et al.* [50] developed an approach which is close to UCheck. In their approach, Coblenz *et al.* also use header information to reason about errors.

Besides spreadsheet debugging approaches, there are testing approaches, e.g., WYSIWYT [51] Expector [65], [52], AutoTest [53], and metamorphic testing [69]. Testing approaches can be used to identify a misbehavior, i.e., a wrong result. Once abnormal behavior has been identified, our MBSD approach can be used in order to locate the root source of the observed misbehavior, i.e., the faulty formula(s). WYSIWYT works as follows: users indicate correct and erroneous values in their spreadsheet and WYSIWYT informs the users about the testedness of their spreadsheet via computing metrics relying on the definition-use (DU) coverage criteria: A cell C is used in another cell C_1 for computing C_1's value or within C_1's predicate (if C_1's formula contains a conditional). DU links a definition of C with a use

of C. A DU is exercised when there are inputs which cause that the definition of C is used in C_1. When all possible DU pairs are covered by the given test cases, 100% DU coverage is achieved. In contrast, Expector makes use of test formulas which are basically conditional formulas, e.g., `IF(A1 > 10;''OK'';''ERROR'')`. The work mentioned in [52] and AUTOTEST [53] are automated testing techniques. Metamorphic testing originates from the software domain [70]. Poon *et al.* [69] proposed to use this testing technique to test spreadsheets which suffer from the test oracle problem. These are for example large spreadsheets where the computation of concrete values is impractical due to large amount of data.

Several researchers have adapted techniques well known from software engineering to spreadsheets, e.g., assertions [55], model-driven spreadsheet engineering [56], complexity metrics for spreadsheets [59], and code smells [60], [61]. In particular, code smells are important for improving the quality of spreadsheets, which is defined by the terms usability, maintainability, and error frequency. Amongst other smells, in particular, code smells defined for object-oriented programs (e.g., coupling and cohesion of classes) were successfully used in the spreadsheet domain [60], [61]. Besides their work on code smells, Hermans *et al.* focused on the visualization of the dataflow in spreadsheets [62], data clone detection [63], and complexity metrics [64]. Cunha *et al.* [57] have developed a catalog of smells for spreadsheets. This catalog covers statistical smells (e.g., the standard deviation smell), type smells (e.g., empty cell and pattern finder), content smells (e.g., string distance and reference to empty cells), and functional dependency based smells (e.g., quasi-functional dependencies). A follow-up work [54] used these smells to localize faulty cells within spreadsheets. The concept of code smells differs from MBSD in the following aspect: Code smells match certain patterns, and therefore, point to cells which might cause a problem with respect to complexity (e.g., interworksheet smells) and wrong input data (e.g., typos detected via standard deviation or string distance). MBSD requires that a misbehavior, i.e., an erroneous output is already detected, e.g., by test cases, which is not necessary for code smell techniques. In contrast, MBSD identifies faulty formulas which might not be detected by pattern matching approaches.

Several researchers have addressed the challenge of automatically creating models from spreadsheets. For example, Cunha *et al.* [13] have inferred ClassSheet models from spreadsheets. ClassSheet models are the spreadsheet counterpart to entity relationships for databases. The intent of [13] is to obtain a business model from an existing spreadsheet. In contrast, our models are not mental models, and therefore, they cannot support the user in understanding a spreadsheet. In [15], Cunha *et al.* present a framework which supports users in the model-driven development of spreadsheets. This framework supports the user to coevolve the model and the data. The developed models represent the entity relationships of the data and they strongly differ from the models which we automatically derive from faulty spreadsheets for debugging purposes. Another interesting work of Cunha *et al.* [14] directly focuses on the conversion of spreadsheets into relational databases and vice versa. This approach is used to detect data redundancy errors. In contrast, our approach helps to identify faulty formulas once an erroneous output has been detected.

For an exhaustive survey on different techniques and methods for detecting, localizing, and repairing spreadsheets, we refer the interested reader to Jannach *et al.* [66].

III. PRELIMINARIES

To be self-contained, we briefly discuss the basic definitions required in this paper and state the spreadsheet debugging problem formally. Most of the definitions given in this section are taken from [9]. We introduce a spreadsheet Π as a set of cells, where each cell c has an attached formula $\ell(c)$ and a value $\nu(c)$. We assume that if a cell has no formula, then ℓ returns 0, and its value is undefined (ϵ). The value of each cell is computed directly from the execution of its corresponding ℓ function. For specifying formulas in [9], Abreu *et al.* introduce the programing language $\mathcal{L}$, which we describe later. When executing a formula, the resulting value might be an error $\perp$ (in case of a faulty execution), or any other defined data-type like Number, Boolean, or String. For simplicity, we ignore undefined cells, which are never used in a formula) and assume that there is a function *CELLS* that maps a spreadsheet Π to a set of cells that are either used in a formula or have an attached formula.

Cells might be accessed using their column and row number. In most spreadsheet languages, the rows have numbers and columns have a corresponding letter, e.g., `A3` denotes the cell in the third row and the first column (A). In [9], Abreu *et al.* also introduce a function φ mapping cell names from $CELLS(\Pi)$ to their corresponding position (x, y) where x represents the column and y the row number. The functions φ_x and φ_y return the column and row number of a cell, respectively. In addition to accessing single cells, all spreadsheet languages also allow for accessing sets of cells that are in close proximity. These cell areas are defined as follows:

Definition 1 (Area (from [9])): An area $c_1 : c_2$ is the set of neighboring cells inside the box spanned by the cells c_1, c_2:

$$c_1 : c_2 \equiv_{def} \left\{ c \in CELLS(\Pi) \;\middle|\; \begin{array}{l} \varphi_x(c_1) \leq \varphi_x(c) \leq \varphi_x(c_2) \wedge \\ \varphi_y(c_1) \leq \varphi_y(c) \leq \varphi_y(c_2) \end{array} \right\}.$$

In the following, we discuss the language $\mathcal{L}$ taken from [9] but we restrict our view on its syntax, because this part is needed for the other definitions. We refer the interested reader to [9] for the details of the semantics of $\mathcal{L}$ as well as functions mapping $\mathcal{L}$ to a constraint representation. It is worth noting that $\mathcal{L}$ relies on the cell values, constants, operators, and functions for computing values of other cells. Recursive functions are not allowed in $\mathcal{L}$ because in practice they are not often used.

Definition 2 (Syntax of $\mathcal{L}$ (from [9]; slightly modified)): The syntax of $\mathcal{L}$ is recursively defined as follows:

1) Constants k representing ϵ, number, Boolean, or string values are elements of $\mathcal{L}$ (i.e., $k \in \mathcal{L}$).
2) All cell names are elements of $\mathcal{L}$ (i.e., $CELLS(\Pi) \subset \mathcal{L}$).
3) Areas $c_1 : c_2$ are elements of $\mathcal{L}$.
4) If $e_1, e_2, \ldots, e_n$ are elements of the language ($e_1, e_2, \ldots, e_n \in \mathcal{L}$), then the following expressions are also elements of $\mathcal{L}$:

1) (e_1) is an element of $\mathcal{L}$.
2) If o is an operator ($o \in \{+, -, *, /, \leq, =, \geq\}$), then $e_1\ o\ e_2$ is an element of $\mathcal{L}$.
3) A function call $f(e_1, \ldots, e_n)$ is an element of $\mathcal{L}$ where f denotes functions like (but not limited to) IF, SUM, AVG.

In this paper, we need information regarding the dependencies of cells. These dependencies are obtained directly from the information given in the formulas. Like in other programing languages, we are able to define a data dependence between cells, if the function assigned to one cell makes use of the value of another cell. To define these dependencies, we introduce the function $\rho : \mathcal{L} \mapsto 2^{CELLS(\Pi)}$ returning the set of referenced cells for a given formula from $\mathcal{L}$.

Definition 3 (The function ρ (from [9])): Let $e \in \mathcal{L}$ be an expression. We define ρ recursively as follows:

1) If e is a constant, then $\rho(e) = \emptyset$.
2) If e is a cell c, then $\rho(e) = \{c\}$.
3) If $e = (e_1)$, then $\rho(e) = \rho(e_1)$.
4) If $e = e_1\ o\ e_2$, then $\rho(e) = \rho(e_1) \cup \rho(e_2)$.
5) If $e = f(e_1, \ldots, e_n)$, then $\rho(e) = \bigcup_{i=1}^{n} \rho(e_i)$.

From the definition of ρ, direct data dependencies can be easily defined.

Definition 4 (Direct Data Dependency (from [9])): A cell c is direct data dependent on another cell c' if and only if the cell c' is referenced in $\ell(c)$:

$$dd(c', c) \Leftrightarrow c' \in \rho(\ell(c)).$$

It is useful to distinguish different types of cells. There are cells that do not reference other cells but that are referenced by others. Such cells work as input to further computations. Moreover, there are cells, which reference other cells but which are never referenced by other cells. Such cells are outputs for a given spreadsheet Π.

Definition 5 (Input Cell (from [9])): An input cell c is a cell that does not reference another cell c', i.e., there exists no cell c' with a direct data dependency to c. The set of input cells is computed as follows:

$$Input(\Pi) = \{c | (\nexists c' : dd(c', c) \wedge \exists c'' : dd(c, c''))\}.$$

Definition 6 (Output Cell (From [9])): An output cell is a cell that references other cells but is not referenced by other cells. The set of output cells is computed as follows:

$$Output(\Pi) = \{c | (\nexists c' : dd(c, c') \wedge \exists c'' : dd(c'', c))\}.$$

It is often the case that spreadsheets contain cells that serve only as labels. Such cells are not referenced by other cells, and thus, do not contribute to any computations. Hence, we are able to exclude these cells from the set of input cells. From here on, we also assume that such cells are removed from $CELLS(\Pi)$.

