Automated Software Engineering manuscript No.
(will be inserted by the editor)

Model-based diagnosis of spreadsheet programs

A constraint-based debugging approach

Dietmar Jannach · Thomas Schmitz

Abstract Spreadsheet programs are probably the most successful example of end-user software development tools and are used for a variety of purposes. Like any type of software, they are prone to error, in particular as they are usually developed by non-programmers. While various techniques exist to support the developer in finding errors in procedural programs, the tool support for spreadsheet debugging is still limited. In this paper, we show how techniques from model-based diagnosis can be applied and extended for spreadsheet debugging by translating the relevant parts of a spreadsheet to a constraint satisfaction problem. We additionally propose both problem-specific and generalizable extensions to the classical diagnosis algorithms which help to detect potential problems in a spreadsheet based on user-provided test cases more efficiently. The proposed techniques were integrated into a modular framework for spreadsheet debugging and evaluated with respect to scalability based on a number of real-world and artificially created spreadsheets. An additional error detection exercise involving 24 subjects was performed to assess the general applicability of such advanced spreadsheet debugging techniques for end users.

Keywords Model-based diagnosis · Constraints · Spreadsheet programs

1 Introduction

Spreadsheets are interactive data organization and calculation programs which are usually developed by non-programmers for a variety of purposes in different environments. The most typical application scenario is probably the use of spreadsheets for financial calculations. According to a recent meta-survey by Panko et al. [43], the use of spreadsheets in industry is however not limited to financial

D. Jannach
TU Dortmund, Germany
E-mail: dietmar.jannach@tu-dortmund.de

T. Schmitz
TU Dortmund, Germany
E-mail: thomas.schmitz@tu-dortmund.de

departments and spreadsheets based on Microsoft Excel can be found in most departments and more or less at all levels of a company.

Unfortunately, like any other software application, spreadsheet programs[1] can contain errors. In some research studies on error rates, every single analyzed spreadsheet had at least one error in it, see e.g. [41]. This phenomenon can be explained by two particularities of how spreadsheets are created. First, spreadsheet programs are mostly not subject to standard quality assurance procedures like other corporate IT applications, which is why Panko et al. even call spreadsheet programs the "dark matter (and dark energy) of corporate IT". Second, spreadsheets are often developed by people with no formal education in software development so that in most cases no requirements specification exists or no systematic tests are done [45].

Despite this lack of formal quality control procedures, spreadsheets are often used as a basis for business decisions and a number of examples of real-world "horror stories" exist, where faulty spreadsheets caused significant financial damage to companies or government organizations[2]. Very recently for example, it became apparent that the often-cited analysis on the relationship of the gross domestic product growth rates and public debt from Reinhart and Rogoff [48] contained a number of possible flaws, one of them a simple range error in a spreadsheet calculation [30].

The problems of the risk associated with faulty spreadsheets are however not new and have been discussed soon after this type of programs became popular in the early 1980s, see, e.g., [15] or [17]. Since then, a number of proposals have been made in the literature for assuring and improving the quality of spreadsheets. Two obvious options are to try to improve the education level of the spreadsheet developers and to define formal QA procedures for spreadsheets[3]. Another class of approaches consists in the provision of additional tools within the spreadsheet environment that support the developer in the construction and validation of the application, e.g. through user-oriented visualizations or the provision of special techniques for testing, error localization and "repair" [1, 3, 4, 6, 12, 32, 47].

The work presented in this paper falls into the latter class of techniques and shows how model-based diagnosis (MBD) can be applied to help the user locate certain types of errors in spreadsheet applications. Model-based diagnosis is a general approach from the field of Artificial Intelligence (AI), which can be used to reason about the potential causes of an observed, unexpected behavior of a system [49]. Originally, MBD was mostly applied to find errors in hardware artifacts such as electronic circuits. Later on, it was however shown that the MBD principle can also be applied to various types of software systems including logic programs, VHDL specifications, ontologies and even to Java programs [14, 20, 24, 39].

In our previous work [32], we have shown how model-based diagnosis can in principle be used to detect errors in spreadsheet programs by transforming parts of the spreadsheet into a constraint satisfaction problem (CSP) [56]. A similar idea was proposed later on in [6]. The existing work in the area however has a number of limitations. The experimental evaluations were for example done based on com-

[1] In the following, we will use the terms "spreadsheet", "spreadsheet program" and "spreadsheet application" interchangeably as done in the literature.

[2] `http://www.eusprig.org/horror-stories.htm`, last accessed November 2013.