Using the already introduced definitions, we are accordingly to [9] able to state the spreadsheet debugging problem formally. For this purpose, we first introduce the notation of environments. Environments are used to store expected values for cells. Note that there might be cells for which no value is defined.

Definition 7 (Environment (from [9])): An environment is a set of pairs (c, v) where c is a cell and v its value. There is at most one pair for each cell in an environment.

Environments form the basis for defining test cases formally.

Definition 8 (Test case (from [9])): A test case for a spreadsheet Π is a tuple (I, O) where I is the input environment specifying the values of all input cells used in Π, and O the environment defining expected values for some formula cells of Π.

This definition slightly differs from our previous definition [20] as expected values can now be indicated for arbitrary formulas instead of output cells only. There is no need to specify values for all output cells.

Example 1: A test case for our running example from Fig. 1 is $I = \{B2 = 0, B3 = 2, B4 = 10, C3 = 0, C4 = 10,000, D3 = -4, D4 = 5\}$ and $O = \{B6 = 100, C6 = 200,100, D6 = 200,150, E2 = 0\}$.

Following the usual definitions from testing, we say that a test case fails if there exists at least one output cell whose calculated value differs from its expected value; otherwise, the test case passes. If a test case fails, we have a debugging problem, i.e., we need to identify the reasons for the deviation between the expected value and the computed value.

Definition 9 (Spreadsheet Debugging Problem (from [9])): Let $\Pi \in \mathcal{L}$ be a spreadsheet and T a failing test case of Π, then (Π, T) is a debugging problem.

Fault localization techniques such as MBSD provide solutions to the spreadsheet debugging problem by explaining why the spreadsheet has failed on test case T.

IV. Model-Based Spreadsheet Debugging

Given a spreadsheet debugging problem (Π, T), we are interested in finding the root cause for the failing test case T. Accordingly to Reiter [21], such a problem can be solved using a model that describes the components of a system, their behavior, and their interconnections, and observations. In the context of spreadsheet debugging, the model is a representation of the cells and their behavior, and the observations are representations of the failing test case. Usually, a (correct) formal model of a faulty system is not available. In particular in the spreadsheet domain, such formal models are hardly found. Therefore, the model in MBSD is built directly from the faulty system (or spreadsheet). Consequently, the model maps the real behavior instead of the expected behavior, which is encoded in the observations.

Let us assume that M is the model of Π, *CELLS* the cells of Π, i.e., $CELLS = CELLS(\Pi)$, and *OBS* the logical representation of the test case T. T can either be retrieved from a formal specification or from the user. Since a formal specification is rarely available in the domain of spreadsheets, the information stored in OBS is usually given by the user. In our running example, we argue that a person who is familiar with velocity knows what to expect approximately. The user needs not to indicate the exact expected value. An approximate value or even the information that the computed value is wrong is sufficient.

Algorithm 1: ConDiag$(M, CELLS, n)$ ([24], Modified).

Input: A constraint model M, the set of *CELLS* and the upper bound of the diagnosis cardinality n
Output: All minimal diagnoses up to the predefined cardinality n

1: DS $= \{\}$
2: **for** $i = 1$ **to** n **do**
3: CM $= M \cup \left\{ \left(\sum_{j=1}^{|CELLS|} \text{AB}[j] \right) == i \right\}$
4: $S = \mathcal{P}\left(\mathbf{CSolver}(\text{CM})\right)$
5: DS $=$ DS $\cup\, S$.
6: $M = M \cup \bigcup_{s \in S} \neg(s)$
7: **end for**
8: **return** DS

M consists of the behavior of the individual cells. The behavior of a cell $c \in CELLS$ is given in the form $\neg\text{AB}(c) \rightarrow Behav(c)$ where AB stands for abnormal—either the formula of cell c is abnormal (AB(c)) or the formula of c is correct—and $Behav(c)$ is a placeholder for the behavior of the cell: The cell's formula can be indicated in two ways either exact (VBM, see Section IV-A) or abstract (DBM, see Sections IV-B and IV-C). $(M, CELLS, OBS)$ is the corresponding *diagnosis problem* for a spreadsheet debugging problem (Π, T): If (Π, T) is a spreadsheet debugging problem, then the logical sentence $M \cup OBS \cup \{\neg\text{AB}(c) | c \in CELLS\}$ is not satisfiable[2], i.e., the model M (i.e., the real behavior) and the observations *OBS* (i.e., the expected behavior) are in contradiction.

Example 2: Assume we have the following sets of constraints as model of the real behavior M and the expected behavior *OBS*, i.e., there is a contradiction in the constraints: $M = \{\text{AB}(\text{c}_{A1}) \vee A1 = A2 + 1\}$ and $OBS = \{A1 = 1, A2 = 2\}$. The logical sentence $M \cup OBS \cup \neg\text{AB}(\text{c}_{A1})$ is not satisfiable:

$$
\begin{aligned}
& M \cup OBS \cup \neg\text{AB}(\text{c}_{A1}). && \leftrightarrow \\
& \{\text{AB}(\text{c}_{A1}) \vee A1 = A2 + 1\} \cup \{A1 = 1, A2 = 2\} && \\
& \cup \neg\text{AB}(\text{c}_{A1}). && \leftrightarrow \\
& \{false \vee A1 = A2 + 1\} \cup \{A1 = 1, A2 = 2\} && \leftrightarrow \\
& \{false \vee 1 = 2 + 1\} && \leftrightarrow \\
& \{false \vee false\} && \leftrightarrow \\
& false. &&
\end{aligned}
$$

The basic idea of MBSD is to find settings for the AB predicates which remove this contradiction. These settings are called diagnoses. Formally, a diagnosis is defined as follows:

Definition 10 (Diagnosis (from [21])): Given a diagnosis problem $(M, CELLS, OBS)$ then $\Delta \subseteq CELLS$ is a diagnosis, if and only if $M \cup OBS \cup \{\neg\text{AB}(C) | C \in CELLS \setminus \Delta\} \cup \{\text{AB}(C) | C \in \Delta\}$ is satisfiable. A diagnosis Δ is said to be minimal if and only if no proper subset of Δ is a diagnosis.

[2]A set of constraints $C = \{c_1, \ldots, c_n\}$ is consistent or satisfiable, if all constraints of this set are satisfiable ($c_1 \wedge \cdots \wedge c_n \neq \bot$).

We use constraint or SMT solvers for computing the diagnoses. Therefore, we convert the spreadsheet diagnosis problem into its corresponding CSP. A CSP [22] 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 these variables. A CSP solution assigns values to variables such that all constraints are fulfilled. For more information about constraint solving in the context of software debugging, we refer the interested reader to Wotawa *et al.* [23].

Algorithm 1 illustrates the usage of a constraint solver to compute diagnoses up to a predefined cardinality (e.g., all single, double, and triple faults). This algorithm is a modified version of the Algorithm ConDiag [24]. It takes a constraint representation M of the spreadsheet program Π, the set of cells $CELLS (= CELLS(\Pi))$, and the maximum fault cardinality n as inputs. In M, the abnormal predicates AB are represented as an array: for each cell $c \in CELLS$, there is an index $i \in \{1, \ldots, |CELLS|\}$ for which AB$[i]$ represents the predicate AB(c) in the constraint representation. In Line 3, a constraint which ensures that only diagnoses of size i are reported is added to the set of constraints. The resulting constraint representation CM is given to a constraint solver, which returns a set of possible values for the AB array elements (Line 4). We add this AB array to the set of diagnoses DS (Line 5). In addition, we add the negations of the found solutions as blocking clauses to the model (Line 6). These blocking clauses are necessary to prevent the solver from reporting supersets of the already found diagnoses. This ensures that the solver always computes minimal diagnoses.

We discuss three different types of constraint representations (models) and how to obtain them from spreadsheets in the following sections. All models are automatically derived from the spreadsheets—there is no need for user interaction. Since the spreadsheets from which the models are derived are usually faulty, the models also contain the fault(s). The faults can be automatically identified from the models using an MBD algorithm like ConDiag.

A. Value-Based Models

Abreu *et al.* [9]–[11] proposed VBMs for spreadsheet debugging. In order to be self-contained, we briefly explain the conversion of spreadsheets into VBMs (following the definitions given in [9]–[11], and [20]) in this section and demonstrate what the VBM for the running example looks like.

A value-based constraint system contains:

1) the input cells and their values,
2) the output cells and their expected values,
3) all formulas concatenated with their abnormal variable.

The constraint representation handles the formulas as equations instead of assignments. Equations make it possible to draw conclusions on the input from the output of a formula. The function Γ_V maps spreadsheets to a set of equations.

Definition 11 (The function $\Gamma_V(\Pi)$ (modified version of the ConvertSpreadsheet Algorithm from [9])): Let Π be a spreadsheet comprising the cells $\{c_1, \ldots, c_k\}$. The conversion of Π

into a set of equations is defined as follows:

$$\Gamma_V(\Pi) = \bigcup_{i=\{1,\ldots,k\}} \{AB_{ind(c_i)} \vee c_i == \Gamma_V(\ell(c_i))\}$$

where $ind(c_i)$ maps a cell to a unique index and Γ_V for expressions $e \in \mathcal{L}$ is recursively defined as follows (modified version of the CONVERTEXPRESSION Algorithm from [9]):

1) If e is a constant or a cell name c, then the constant is given back as result, i.e., $\Gamma_V(e) = c$.
2) If e is of the form (e_1), then $\Gamma_V(e) = (\Gamma_V(e_1))$.
3) If e is of the form $e_1\ o\ e_2$, then $\Gamma_V(e) = \Gamma_V(e_1)\ o^*\ \Gamma_V(e_2)$ where o^* is the corresponding constraint representation of operator o.
4) If e is of the form $f(e_1, \ldots, e_n)$, then $\Gamma_V(e) = f^*(\Gamma_V(e_1), \ldots, \Gamma_V(e_n))$ where f^* is the corresponding constraint representation of the function f .

The observations *OBS* are obtained from the test case (I, O) and comprise the values of the input cells and the expected values of the cells which are indicated in O. The conversion of a test case (I, O) into the set of constraints OBS is straightforward: For all $(x, v) \in I$ and for all $(x, v) \in O$, add the equation $x == v$ to the constraint representation. Formally, the conversion of a test case $T = (I, O)$ into a set of constraints OBS is defined as

$$\text{OBS}_V(T) = \bigcup_{(x,v)\in\{I\cup O\}} \{x == v\}.$$

When using the test case conversion described above and the specified conversion function Γ_V, we obtain the following constraints for our running example from Fig. 1(b):

Input:

Input:	*Output*:
$B2 == 0$	$B6 == 100$
$B3 == 2$	$C6 == 200, 100$
$B4 == 10$	$D6 == 200, 150$
$C3 == 0$	$E2 == 0$
$C4 == 10,000$	
$D3 == -4$	
$D4 == 5$	

Formula constraints:

$$AB(\text{cell}_{B5}) \vee B5 == B2 * B4 + B3 * B4 * B4/2$$
$$AB(\text{cell}_{B6}) \vee B6 == B5$$
$$AB(\text{cell}_{C2}) \vee C2 == B2 + B3 * B4$$
$$AB(\text{cell}_{C5}) \vee C5 == B2 * C4 + C3 * C4 * C4/2$$
$$AB(\text{cell}_{C6}) \vee C6 == B5 + C5$$
$$AB(\text{cell}_{D2}) \vee D2 == C2 + C3 * C4$$
$$AB(\text{cell}_{D5}) \vee D5 == D2 * D4 + D3 * D4 * D4/2$$
$$AB(\text{cell}_{D6}) \vee D6 == B5 + C5 + D5$$
$$AB(\text{cell}_{E2}) \vee E2 == D2 + D3 * D4.$$

The input and the output constraints are the set of observations OBS; the formula constraints are the model M. The logical sentence $M \cup OBS \cup \{\neg\text{AB}(c) | c \in CELLS\}$ (i.e., assuming all formulas are correct) is not satisfiable because the propagation of the input values in the model M leads to different output values for $C6$ and $D6$. The model M and the observations *OBS* are in contradiction. When calling ConDiag$(M \cup OBS, CELLS(\Pi), 1)$ for determining which formulas might be faulty (i.e., abnormal), the algorithm returns $\{\{C5\}\}$ as solution.

B. Original DBMs

When using DBMs [19], only the information about whether the computed values are correct is propagated. For example, if we have the formula "A1+A2" stored in a cell $A3$, then the value of $A3$ can only be correctly computed if 1) the values of $A1$ and $A2$ are correct, and 2) A3's formula 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 representing correct result cells $(\{c|(c, v) \in O \wedge v = \nu(c)\})$ are also initialized with *true*. The variables representing erroneous result cells $(\{c|(c, v) \in O \wedge v \neq \nu(c)\})$ are initialized with *false*. Formally, the conversion of a test case $T = (I, O)$ into the set of constraints OBS is defined as follows:

$$\text{OBS}_D(T) = \bigcup_{(x,v)\in I} \{x == \text{true}\} \cup \bigcup_{(x,v)\in O \text{ where } [\![x]\!]=v} \{x == \text{true}\} \cup \bigcup_{(x,v)\in O \text{ where } [\![x]\!]\neq v} \{x == \text{false}\}.$$

In addition to the test case, we convert the content of the cells to constraints by using the function Γ_D.

Definition 12 (The Function $\Gamma_D(\Pi)$): Let Π be a spreadsheet comprising the cells $\{c_1, \ldots, c_k\}$. The conversion of Π into a set of equations is defined as follows:

$$\Gamma_D(\Pi) = \bigcup_{i=\{1,\ldots,k\}} \{AB_{\text{ind}(c_i)} \vee \Gamma_D(c_i)\}$$

where $ind(c_i)$ is the unique index for cell c_i and Γ_D on a cell c_i maps the cell's dependencies to a rule-like representation.

The idea behind this conversion is as follows: Instead of using the concrete formulas in the constraints, only the correctness relation is modeled.

Definition 13 (The Function $\Gamma_D(c_i)$): If the input values of a formula stored in cell c_i are correct, then the formula stored in cell c_i must compute a correct value, i.e.,

$$\Gamma_D(c_i) = \left(\bigwedge_{c'\in\rho(c_i)} c'\right) \rightarrow c_i.$$

Details about this modeling for software written in an imperative language can be found in [19]. The dependency based constraints for our running example are as follows:

TABLE I
SITUATIONS WHERE COINCIDENTAL CORRECTNESS MIGHT OCCUR

Category	Functions / Operators	Description
Conditional functions	IF* SUMIF COUNTIF	The value computed by evaluating a conditional function might be correct even though the referenced cells contain wrong values. This is the case when the erroneous value does not contribute to the value computed by the formula because of the evaluation of the condition. For example, in the formula $\ell(C1) = IF(A1 > 0; B2; 0)$ the value of B2 only contributes to the final result if $A1 > 0$ evaluates to true.
Abstraction functions	MIN* MAX* COUNT* COUNTIF	The value computed by an abstraction function either uses only one value of the referenced area (MIN, MAX) or does not use the values at all (COUNT, COUNTIF). Therefore erroneous values are easily masked by abstraction functions.
Rounding functions	ROUND ROUNDUP ROUNDDOWN FLOOR	If the erroneous value has only a small delta compared to the correct value, a rounding function might mask the fault.
Absolute value	ABS	If a value is erroneous because it has the wrong sign, the fault gets masked by the absolute value function.
Boolean functions	AND OR	When using Booleans, a fault can be easily masked. For example, given that $\nu(A1) = false$, the formula $\ell(C1) =$ AND$(A1; B2)$ evaluates to *false*, independent of the (maybe erroneous) value of B2. The same scenario is valid for the OR function, where it is sufficient that one of the referenced cells evaluates to *true*.
Operators	PRODUCT* SUMPRODUCT POWER*	An erroneous value is masked when it is multiplied with 0. Similar, an erroneous value is masked if it is used in the power function and the second value (i.e. either the base or the exponent) is 0. If the base is 1, a erroneous exponent is also masked.
Relational Operators	$>$ $<$ $<>$	Erroneous values used in relational operators are often masked. For example, given that $\nu(A1) = 1$ and $\nu(B2) = 2$ (instead of $\nu(B2) = 3$), the formula $\ell(C3) = A1 > B2$ will evaluate to *false* and the fault is masked.
Irreversible functions	SIN COS TAN	In particular, periodic functions can easily mask an erroneous value. In case of the trigonometrical functions, an erroneous input is masked if it has an 2Π offset from the correct value.

Functions and operators marked with * are taken from our previous work [20]. There are Functions that belong to several categories.

$$\begin{array}{cc} \textit{Input}: & \textit{Output}: \\ B2 == \text{true} & B6 == \text{true} \\ B3 == \text{true} & C6 == \text{false} \\ B4 == \text{true} & D6 == \text{false} \\ C3 == \text{true} & E2 == \text{true} \\ C4 == \text{true} & \\ D3 == \text{true} & \\ D4 == \text{true} & \end{array}$$

Formula constraints:

$$\begin{aligned} AB(\text{cell}_{B5}) &\vee (B2 \wedge B3 \wedge B4 \rightarrow B5) \\ AB(\text{cell}_{B6}) &\vee (B5 \rightarrow B6) \\ AB(\text{cell}_{C2}) &\vee (B2 \wedge B3 \wedge B4 \rightarrow C2) \\ AB(\text{cell}_{C5}) &\vee (B2 \wedge C3 \wedge C4 \rightarrow C5) \\ AB(\text{cell}_{C6}) &\vee (B5 \wedge C5 \rightarrow C6) \\ AB(\text{cell}_{D2}) &\vee (C2 \wedge C3 \wedge C4 \rightarrow D2) \\ AB(\text{cell}_{D5}) &\vee (D2 \wedge D3 \wedge D4 \rightarrow D5) \\ AB(\text{cell}_{D6}) &\vee (B5 \wedge C5 \wedge D5 \rightarrow D6) \\ AB(\text{cell}_{E2}) &\vee (D2 \wedge D3 \wedge D4 \rightarrow E2). \end{aligned}$$

Let the input and the output constraints be the set of observations OBS and let the formula constraints be the model M. When calling ConDiag($M \cup OBS$,*CELLS*(Π),1), the algorithm returns {{B5},{C5}} as a solution. This DBM computes more diagnoses because of the implication. In the VBM, the cell B5 is excluded from the set of possible diagnoses because it is used to compute B6, 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 DBM (NDM) that is explained in the following section.

C. Novel Dependency-Based Models

In the NDM [20], we make use of 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 conversion of test cases is exactly the same as for the original DBM. The mapping of spreadsheets to constraints changes. We introduce the function Γ_N, which is similar to Γ_D.

Definition 14 (The Function $\Gamma_N(\Pi)$): Let Π be a spreadsheet comprising the cells $\{c_1, \ldots, c_k\}$. The conversion of Π into a set of equations is defined as follows:

$$\Gamma_N(\Pi) = \bigcup_{i=\{1,\ldots,k\}} \{AB_{\text{ind}(c_i)} \vee \Gamma_N(c_i)\}$$

where $ind(c_i)$ is the unique index for cell c_i and Γ_N on a cell c_i maps the cell's dependencies to a rule-like representation.