[3] The second option has its limitations because the non-existence of formal processes is probably one of the reasons for the success of spreadsheets in the first place.

parably small problem instances and the scalability of the approaches remained partially unclear. Furthermore, questions of how to integrate these techniques into existing spreadsheet environments have not been sufficiently discussed. In this work, we therefore aim to develop these existing works further in different ways. The contributions of this paper are as follows.

- We provide a precise formal characterization of how constraint programming and model-based diagnoses techniques can be applied to the spreadsheet debugging problem.
- Fast response times are crucial in interactive debugging. We propose novel algorithmic contributions to improve the scalability of the basic approach through automated dependency analysis and a parallelization approach that can take advantage of the architecture of modern desktop and laptop computers.
- Finally, we validate the approach in a systematic manner. We first analyze the run-time behavior and the scalability of our approach on both artificial and real-world spreadsheets. Furthermore, we report the results of a first user study whose goal was to assess to which extent our tool can actually help users find existing errors in spreadsheets.

The paper is organized as follows. After giving an introductory example, we will formally describe the foundations of our MBD- and constraint-based spreadsheet debugging approach in Section 2 and show – based on this formalization – how the candidate space can be pruned without losing any of the diagnoses. Section 3 describes our new algorithm for parallelized diagnosis computation. The results of a performance evaluation are reported in Section 4. In Section 5 we outline the functionality of the interactive Exquisite debugging environment, report the results of the user study and discuss open challenges and future works. The final sections address related works and give a summary of this paper.

2 Model-based spreadsheet diagnosis

2.1 Model-based diagnosis principles and example

Model-based diagnosis is a general technique to analyze the reasons of an unexpected behavior of an observed system. The main assumption of MBD usually is that we know some internals of the analyzed system: (a) the components it is made of, (b) how they are connected to each other and (c) how the components work when they are not faulty. A typical example for such a system is an electronic circuit consisting of a number of connected gates with a defined specification of the expected input-output behavior.

The other assumption of MBD is that the system has a number of observable inputs and outputs. Thus, given that the system's expected behavior is fully specified and deterministic, we know which outputs we can expect for some given inputs. A diagnosis problem arises in case we observe a discrepancy between the expected and the observed outputs. A *diagnosis* is then a subset of the components of the system, which, if they are assumed to be faulty[4], explain the system's unexpected behavior [49].

[4] When a component is considered to be faulty, we assume that it can produce an arbitrary output within the range of generally possible values.

Let us illustrate this idea using an example in which we apply the general MBD principle to a simple spreadsheet program as shown in Figure 1.

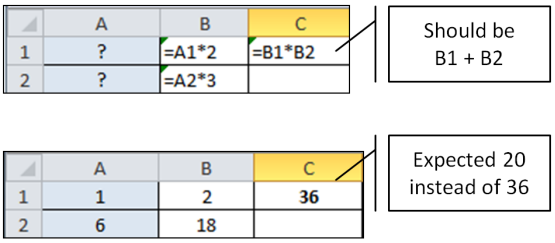

Fig. 1 A faulty spreadsheet and a test case leading to an unexpected output.

The example spreadsheet consists of two input cells (A1, A2) and three cells with formulas (B1, B2, C1). Since there is no cell with a formula referring to C1, we consider C1 to be an *output cell*. Let us assume that the spreadsheet developer made a mistake and entered a multiplication symbol instead of a plus in cell C1, which would correspond to a "mechanical spreadsheet error" according to the classification of [42].

When the user tests the spreadsheet he provides the values 1 and 6 as inputs for A1 and A2 and expects the value 20 for C1, i.e., the sum of B1 and B2. He however observes 36 as a value for C1, so the question arises, which of the formulas are wrong. When we apply the MBD principle to the spreadsheet program, we would consider the cells with formulas (B1, B2, C1) to be the components of the system. According to the above descriptions, a simple diagnosis would be to assume that all formulas are wrong. Obviously, this is not particularly helpful and we are usually interested in minimal subsets that explain the observation. The single-element set {C1} alone, however, would also be a diagnosis given this test case. We can check that by answering the question if there exists any formula for C1, which – given the inputs for A1 and A2 and assuming that the other two formulas are correct – leads to the observed output 20. Obviously, such formulas exist, including the intended one with the plus symbol but also the simple constant formula "=20".