Unfortunately, using bi-implications instead of ordinary implications is not always correct. The bi-implication cannot be used in case of coincidental correctness [25]. Coincidental correctness has to do with fault masking where an output value is correct even when a faulty formula was involved in its computation. For example, consider a conditional statement "IF$(A1, A2, A3)$" where $A2$ is returned because $A1$ is true, and the value of $A3$ is erroneous. In this case, the fault that is propagated to $A3$ gets masked. When using a bi-implication, we would misleadingly assume that $A3$ is correct too. We have to treat formulas where coincidental correctness might happen

differently. Table I lists situations where coincidental correctness might occur. This table has been manually created and it is not complete 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. For simplicity, we introduce a function CC on a cell c that returns *true* if the equation used in c raises the problem of coincidental correctness, and *false*, otherwise.

When taking care of what we have said about fault masking, we are able to define Γ_N for cells as follows:

Definition 15 (The Function $\Gamma_N(c_i)$):

$$\Gamma_N(c_i) = \begin{cases} \Gamma_D(c_i), & \text{if CC}(c_i) = true, \\ \left(\bigwedge_{c' \in \rho(c_i)} c'\right) \leftrightarrow c_i, & \text{otherwise.} \end{cases}$$

For our running example, the constraints for the input and the output are the same as in the original DBM. The constraints for the formulas are the following:

$$\begin{aligned} AB(\text{cell}_{B5}) \ &\vee \ (B2 \wedge B3 \wedge B4 \rightarrow B5) \\ AB(\text{cell}_{B6}) \ &\vee \ (B5 \leftrightarrow B6) \\ AB(\text{cell}_{C2}) \ &\vee \ (B2 \wedge B3 \wedge B4 \leftrightarrow C2) \\ AB(\text{cell}_{C5}) \ &\vee \ (B2 \wedge C3 \wedge C4 \rightarrow C5) \\ AB(\text{cell}_{C6}) \ &\vee \ (B5 \wedge C5 \rightarrow C6) \\ AB(\text{cell}_{D2}) \ &\vee \ (C2 \wedge C3 \wedge C4 \rightarrow D2) \\ AB(\text{cell}_{D5}) \ &\vee \ (D2 \wedge D3 \wedge D4 \leftrightarrow D5) \\ AB(\text{cell}_{D6}) \ &\vee \ (B5 \wedge C5 \wedge D5 \rightarrow D6) \\ AB(\text{cell}_{E2}) \ &\vee \ (D2 \wedge D3 \wedge D4 \leftrightarrow E2). \end{aligned}$$

Let the input and the output constraints be the set of observations OBS and let the formula constraints be the model M. When calling ConDiag($M \cup OBS$,*CELLS*(Π),1), the algorithm returns {{C5}} as a solution—similar as when using the VBM. Does the NDM always result in the same number of diagnoses as the VBM? The empirical evaluation in Section VI shows that this is not the case. Therefore in the next section, we introduce the diagnosis partitioning approach which improves MBSD by combining the value-based and the DBMs.

With respect to the user input, this NDM differs from the original modelDBM in the following aspect: If a user classifies a cell value as correct, this information gets lost in the original DBM, since the information cannot be propagated backward in the model. Therefore, in the original modelDBM, the user only has to indicate incorrect cell values. If the user does not indicate cells whose values are correct, the diagnosis results of the NDM are equal to the results of the original DBM. For the VBM, both correct output values and expected values for incorrect output cells are required. Indicating expected values might be more challenging for end users than just determining that a value is incorrect.

Algorithm 2: ConDiag $(\Pi, (I, O), n)$.

Input: A spreadsheet Π, a failing test case (I, O), and the upper bound of the diagnosis cardinality n
Output: All minimal diagnoses of high and low priority up to the predefined cardinality n prioritized by their cardinality

1: Let *DM* be a DBM of Π and (I, O).
2: $DS_{DM} = \text{ConDiag}(DM, CELLS(\Pi), n)$
3: Let *VM* be the VBM of Π and (I, O) with abnormal predicated for all cells in DS_{DM}.
4: $DS_H = \emptyset$
5: $DS_L = \emptyset$
6: **for** $\Delta \in DS_{DM}$ **do**
7: $A = \emptyset$
8: **for** all cells;*c*;of;Π **do**
9: **if** $c \in \Delta$ **then**
10: $A = A \cup \{AB(c) = \text{true}\}$
11: **else**
12: $A = A \cup \{AB(c) = \text{false}\}$
13: **end if**
14: **end for**
15: **if CSolver**$(VM \cup A)$ is consistent **then**
16: $DS_H = DS_H \cup \{\Delta\}$
17: **else**
18: $DS_L = DS_L \cup \{\Delta\}$
19: **end if**
20: **end for**
21: **return** (DS_H, DS_L)

V. Diagnosis Partitioning

Whereas the DBMs have a low computational complexity, the VBM allows for computing the smaller number of diagnoses. In this section, we present an approach for combining these models. First, we compute the set of all diagnoses DS_{DM} using any of the DBMs. Afterward, we create the VBM, but instead of adding the abnormal predicates AB to all cells, we add these predicates only to those cells which are contained in any of the previously computed diagnoses DS_{DM}. Instead of computing all diagnoses in the VBM again, we set the AB predicate values according to the diagnoses in DS_{DM} and, then, ask the solver for satisfiability only. If consistency cannot be ensured, we would in principle remove this diagnosis from the set of overall diagnoses. Unfortunately, removing such diagnoses is not possible because there are cases where supersets of such diagnoses are the real diagnoses. In order not to lose such cases, we decided to classify such diagnoses as low priority diagnoses (LPD). Diagnoses passing the consistency check are classified as high priority diagnoses (HPD), which should be looked at first. This means that the VBM is used as a filter or classifier for the diagnoses computed by the DBMs.

In Algorithm 2, we summarize the diagnosis partitioning approach. In Line 1, we compute the dependency model of spreadsheet Π and the given failing test case (I, O). From this model, we compute the set of diagnoses DS_{DM}. In Line 3, we compute

the VBM for Π and (I, O), where only the cells that are in one of the diagnoses $\mathrm{DS}_{\mathrm{DM}}$ have abnormal predicates. Between lines 6 and 20, we apply filtering. For this purpose, every diagnosis $\Delta \in \mathrm{DS}_{\mathrm{DM}}$ is tested using the VBM. All elements of Δ are set to behave incorrectly (Line 10) and all other cells behave as expected (Line 12). The VBM together with these correctness assumptions are given to the constraint solver, which—in this case–checks consistency only. In case Δ does not lead to an inconsistency, we add Δ to the set of HPD (Line 16). Otherwise, the diagnosis is added to the set of LPD (Line 18). The sets of high priority and LPD are returned in Line 21.

We now discuss the complexity of the whole approach. The complexity of the conversion of a spreadsheet Π into a set of constraints is $O(|\Pi| * |e|)$ where $|e|$ equals to the number of operands in the most complex formula of Π. This complexity applies to both the value-based and the DBMs. As $|\Pi|$ consists of a finite set of cells and each formula consists of a finite set of subexpressions, the conversion obviously terminates. Algorithm 1 (ConDiag) has a time complexity of $O(2^{|\Pi|*|e|})$ in the worst case since 1) constraint solving is NP-complete and 2) we have $|\Pi| * |e|$ variables in the constraint model (there can be a temporary variable for each operator in each formula cell in Π). The loop enclosing the solver call (Lines 2–7) can be neglected because in practice we are only interested in single, double, and triple faults. Given that the solver terminates, the algorithm terminates too. Therefore, the time complexity of Algorithm 2 is also $O(2^{|\Pi|*|e|})$. In the worst case, the complexity of the overall approach is exponential in the size of the cells because of the underlying complexity of diagnosis and constraint solving. However, when restricting diagnosis computation to single or double faults, the overall approach is feasible.

VI. Empirical Evaluation

This section consists of two major parts: the empirical setup (discussing the prototype implementation, the used platform, and the evaluated spreadsheet corpus, see Section VI-A) and the results (see Section VI-B) showing the following:

1) that the DBM (both with and without diagnosis partitioning) solves more spreadsheet diagnosis problems than the VBM under given computational restrictions,
2) that the basic DBMs compute more diagnoses, but in a fraction of the time required for the VBM,
3) that the DBMs with diagnosis partitioning have a 30% lower median runtime than the median runtime of the VBM,
4) that all models reduce the search space for most of the investigated spreadsheets by more than 90%, and
5) that the model can also be used to diagnose double and triple faults.

A. Empirical Setup

We developed a prototype in Java for performing the empirical evaluation. In contrast to previous work [20] where we used Minion [16] for solving the constraints, this new prototype uses Z3 [17] because Z3 allows us to model spreadsheets containing real numbers and Z3 performs better than Minion [12]. The evaluation has been performed on a computer with an Intel Core i7-3639QM (2.40 GHz quad-core) processor and 8 GB RAM, using a 32-Bit Java VM 1.8.0 Update 20 within a 64-Bit Windows 7. The computation time is the average time over 25 runs. Since we are interested in a real-time approach, we set the timeout limit for Z3 to 5 min.

We evaluated the diagnosis models by means of spreadsheets of the publicly available EUSES corpus [26]. This corpus consists of spreadsheets from different domains, e.g., financial calculations, and student homework. The number and location of faults in this corpus is unknown. For evaluating the fault localization capabilities of the different models, we need, however, the location of the faults and additionally data which simulates the user input (i.e., information about correct and incorrect cell values). Therefore, we use a modified version of the corpus [27] that comes with predefined faults and expected values for output cells. This modified corpus contains only a subset of spreadsheets from the original corpus; more precisely, it contains only those spreadsheets which consist of more than five formulas and whose input cells are nonempty. This smaller corpus consists of 267 spreadsheets, the smallest containing six formulas, the largest containing 604 formulas. On average, a spreadsheet contains 105 formula cells. Faulty versions of the spreadsheets were created by randomly selecting formulas and applying mutation operators [28] on them. There are between one and five faulty versions per spreadsheet, each faulty file containing exactly a single fault. In total, there are 267 single-fault spreadsheets. A detailed list of the types of faults that where inserted can be found in [27]. The corpus can be downloaded from http://spreadsheets.ist.tugraz.at. This website additionally offers a qualitative and quantitative description of the corpus. We refer the interested reader to this website for more details about the characteristics of this corpus.

Since this corpus contains only spreadsheets with single faults, we created spreadsheets with double and triple faults by combining the faults. We created as many double and triple fault combinations as possible, but we could not create double and triple faults for every type of spreadsheet because the modified EUSES corpus contains many spreadsheets with a single faulty version. In cases where there existed several faulty versions of the spreadsheet, we created all possible combinations, i.e., for a spreadsheet with n single fault versions, we created $\binom{n}{2}$ spreadsheets with double faults and $\binom{n}{3}$ spreadsheets with triple faults. In total, we created 122 spreadsheets with double faults and 95 spreadsheets with triple faults. These newly created faulty versions can be downloaded from http://spreadsheets.ist.tugraz.at/wp-content/uploads/benchmarks/EUSES-double-triple-faults.zip.

B. Results

First, we indicate how many spreadsheets have to be excluded from the evaluation because Z3 results in a timeout or runs out of memory for at least one of the models. Afterward, we compare the required runtime for solving the models of the remaining spreadsheets. Finally, we compare the diagnostic accuracy when using the different models.

TABLE II
ABSOLUTE AND RELATIVE NUMBER OF SPREADSHEETS WHERE Z3 HAD A TIMEOUT OR RAN OUT OF MEMORY WHEN USING THE VBM, THE ORIGINAL DBM, THE NDM, AND THE DBMS WITH DIAGNOSIS PARTITIONING(*)

Single fault	VBM	DBM	NDM	DBM*	NDM*	Total
Timeouts						
Absolute	16	7	7	8	8	16
Relative	6.0%	2.6%	2.6%	3.0%	3.0%	6.0%
Out of memory						
Absolute	4	3	3	4	3	5
Relative	1.5%	1.1%	1.1%	1.5%	1.1%	1.9%
Multiple fault	VBM	DBM	NDM	DBM*	NDM*	Total
Timeouts						
Absolute	7	3	3	5	4	7
Relative	3.2%	1.4%	1.4%	2.3%	1.8%	3.2%

TABLE III
RUNTIME COMPARISON OF THE VBM, THE ORIGINAL DBM, THE NDM, AND THE DBMS WITH DIAGNOSIS PARTITIONING(*)

	VBM	DBM	NDM	DBM*	NDM*
Accumulated [s]	329.54	94.33	44.37	337.44	234.98
Average [s]	1.34	0.38	0.18	1.37	0.96
Median	0.024	0.002	0.002	0.016	0.017
Standard deviation	8.33	2.65	1.08	9.29	6.17

1) Timeouts and Out-of-Memory: Z3 could not handle all spreadsheet diagnosis problems because of occurring timeouts and out-of-memory exceptions. Table II indicates the absolute and relative numbers of the affected spreadsheets for the different types of models, where the rightmost column named "Total" indicates the total number of spreadsheets that have to be excluded from the evaluation because of either a timeout or an out-of-memory exception, i.e., this column states how many spreadsheets were excluded from the following evaluation. The DBMs have the lowest number of timeouts and out-of-memory problems, followed by the DBMs that use diagnosis partitioning. The VBMs have twice as many timeouts as the other models. There were no out-of-memory exceptions for the spreadsheets with double and triple faults. We excluded all spreadsheets where Z3 either had a timeout or ran out of memory from the following evaluation.

The DBMs (DBM and NDM) have fewer timeouts than the VBM because the variables used in the constraint models are Boolean instead of Integer and Real. This reduces the complexity of the solving process. The DBMs with diagnosis partitioning (DBM* and NDM*) have still fewer timeouts than the VBM because the number of variables in the constraint model is reduced in the model—there are only abnormal variables AB for the formulas that are already suspected to be a diagnosis. Having fewer abnormal predicates means that there is less reification required, which is an expensive task when solving constraints, because it prevents value propagation.

2) Runtime: In Table III, we show the accumulated and average runtime for the different types of models. The DBMs without diagnosis partitioning have the lowest runtime. The DBMs with diagnosis partitioning and the VBM have nearly the same average runtime. However, the median of the DBMs with diagnosis partitioning is lower than the median of the VBM. This is an indicator that the high average runtime of the DBMs is caused by outliers. The DBMs without diagnosis partitioning (DBM and NDM) can be significantly solved faster than VBM because of the restriction of the domain. 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 VBMs, the cell values have an infinite number of possible instantiations. When using the diagnosis partitioning for the dependency based values, the runtime increases again, but the median solving time is smaller than the median solving time of the VBM.

In Fig. 2, we compare the runtime of the models pairwise. Instead of showing all pairwise combinations, we show the most interesting ones. Please note that the scale is logarithmic. Therefore, data points shown on the top-right corner of the figures have a higher impact on the average runtime than other data points. This figure shows that both DBMs require approximately the same amount of solving time (DBM versus NDM). There is no superior model. The same holds for the DBMs with diagnosis partitioning (DBM* versus NDM*). When comparing the DBMs with diagnosis partitioning to the models without diagnosis partitioning (DBM versus DBM*/NDM versus NDM*), we see that the models with diagnosis partitioning always require more runtime. This is not surprising, since the diagnosis partitioning process starts after the diagnoses have been computed for the base models (DBM/NDM). When comparing the DBMs with diagnosis partitioning to the VBM (DBM* versus VBM/NDM* versus VBM), we see that for the majority of the spreadsheets, the DBMs with diagnosis partitioning compute the diagnoses faster than the VBM.

Since we want to use model-based debugging as a real-time approach, we are particularly interested in the number of diagnosis problems that could be solved within 1 s. Table IV lists the number of diagnosis problems that could be solved within 1 s. From the 267 diagnosis problems, 235 (88.0%) are solved within 1 s when using VBM, DBM* or NDM*; 237 (88.8%) are solved when using DBM or NDM.

Fig. 3 illustrates the number of diagnosis problems that can be solved within a certain amount of time. The DBMs without diagnosis partitioning (DBM and NDM) are able to solve a larger number of spreadsheet than VBM, DBM*, and NDM* within a certain amount of time. DBM* and NDM* perform slightly better than VBM. However, the advantage of the DBMs with diagnosis partitioning is that we can immediately present the results of DBM or NDM to the user. While the user inspects the diagnoses, we can refine them by applying the diagnosis partitioning approach (*).

3) Diagnostic Accuracy: Both the value-based and the DBMs (with and without diagnosis partitioning) can be used to debug spreadsheets which contain more than one fault. For evaluating the diagnostic accuracy of the different models, we, however, rely only on the single-fault corpus for the following reason: ConDiag reports only minimal diagnoses; a diagnosis is minimal if no proper subset of it is a valid diagnosis. By

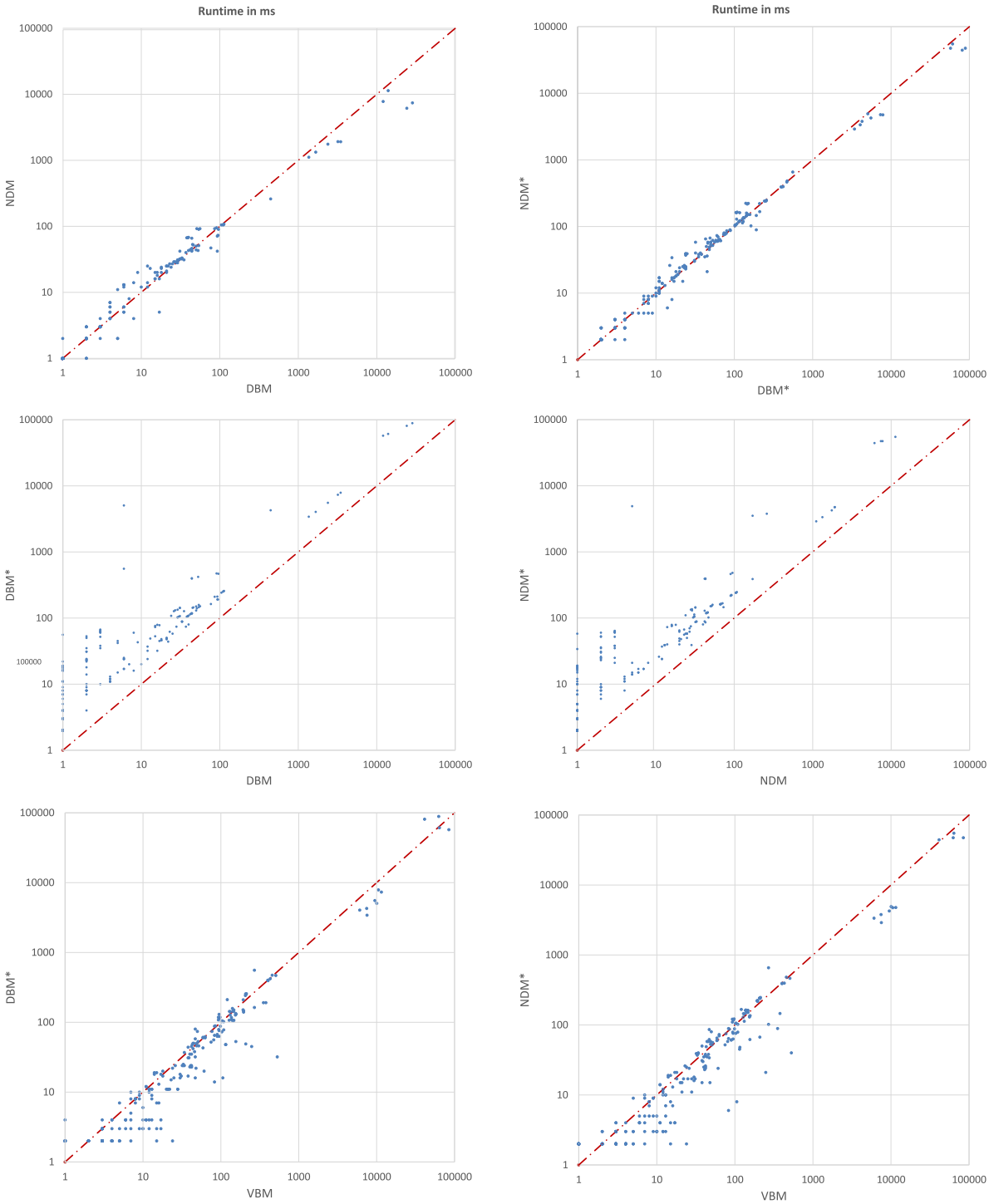

Fig. 2. Pairwise runtime comparison of the VBM, the original DBM with and without diagnosis partitioning (DBM*/DBM), and the NDM with and without diagnosis partitioning (NDM*/NDM). Each (blue) data point represents the runtime (in milliseconds) for one spreadsheet. Data points on the dashed (red) line indicate that the compared models have the same runtime for this spreadsheet. Data points below this line indicate that the model on the *y*-axis has a lower runtime than the model given on the *x*-axis.