There also might be other explanations for the observed faulty output 36 in C1. If we assume B2 to be faulty and B1 and C1 to be correct, possible formulas for B2 would be "=10" or "=A2+4" to end up with the expected result of 20 in C1. However, B1 alone cannot be the single source of the problem and a diagnosis if we assume that only integer values are allowed in cells. The reason is that B2 and C1 cannot be correct at the same time when the expected output is 20, because the assumedly correct value 18 for B2 is not a factor of 20.

Therefore, we can rule out B1 as a single-element diagnosis for this test case and the spreadsheet developer could focus on the two error candidates "C1 is incorrect" and "B2 is incorrect", one of which actually comprises the true cause of the error.

Note that different test cases can lead to different sets of diagnosis candidates[5] and in many situations we face the problem that the number of candidates can be high. Thus, if the user provides additional test cases, this should help us narrow

[5] In this paper we use the terms "diagnosis candidates" and "diagnoses" interchangeably.

down the candidates even further or help us to better discriminate the candidates. Let us assume that the user provided three more test cases as shown in Table 1.

ID	Input A1	Input A2	Expected at C1	Observed at C1	Diag. candidates
1	1	6	20	36	{B2}, {C1}
2	4	5	23	120	{B1, B2}, {C1}
3	15	10	60	900	{B1}, {B2}, {C1}
4	6	1	15	36	{B1}, {C1}

Table 1 Multiple test cases for the faulty spreadsheet.

We can observe that in fact the set of diagnosis candidates varies for each chosen test case and individual test cases such as the one with ID 3 are not very informative. However, we can also see as expected that {C1} as the true source of the problem is a diagnosis for all test cases. The only other combination of formulas that would explain all test cases is the set {B1, B2}, because {B1} alone is a diagnosis for the test cases 3 and 4, {B2} would explain test case 1, but only {B1, B2} in combination explains test case 2[6]. While {B1, B2} beside {C1} remains as a possible diagnosis given these 4 test cases, it represents a double-fault error and can be considered to be less likely than the single-element diagnosis {C1}. The multiple test cases have thus helped us to better focus the debugging process.

So far, we have only considered "positive" test cases that describe the expected system behavior. In a spreadsheet debugging example however also other forms of immediate user feedback are plausible. First, the user could specify that a given constellation of inputs and outputs is wrong, which would represent a "negative" test case in the sense of [20]. Second, the user might provide some assertions for individual cells as proposed for spreadsheet programs in [10].

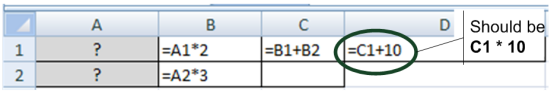

Fig. 2 Another faulty spreadsheet.

Consider the slightly extended spreadsheet program shown in Figure 2. In this case, the formula in C1 is designed to be an addition and correct, but an error occurred in cell D1 where the user entered a plus symbol instead of a multiplication. Let us assume that the user provided the test case {A1=4, A2=5, D1=230}. Unfortunately, this unluckily chosen test case would return the four single-element diagnoses {B1}, {B2}, {C1} and {D1}. However, if the user had already an idea about the possible value combinations for individual cells when designing the spreadsheet, he could have entered, for example, a plausibility-checking rule that states that whenever C1 is greater than 100, D1 cannot be less than 1000, i.e. $\neg$ (C1>100 $\wedge$ D1<1000). Given this additional piece of information, the space of possible diagnoses would immediately be narrowed down through the MBD principle to the single-element diagnosis {D1}.

[6] An algorithm for the treatment of multiple test cases during diagnosis is given in [20].

Vice versa, the user could also state assertions about generally allowed value constellations instead of specific forbidden value combinations. Let us assume that in our spreadsheet examples, plausible values for C1 can only be between 15 and 60 as in the examples shown in Table 1. The user could have added a corresponding assertion of the form (C1≥15 ∧ C1≤60). Given that additional information, the single test case would again be sufficient to detect {D1} as the only possible diagnosis[7].

Overall, the examples illustrate how the MBD principle can be applied to locate those formulas that can theoretically be the true cause of the error. Note however that the basic MBD-approach does not make any assumption about how the correct formula or a possible "repair" should look like[8]. In the following section, we will first review the formal definition of MBD according to Reiter and discuss how MBD techniques can be applied to spreadsheet debugging by transforming the spreadsheet into a corresponding declarative CSP representation.

2.2 Model-based diagnosis for Constraint Satisfaction Problems

2.2.1 Reiter's general theory of diagnosis

A formal characterization of model-based diagnosis based on first-order logic is given by Reiter in [49] and can be summarized as follows[9].

Definition 1 (Diagnosable System) A diagnosable *system* is described as a pair (SD, COMPONENTS) where SD is a system description (a set of logical sentences) and COMPONENTS represents the system's components (a finite set of constants).

The normal behavior of components is described by using a distinguished (usually negated) unary predicate AB(.), meaning "abnormal". A diagnosis problem arises when some observation $o \in$ OBS of the system's input-output behavior (again expressed as first-order sentences) deviates from the expected system behavior.

Definition 2 (Diagnosis) Given a diagnosis problem (SD, COMPONENTS, OBS), a diagnosis is a set $\Delta \subseteq$ COMPONENTS such that SD $\cup$ OBS $\cup\{$ AB(c)$|c \in \Delta\} \cup \{\neg$ AB$(c)|c \in$ COMPONENTS$\backslash\Delta\}$ is consistent.

In other words, a diagnosis is a subset of the system's components, which, if assumed to be faulty (and thus behave abnormally) explain the system's behavior, i.e., are consistent with the observations.

Definition 3 (Minimal Diagnosis) A diagnosis Δ for (SD, COMPONENTS, OBS) is minimal, if there exists no $\Delta' \subset \Delta$ which is also a diagnosis of (SD, COMPONENTS, OBS).

Finding all minimal diagnoses can in theory be done by simply trying out all possible subsets of COMPONENTS and checking their consistency with the observations. In [49], Reiter however proposes a more efficient procedure based on the concept of *conflicts*.

[7] Alternatively, the user could have restricted the domain size of variable C1.

[8] The correct formulas in the figures are only used to better illustrate the problem setting.

[9] In the formalization of MBD and CSP concepts, we will rely on the notations and terms used, e.g., in [49] or [20], to illustrate the correspondence of our approach with previous works.

Definition 4 (Conflict) A conflict for (SD, COMPONENTS, OBS) is a set $\{c_1, ..., c_k\} \subseteq$ COMPONENTS such that SD $\cup$ OBS $\cup\{\neg$ AB$(c_1), ... \neg$ AB$(c_k)\}$ is inconsistent.

A (minimal) conflict thus corresponds to a (minimal) subset of the components, which, if assumed to behave normally, are not consistent with the observations. Considering our first spreadsheet example in Section 2.1, the set of formulas {B2,C1} represents a *conflict* as considering both components to be correct leads to an inconsistency when we assume that only integers are allowed.

Reiter finally shows that finding all diagnoses can be accomplished by finding the *hitting set* of the given conflicts. Furthermore, he proposes a breadth-first search procedure and the construction of a corresponding hitting set tree (HS-Tree) to determine the minimal diagnoses. Later on, Greiner et al. in [26] found a potential error that can occur during the HS-tree construction in presence of non-minimal conflicts. To fix the problem, they proposed an extension to the algorithm which is based on a directed acyclic graph (DAG) instead of the HS-tree.

In [20], finally another extension to the algorithm was presented which supports the consideration of multiple positive and negative test cases during the construction of the HS-DAG, which is also particularly relevant in the context of our approach to model-based spreadsheet debugging.

2.2.2 Diagnosing Constraint Satisfaction Problems

Constraint satisfaction can be seen as a general framework for modeling and solving a number of combinatorial problems. Typical real-world applications that can be modeled as constraint satisfaction problems (CSP) include a number of operations research and resource allocation problems, crew scheduling and planning or product configuration.

Definition 5 (CSP) A CSP is defined as a triple $\langle X, D, C\rangle$ where $X = \{x_1, x_2, \ldots, x_n\}$ is a set of variables. Each variable in X has a defined range of allowed values called the domain. The domain of x_i is denoted as D_i. $D = \{D_1, D_2, \ldots, D_n\}$ is the set of all domains. C finally is a set of constraints which describe legal value constellations of the variables, that is, which values individual variables can simultaneously take.

The problem of solving a CSP consists in assigning to each variable x_i from X a value from its domain D_i in a way that none of the constraints is violated. Due to their applicability to a large class of problems and their high relevance in practice, a number of highly efficient deterministic and heuristic algorithms have been developed during the last decades. Likewise, highly optimized commercial and open-source solver libraries for different programming languages are available today.

Applying the MBD-principle to the problem of detecting faulty constraint specifications in a CSP based on test cases is straightforward[10].