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, or 2) we also count the number of all supersets of the minimal diagnoses. Option 1 would result in fewer diagnoses for the DBMs than the VBM because the VBM could compute combinations of pairs that are already reported as single-fault diagnoses in the DBMs. Option 2 would result in a huge number of diagnoses for all types of models (the value-based model would compute the smallest number of diagnoses, followed by

TABLE IV
ABSOLUTE NUMBER OF DIAGNOSIS PROBLEMS THAT COULD BE SOLVED IN REAL TIME (I.E., IN LESS THAN 1 S)

	VBM	DBM	NDM	DBM*	NDM*
real time	235	237	237	235	235
nonreal time	11	9	9	11	11

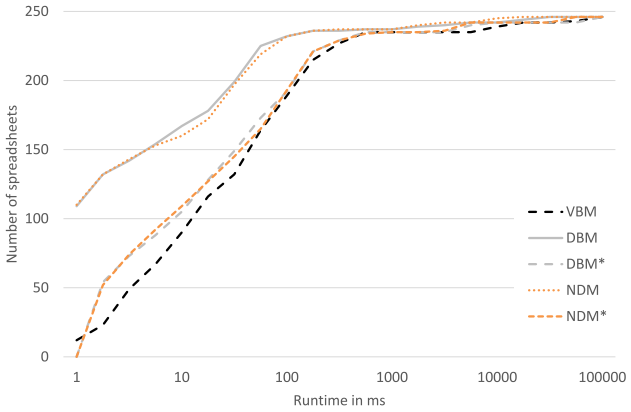

Fig. 3. Number of spreadsheets whose diagnosis problem could be solved within the indicated time.

TABLE V
ANSFD USING THE VBM, THE ORIGINAL DBM, THE NDM, AND THE DBMS WITH DIAGNOSIS PARTITIONING (DBM* AND NDM*)

VBM	DBM	NDM	DBM*	NDM*
6400	6555	6413	6400 HPD 155 LPD	6399 HPD 14 LPD

the NDM). For both options, the presented data would not be meaningful. Therefore, the following evaluation is restricted to the single-fault corpus.

Table V reports the accumulated number of single fault diagnoses (ANSFD) for the different models. We measure ANSFD as follows:

$$\text{Ansfd} = \sum_{S \in \text{corpus}} |\{\Delta \in \text{ConDiag}(S, \text{CELLS}(S), 1)\}|$$

i.e., we sum up the number of single-fault diagnoses for all spreadsheets that we investigate. For the DBMs with diagnosis partitioning (DBM* and NDM*), the diagnoses are divided into HPD and LPD. The true fault is always reported as a diagnosis for all types of models. The DBMs with diagnosis partitioning have approximately the same number of LPD as the VBM has in total. The DBMs without diagnosis partitioning have 2.4% and 0.2% more diagnoses than the VBM.

The NDM allows by means of the bi-implication to exclude more diagnoses. It computes nearly the same number of diagnoses as the VBM because both models rely on the same amount of information (i.e., cells with correct and incorrect values). In the original DBM, the information of correct cell values cannot be propagated which results in more diagnoses.

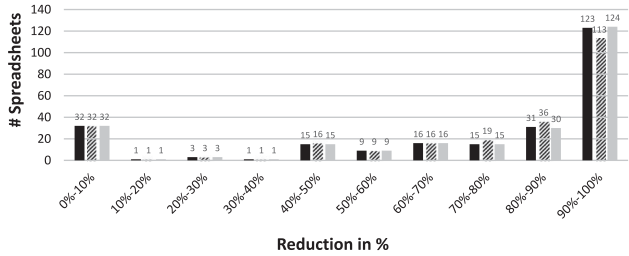

Fig. 4. REDUCTION of the search space for the VBM, the original DBM, and the NDM without diagnosis partitioning. The results for DBM* and NDM* are the same as for VBM.

We compare the diagnostic accuracy of the different models by means of the average achieved REDUCTION:

$$\text{Reduction} = \left(1 - \frac{|\text{Diagnoses}|}{|\text{Formula cells}|}\right) * 100\,\%.$$

The REDUCTION metric indicates reduction of the search space when using MBD in percentage of the total number of formula cells. This simplified formula applies for single-fault diagnoses. In case of higher order diagnoses, the formula is

$$\text{Reduction} = \left(1 - \frac{\left|\bigcup_{\text{Diagnoses}}\right|}{|\text{Formula cells}|}\right) * 100\,\%.$$

Fig. 4 illustrates to what extent the search space could be reduced for the basic models (without diagnosis partitioning). The reduction for the DBMs with diagnosis partitioning is not given because the results are the same as those of the VBM. The figure shows that for more than 50% of the spreadsheets, the search space is reduced by more than 90%. For 13% of the spreadsheet, no reduction or only a small reduction can be achieved because of the structure of these spreadsheets: They have only a single faulty output cell and no correct output values indicated. This small amount of information makes it difficult to reduce the search space automatically.

4) Multiple Faults: Having investigated the diagnostic accuracy of the models for single faults, we now investigate for how many spreadsheets with double and triple faults, a subset of the true fault is already reported as a single fault. The structure of the fault and the observations influence whether a fault is reported as a single, double, or triple fault. For example, in case of two subsequent faults (e.g., $A1 = 1$; $A2 = A1 * 3$; $B2 = A2$; with $A2$ and $B2$ being faulty) MBSD would point to at least one of these faults. Per definition all supersets of a diagnosis are also diagnoses. If either $A2$ or $B2$ is reported as a diagnosis, the superset $\{A2, B2\}$ is therefore also a diagnosis. As previously discussed, we only report minimal diagnoses and Fig. 5 shows that for approximately 50% of the spreadsheets with double faults, the fault is already reported as a single-fault diagnosis and that for more than 30% of the spreadsheets with triple faults, the fault is reported as a single fault. These results confirm our argumentation from Section VI-B3 that we cannot compare the diagnostic accuracy of the different models in case of multiple faults because subsets of the true fault can be reported as a diagnosis.

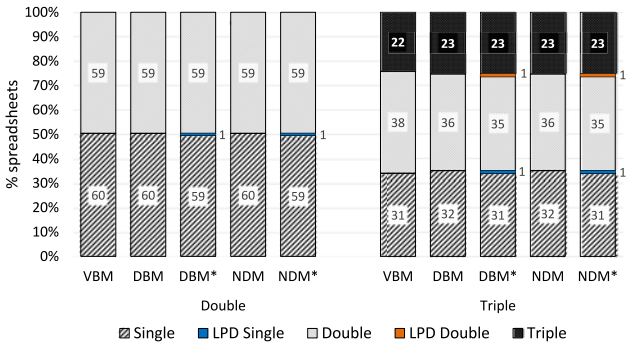

Fig. 5. Percentage of spreadsheets with double and triple faults where a subset of the true diagnosis is reported as a single, double, or triple fault.

Furthermore, Fig. 5 shows that in rare cases a LPD is a subset of the real fault from which we conclude that the LPD cannot be discarded. Three faults are reported as LPD (one double fault and two triple faults). If we would have discarded the LPD, our approach would not be able to report the correct diagnoses. However, this evaluation shows that users should focus on the HPD. They should only look at the LPD if they have not found the fault in the HPD.

Even though, we cannot compare the different models in case of multiple faults, all models can be used to locate multiple faults in practice. There are two possible scenarios for localizing n faults: 1) A real subset of the diagnosis is reported as diagnosis (either as a single fault diagnosis or as any other real subset). 2) The diagnosis is reported as a whole. In both scenarios, the users investigate the reported single-fault diagnoses first. In the first scenario, they would identify that one of the reported formulas is faulty. After correcting this formula, the users can rerun the fault localization process for $n-1$ faults. In the second scenario, they are not able to identify any faulty formula. Therefore, they investigate all double-fault diagnoses and so on until the real diagnosis is reported.

C. Threats to Validity

Like in other empirical evaluations, the obtained results depend on the underlying corpus used to carry out the experiments. In our case, we rely on the well-known EUSES spreadsheet corpus, which represents a wide area of different spreadsheets, ranging from financial reports over databases to students homework. The EUSES corpus has been often used in the spreadsheet domain for evaluation purposes. Although, the corpus can be considered as representative for spreadsheets, the faults introduced in the spreadsheets are artificial ones. Hence, the faults may not represent real faults occurring in daily used spreadsheets. However, the faults have not been introduced for a particular purpose, and thus, the obtained results cannot be considered as biased toward a certain direction.

Another thread to validity relies in the underlying method of comparison of the different debugging methods. For all different diagnosis approaches, we used the same setup for the experiments. Hence, the obtained results can be considered as fair as possible. This does hold for both the measured runtime for computing diagnoses as well as for the presented reduction. In the evaluation, we particularly focused on spreadsheets containing single faults because the number of diagnoses can be better compared as explained in Section VI-B3. The diagnostic behavior of spreadsheets with multiple faults might be different, since faults might mask each other or the diagnoses might contain only some of the faulty cells.

VII. Conclusion

Locating faulty formulas in spreadsheets can be extremely time consuming and exhausting especially when considering larger spreadsheets or spreadsheets written by others. In this paper, we address the challenge of fault localization support in spreadsheets by means of MBSD. We build upon previous work [20] where we have proposed an NDM. The advantage of DBMs lies in their lower computation times compared to VBMs. Unfortunately, DBMs compute more diagnoses than their value-based counterparts. The NDM proposed in [20] addresses this issue: By using the bi-implication instead of the implication, whenever possible, the number of diagnoses can be reduced. Originally, DBMs use implications instead of bi-implications because faults might have been masked otherwise (coincidental correctness). In case of fault masking, MBSD could not detect the faults when the bi-implication is used instead of an implication. In the NDM, the effect of fault masking is considered: The bi-implication is only used for modeling the behavior of those cells where fault masking cannot happen.

In this paper, we improve our previous work with the following three main contributions:

1) We improve the NDM by providing a comprehensive list of possible fault masking operations and functions (see Table I).
2) We combine VBMs and DBMs by using the VBM as a filter for the computed diagnoses (diagnosis partitioning). Instead of using the results obtained from the less computational demanding DBMs directly for focusing on certain cells, the diagnosis partitioning approach uses the VBM to check whether a diagnosis obtained from the DBMs is satisfiable when concrete values are used or not. The diagnosis partitioning approach classifies the diagnoses into two sets: the set of HPD and the set of LPD. HPD are diagnoses that are satisfiable even when considering the more accurate VBM.
3) We provide a comprehensive empirical evaluation comparing the different models. This evaluation shows that the DBMs solves more spreadsheet diagnosis problems than the VBMs. The DBMs without diagnosis partitioning require only a fraction of the time which the VBM requires. The diagnosis partitioning approach requires more runtime than the basic DBMs, but less time than the VBM—the median runtime is 30% lower—while still offering the same diagnostic accuracy. Besides the lower median runtime, the DBMs without diagnosis partitioning offer an additional advantage: First results can be early reported to the user. While the VBM still computes its diagnoses, these models provide early feedback to the user. The provided diagnoses are refined afterward.

Hence, the diagnosis partitioning technique is a promising candidate for being used in tools for spreadsheet fault localization.

Future research will include user studies and also comparisons with other fault localization approaches. Moreover, we plan to adapt the NDMs for other programing languages like imperative or object-oriented languages. The challenge of the adaption lies in the identification of all cases where fault masking could happen in object-oriented programs. Lyle and Weiser proposed to use program dicing [67] 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 programer 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. Another interesting research topic is the combination of MBSD with spreadsheet smells. Since MBSD and smells are orthogonal spreadsheet quality assurance techniques, they might complement each other.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their comments and valuable suggestions for improving the paper.

REFERENCES

[1] A. J. Ko *et al.*, "The state of the art in end-user software engineering," *ACM Comput. Surv.*, vol. 43, no. 3, 2011, Art. no. 21.

[2] D. Chadwick, B. Knight, and K. Rajalingham, "Quality control in spreadsheets: A visual approach using color codings to reduce errors in formulae," *Softw. Qual. J.*, vol. 9, no. 2, pp. 133–143, 2001.

[3] C. M. Reinhart and K. S. Rogoff, "Growth in a time of debt," *Amer. Econ. Rev.*, vol. 100, no. 2, pp. 573–578, 2010.

[4] T. Herndon, M. Ash, and R. Pollin, "Does high public debt consistently stifle economic growth? A critique of Reinhart and Rogoff," *Cambridge J. Econ.*, 2013. doi: 10.1093/cje/bet075.

[5] "Report of JPMorgan chase & co. management task force regarding 2012 cio losses," Jan. 2013. [Online]. Available: http://files.shareholder.com/downloads/ONE/2261602328x0x628656/4cb574a0-0bf5-4728-9582-625e4519b5ab/Task_Force_Report.pdf

[6] R. R. Panko, "Thinking is bad: Implications of human error research for spreadsheet research and practice," 2008.

[7] L. Console, G. Friedrich, and D. T. Dupré, "Model-based diagnosis meets error diagnosis in logic programs," in *Proc. Int. Joint Conf. Artif. Intell.*, 1993, pp. 1494–1501.

[8] D. Jannach and T. Schmitz, "Model-based diagnosis of spreadsheet programs-A constraint-based debugging approach," *Autom. Softw. Eng.*, vol. 23, no. 1, pp. 105–144, 2014.

[9] R. Abreu, B. Hofer, A. Perez, and F. Wotawa, "Using constraints to diagnose faulty spreadsheets," *Softw. Qual. J.*, vol. 23, no. 2, pp. 297–322, 2014. [Online]. Available: http://dx.doi.org/10.1007/s11219-014-9236-4

[10] R. Abreu, A. Riboira, and F. Wotawa, "Constraint-based debugging of spreadsheets," in *Proc. Ibero-Amer. Conf. Softw. Eng.*, 2012, pp. 1–14.

[11] R. Abreu, A. Riboira, and F. Wotawa, "Debugging of spreadsheets: A CSP-based approach," in *Proc. 3rd IEEE Int. Workshop Program Debugging Softw. Rel. Eng. Workshops*, 2012, pp. 159–164.

[12] S. Außerlechner *et al.*, "The right choice matters! SMT solving substantially improves model-based debugging of spreadsheets," in *Proc. 13th Int. Conf. Qual. Softw.*, 2013, pp. 139–148.

[13] J. Cunha, M. Erwig, and J. Saraiva, "Automatically inferring classsheet models from spreadsheets," in *Proc. IEEE Symp. Vis. Lang. Human-Centric Comput.*, 2010, pp. 93–100.

[14] J. Cunha, J. Saraiva, and J. Visser, "From spreadsheets to relational databases and back," in *Proc. ACM SIGPLAN Workshop Partial Eval. Program Manipulation*, 2009, pp. 179–188.

[15] J. Cunha, J. Fernandes, P. Joao, J. Mendes, and J. Saraiva, "A bidirectional model-driven spreadsheet environment," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 1443–1444.

[16] I. P. Gent, C. Jefferson, and I. Miguel, "Minion: A fast, scalable, constraint solver," in *Proc. 17th Eur. Conf. Artif. Intell.*, 2006, pp. 98–102. [Online]. Available: http://dl.acm.org/citation.cfm?id=1567016.1567043

[17] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. 14th Int. Conf. Tools Algorithms Constr. Anal. Syst.*, 2008, pp. 337–340.

[18] G. Friedrich, M. Stumptner, and F. Wotawa, "Model-based diagnosis of hardware designs," *Artif. Intell.*, vol. 111, no. 2, pp. 3–39, Jul. 1999.

[19] F. Wotawa, "On the relationship between model-based debugging and program slicing," *Artif. Intell.*, vol. 135, pp. 125–143, Feb. 2002.

[20] B. Hofer and F. Wotawa, "Why does my spreadsheet compute wrong values?" in *Proc. 25th IEEE Int. Symp. Softw. Rel. Eng.*, 2014, pp. 112–121. [Online]. Available: http://dx.doi.org/10.1109/ISSRE.2014.23

[21] R. Reiter, "A theory of diagnosis from first principles," *Artif. Intell.*, vol. 32, no. 1, pp. 57–95, 1987.

[22] R. Dechter, *Constraint Processing*. San Mateo, CA, USA: Morgan Kaufmann, 2003.

[23] F. Wotawa, M. Nica, and I. Moraru, "Automated debugging based on a constraint model of the program and a test case," *J. Logic Algebraic Program.*, vol. 81, no. 4, pp. 390–407, 2012.

[24] I. Nica and F. Wotawa, "Condiag-computing minimal diagnoses using a constraint solver," in *Proc. 23rd Int. Workshop Princ. Diagn.*, 2012.

[25] X. Wang, S.-C. Cheung, W. K. Chan, and Z. Zhang, "Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 45–55.

[26] M. Fisher and G. Rothermel, "The EUSES spreadsheet corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[27] B. Hofer, A. Riboira, F. Wotawa, R. Abreu, and E. Getzner, "On the empirical evaluation of fault localization techniques for spreadsheets," in *Proc. 16th Int. Conf. Fundamental Approaches Softw. Eng.*, 2013, pp. 68–82.

[28] R. Abraham and M. Erwig, "Mutation operators for spreadsheets," *IEEE Trans. Softw. Eng.*, vol. 35, no. 1, pp. 94–108, Jan./Feb. 2009.

[29] R. Greiner, B. A. Smith, and R. W. Wilkerson, "A correction to the algorithm in Reiter's theory of diagnosis," *Artif. Intell.*, vol. 41, no. 1, pp. 79–88, 1989.

[30] M. Weiser, "Programmers use slices when debugging," *Commun. ACM*, vol. 25, no. 7, pp. 446–452, Jul. 1982.

[31] M. Weiser, "Program slicing," *IEEE Trans. Softw. Eng.*, vol. 10, no. 4, pp. 352–357, Jul. 1984.

[32] E. Shapiro, *Algorithmic Program Debugging*. Cambridge, MA, USA: MIT Press, 1983.

[33] G. W. Bond, "Logic programs for consistency-based diagnosis," Ph.D. dissertationFaculty Eng., Carleton Univ., Ottawa, ON, Canada, 1994.