Definition 6 (CSP Diagnosis Problem) A CSP diagnosis problem (CDP) is a triple $\langle CSP\langle X, D, C_{ok} \cup C_{pf}\rangle, TC^+, TC^-\rangle$, where $CSP\langle X, D, C_{ok} \cup C_{pf}\rangle$ is a

[10] The detection of faults which are caused by semantic errors or wrong domain definitions are beyond the scope of this work.

constraint satisfaction problem and TC^+ and TC^- are sets of positive and negative test cases. Each test case in TC^+ and TC^- consists of a set of constraints on the variables of the CSP. The set of the CSP's constraints $C_{ok} \cup C_{pf}$ consists of a subset C_{ok} of constraints which are considered to be correct and a subset C_{pf} of constraints which are possibly faulty.

Test cases can thus consist of both unary constraints on variables, e.g., $v1 = 2$, and test-case specific assertions such as $v2 \geq 3$. Furthermore, they can be complete or partial, meaning that not for all variables values have to be specified. Splitting the constraints of the CSP into a subset of assumedly correct and potentially faulty ones allows us to model user-specified assertions as described above or take user-provided input about the correctness of certain constraints into account during the diagnosis.

A solution to the CSP diagnosis problem consists of a subset Δ of the possibly faulty constraints C_{pf}. In order to be a diagnosis, every given test case must be "consistent" with the reduced constraint base from which this assumedly faulty subset is removed. This means that at least one solution for the relaxed CSP must exist. Furthermore, the relaxed CSP must not be consistent with the negative test cases. We achieve this by explicitly forbidding the constellations specified in TC^-.

Definition 7 (CSP Diagnosis) A diagnosis for a given CSP diagnosis problem $\langle CSP\langle X, D, C_{ok} \cup C_{pf}\rangle, TC^+, TC^-\rangle$ is a set $\Delta \subseteq C_{pf}$, such that $\forall tc^+ \in TC^+$ at least one solution for the $CSP\langle X, D, C_{ok} \cup (C_{pf} \setminus \Delta) \cup tc^+ \cup NTC\rangle$ exists, where NTC is the conjunction of the negated test cases in TC^-, i.e., $NTC = \{\bigwedge_{tc^- \in TC^-}(\neg tc-)\}$.

Example 1 Let us assume that the user specified the following negative test cases $TC^- = \{\{v1 = 1, v2 = 1\}, \{v1 = 4, v2 > 10\}\}$ for the variables $v1$ and $v2$ of the CSP. In order to explicitly disallow these combinations, we add the following set of constraints NTC to the CSP to be diagnosed: $NTC = \{(\neg(v1 = 1 \wedge v2 = 1)) \wedge (\neg(v1 = 4 \wedge v2 > 10))\}$.

The definition of the CSP diagnosis problem is similar to the logic-based approach presented in [20]. An important difference however is that "consistency" alone is not enough in the CSP setting. When considering CSPs, there might be partial positive test cases which are consistent with the constraints (even after constraint propagation). Still, this sort of consistency is no guarantee that the partial test case can be extended to a complete CSP solution. Therefore, we require that an arbitrary solution must exist for the modified constraint satisfaction problem.

A diagnosis for the CSP diagnosis problem exists if the positive and negative test cases do not contradict each other and the assumedly correct constraints.

Proposition 1 *A diagnosis for a CDP* $\langle CSP\langle X, D, C_{ok} \cup C_{pf}\rangle, TC^+, TC^-\rangle$ *exists, iff* $\forall tc^+ \in TC^+$ *a solution to the CSP* $\langle X, D, C_{ok} \cup tc^+ \cup NTC\rangle$ *exists.*

Proof Proposition 1 follows directly from Definition 7. Let TC^+ be a set of positive test cases to a CDP $\langle CSP\langle X, D, C_{ok} \cup C_{pf}\rangle, TC^+, TC^-\rangle$, so that $\forall tc^+ \in TC^+$ a solution to the CSP $\langle X, D, C_{ok} \cup tc^+ \cup NTC\rangle$ exists. If such a solution exists, the set $\Delta = C_{pf}$ according to Definition 7 is a diagnosis for the CDP. □

In other words, assuming all potentially faulty constraints to be actually faulty represents one (non-minimal) diagnosis that always exists in case the test cases and the assumedly correct constraints do not represent a contradiction.

2.2.3 Spreadsheets as CSPs

For the proposed spreadsheet debugging approach, we need a way to determine if a certain spreadsheet formula can in principle be the root cause for an unexpected calculation outcome. To perform this kind of reasoning, we propose to translate the relevant parts of the spreadsheet, i.e., the formulas of the cells and those cells referenced by the formulas, into a CSP. The general principle of such a translation is relatively simple and we can consider spreadsheet cells as CSP variables and formulas as constraints.