[34] B. Peischl and F. Wotawa, "Automated source level error localization in hardware designs," *IEEE Des. Test Comput.*, vol. 23, no. 1, pp. 8–19, Jan./Feb. 2006.

[35] M. Stumptner and F. Wotawa, "Debugging functional programs," in *Proc. Int. Joint Conf. Artif. Intell.*, 1999, pp. 1074–1079.

[36] W. Mayer and M. Stumptner, "Model-based debugging-state of the art and future challenges," *Electron. Notes Theor. Comput. Sci.*, vol. 174, no. 4, pp. 61–82, May 2007. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2006.12.030

[37] W. Mayer and M. Stumptner, "Evaluating models for model-based debugging," in *Proc. 23rd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2008, pp. 128–137. [Online]. Available: http://dx.doi.org/10.1109/ASE.2008.23

[38] W. Mayer, "Static and hybrid analysis in model-based debugging," Ph.D. dissertation, , Sch. Comput. Inf. Sci., Univ. South Australia, Adelaide, Australia, 2007. [Online]. Available: http://books.google.at/books?id=P0aQNQAACAAJ

[39] W. Mayer, M. Stumptner, D. Wieland, and F. Wotawa, "Can AI help to improve debugging substantially? Debugging experiences with value-based models," in *Proc. Eur. Conf. Artif. Intell.*, 2002, pp. 417–421.

[40] C. Mateis, M. Stumptner, and F. Wotawa, "Locating bugs in java programs-first results of the java diagnosis experiment project," in *Proc. 13th Int. Conf. Ind. Eng. Appl. Artif. Intell. Expert Syst.*, 2000, pp. 174–183.
[41] C. Mateis, M. Stumptner, D. Wieland, and F. Wotawa, "Model-based debugging of Java programs," in *Proc. 4th Int. Workshop Autom. Debugging*, 2000. [Online]. Available: http://arxiv.org/abs/cs.SE/0011027
[42] F. Wotawa and M. Nica, "On the compilation of programs into their equivalent constraint representation," *Informatica (Slovenia)*, vol. 32, no. 4, pp. 359–371, 2008.
[43] F. Wotawa, J. Weber, M. Nica, and R. Ceballos, "On the complexity of program debugging using constraints for modeling the program's syntax and semantics," in *Proc. 13th Conf. Spanish Assoc. Artif. Intell.*, 2009, pp. 22–31.
[44] D. Jannach and U. Engler, "Toward model-based debugging of spreadsheet programs," in *Proc. 9th Joint Conf. Knowl.-Based Softw. Eng.*, 2010, pp. 252–264.
[45] Y. Ayalew and R. Mittermeir, "Spreadsheet debugging," *Comput. Res. Repository*, 2008. [Online]. Available: http://arxiv.org/abs/0801.4280
[46] J. Ruthruff *et al.*, "End-user software visualizations for fault localization," in *Proc. ACM Symp. Softw. Vis.*, 2003, pp. 123–132. [Online]. Available: http://doi.acm.org/10.1145/774833.774851
[47] R. Abraham and M. Erwig, "Goal-directed debugging of spreadsheets," in *Proc. IEEE Symp. Visual Lang. Hum.-Centric Comput.*, 2005, pp. 37–44.
[48] R. Abraham and M. Erwig, "GoalDebug: A spreadsheet debugger for end users," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 251–260.
[49] R. Abraham and M. Erwig, "Ucheck: A spreadsheet type checker for end users," *J. Visual Lang. Comput.*, vol. 18, pp. 71–95, Feb. 2007.
[50] M. J. Coblenz, A. J. Ko, and B. A. Myers, "Using objects of measurement to detect spreadsheet errors," in *Proc. IEEE Symp. Visual Lang. Hum.-Centric Comput.*, 2005, pp. 314–316.
[51] K. J. Rothermel, C. R. Cook, M. M. Burnett, J. Schonfeld, T. R. G. Green, and G. Rothermel, "WYSIWYT testing in the spreadsheet paradigm: An empirical evaluation," in *Proc. 22nd Int. Conf. Softw. Eng.*, 2000, pp. 230–239.
[52] M. Fisher, M. Cao, G. Rothermel, C. R. Cook, and M. M. Burnett, "Automated test case generation for spreadsheets," in *Proc. 24th Int. Conf. Softw. Eng.*, 2002, pp. 141–151.
[53] R. Abraham and M. Erwig, "Autotest: A tool for automatic test case generation in spreadsheets," in *Proc. IEEE Symp. Visual Lang. Hum.-Centric Comput.*, 2006, pp. 43–50.
[54] R, Abreu, J. Cunha, J. Fernandes, P. Martins, A. Perez, and J. Saraiva, "Smelling faults in spreadsheets," in *Proc. 30th IEEE Int. Conf. Softw. Maint. Evol.*, 2014 pp. 111–120.
[55] M. M. Burnett, C. R. Cook, O. Pendse, G. Rothermel, J. Summet, and C. S. Wallace, "End-user software engineering with assertions in the spreadsheet paradigm," in *Proc. 25th Int. Conf. Softw. Eng.*, 2003, pp. 93–105.
[56] J. Cunha, J. P. Fernandes, J. Mendes, and J. Saraiva, "MDsheet: A framework for model-driven spreadsheet engineering," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 1395–1398.
[57] J. Cunha, J. Fernandes, H. Ribeiro, and J. Saraiva, "Towards a catalog of spreadsheet smells," in *Proc. 12th Int. Conf. Comput. Sci. Appl.*, 2012, pp. 202–216.
[58] R. Mittermeir and M. Clermont, "Finding high-level structures in spreadsheet programs," in *Proc. 9th Working Conf. Reverse Eng.*, 2002, pp. 221–232.
[59] A. Bregar, "Complexity metrics for spreadsheet models," *Comput. Res. Repository*, 2008. [Online]. Available: http://arxiv.org/abs/0802.3895
[60] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting code smells in spreadsheet formulas," in *Proc. 28th IEEE Int. Conf. Softw. Main.*, 2012, pp. 409–418.
[61] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting and visualizing inter-worksheet smells in spreadsheets," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 441–451.
[62] F. Hermans, M. Pinzger, and A. van Deursen, "Supporting professional spreadsheet users by generating leveled dataflow diagrams," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 451–460.
[63] F. Hermans, B. Sedee, M. Pinzger, and A. van Deursen, "Data clone detection and visualization in spreadsheets," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 292–301. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486827
[64] F. Hermans, M. Pinzger, and A. van Deursen, "Measuring spreadsheet formula understandability," *Comput. Res. Repository*, 2012. [Online]. Available: http://arxiv.org/abs/1209.3517
[65] F. Hermans, "Improving spreadsheet test practices," *Proc. Conf. Center Adv. Stud. Collaborat. Res.*, 2013, pp. 56–69.
[66] D. Jannach, T. Schmitz, B. Hofer, and F. Wotawa, "Avoiding, finding and fixing spreadsheet errors–A survey of automated approaches for spreadsheet QA," *J. Syst. Softw.*, vol. 94, pp. 129–150, 2014.
[67] J. Lyle and M. Weiser, "Automatic program bug location by program slicing," in *Proc. 2nd Int. Conf. Comput. Appl.*, Jun. 1987, pp. 877–882.
[68] V. Grigoreanu, M. Burnett, S. Wiedenbeck, J. Cao, K. Rector, and I. Kwan, "End-user debugging strategies: A sensemaking perspective," *ACM Trans. Comput.-Hum. Interact.*, vol. 19, no. 2, pp. 5:1–5:28, 2012.
[69] P.-L. Poon, F.-C. Kuo, H. Liu, and T. Y. Chen, "How can non-technical end users effectively test their spreadsheets?" *IT People*, vol. 27, no. 4, pp. 440–462, 2014.
[70] T. Y. Chen, S. C. Cheung, and S. Yiu, "Metamorphic testing: A new approach for generating next text cases," Dept. Comput. Sci., Hong Kong Univ. Sci. Technol., Hong Kong, *Tech. Rep. HKUST-CS98-01*, 1998.

Birgit Hofer received the Ph.D. degree in computer science in 2013 and the Master's degree in software engineering and economics in 2009 from the Graz University of Technology, Graz, Austria.

She currently works as a Researcher at Graz University of Technology. Her main research interest comprises the automatic localization and correction of faults in imperative and object-oriented software and spreadsheets. In particular, she is interested in combinations of spectrum-based fault localization, model-based debugging techniques, and genetic programming.

Andrea Höfler received the Graduate degree in computer science from the Graz University of Technology, Graz, Austria, in 2015.

In the Master's thesis "On the Usage of Value- and Dependency-based Models for Spreadsheet Debugging with SMT Solvers," she focused on model-based fault localization for spreadsheets. Parts of the result of her Master's thesis have contributed to the work presented in this paper.

Franz Wotawa received the M.Sc. degree in computer science in 1994 and the Ph.D. degree in 1996 from the Vienna University of Technology, Wien, Austria.

He is currently a Professor of software engineering at the Graz University of Technology, Graz, Austria and the Dean of the Computer Science Faculty. Since the founding of the Institute for Software Technology in 2003 to the year 2009, he was the Head of the institute. His research interests include model-based and qualitative reasoning, theorem proving, mobile robots, verification and validation, and software testing and debugging. Beside theoretical foundations, he has always been interested in closing the gap between research and practice. For these purposes, he founded Softnet Austria in 2006, which is a nonprofit organization carrying out applied research projects together with companies. During his career, he has written more than 280 papers for journals, books, conferences, and workshops. He supervised 64 Master's and 27 Ph.D. students.

Dr. Wotawa has been a member of a various number of program committees and organized several workshops and special issues of journals. He is a member of the Academia Europaea, the IEEE Computer Society, ACM, the Austrian Computer Society (OCG), the Austrian Society for Artificial Intelligence, and a Senior member of the AAAI.