For the example in Figure 1, the problem variables would be A1, A2, B1, B2, and C1, the constraints would be "B1=A1*2", "B2=A2*3", "C1=B1*B2" and the single positive test case "{A1=6, A2=10, C1=20}". Since spreadsheet environments like MS Excel do not require explicit data type and domain definitions, appropriate domain definitions for the CSPs have to be derived through domain-independent heuristics or explicitly provided by the user[11].

Even though modern spreadsheet environments like MS Excel provide a lot of functionality including built-in scripting languages or advanced formatting, data organization and visualization facilities, the typical core functionality of spreadsheets is to do mathematical or logical calculations. Many of these mathematical operators and functions can be translated into the language of a typical modern constraint solver. Beside standard arithmetic and logical operators, also IF-THEN-style expressions or the handling of string constants can be encoded as constraints. Overall, while clearly not all of the functionality of a spreadsheet environment can be translated, the proposed approach is able to cover the most typical type of applications where arithmetic functions, e.g. for data aggregation, are the most important aspect and help the developer find errors in formulas.

In standard CSP formulations, the constraints have no "direction" and, correspondingly, we have no explicit information about the inputs and outputs of the application itself. The constraint "C1=B1*B2" could thus be re-written as "B1*B2=C1" or "B2=C1/B1". Formulas in spreadsheets are however different as they are always functions (and not general relations) with one or several input values and one single output value.

In the following, we will show how the functional character of formulas can be used to prune the search space for diagnoses without losing any minimal diagnosis. To that purpose, we will introduce a formalism that describes the relevant aspects regarding the translation process from spreadsheets to CSPs with respect to cell dependencies[12].

Definition 8 (Spreadsheet Program) A spreadsheet program P consists of cells Z which can contain formulas, constant values, or are empty but are referenced by other cells. A formula in cell $z_i \in Z$ can reference other cells in Z. We denote the set of referenced cells of a cell z_i as *inputs*(z_i). The *input cells* in Z are those, which contain no formula but are referenced by other formulas. *Output cells* are cells which contain formulas but are not referenced by other cells.

[11] See later sections for a further discussion of the problem of domain definitions.

[12] In the literature, a number of formalizations for spreadsheets are proposed, e.g., in [1] or [3], and some of them provide their own grammar for expressions or detailed mapping rules to a different representation [6, 31]. Since the specific mapping of the spreadsheet formulas to the CSP representation is not relevant for the subsequently described pruning approach, we will introduce a simple notation that focuses only on the relevant cell dependencies.

The translation of a spreadsheet program P with cells Z to a $CSP\langle X, D, C\rangle$ in our approach is done according to the following rules:

- For each non-empty or referenced cell $z_i \in Z$ we create a CSP variable $x_i \in X$. The domain for x_i can be based on heuristics and defaults or explicitly defined by the spreadsheet designer.
- Whenever z_i contains a formula, we create a constraint $c_i \in C$. We will use the function $cell(c, Z)$ to denote the spreadsheet cell z of P containing the formula from which c was derived. Correspondingly, we will denote the set of constraints that were created from formulas in a set of cells Z as $Constraints(Z)$. With these functions, we can connect the derived constraint with the original spreadsheet to exploit structural information in the spreadsheet in the debugging process as will be shown later on.

In our debugging approach, we only consider cells as diagnosable components that contain formulas. Values in input cells are always assumed to be correct. Cells with constant values that are not referenced by other cells such as table headings or other labels are also not taken into account.

The information about the direction of the functional constraints and their deterministic nature can help us to focus our search for faulty formulas. Consider again our extended example for a faulty spreadsheet in Figure 2. Let us assume that the formula in C1 is wrong again (should be a plus) and the user did not only inspect the final outcomes but also some of the intermediate values. Correspondingly, he might specify that the value in C1 is wrong and that he expects a different value for a given test case. Since the cell formulas are functions and directed, we can immediately focus our search on the calculations "before" the faulty value in C1 and can ignore cell D1[13] using a dependency analysis.

Definition 9 (InputCells) Given a cell z_i of a spreadsheet program P, we will denote the set of cells that are directly or indirectly referenced by z_i through an *input* relationship as $allInputs(z_i) = \{z_i\} \cup \bigcup_{z \in inputs(z_i)} allInputs(z)$.

Based on these definitions, we can prune the set of potential error candidates for a certain test case to the set of all predecessors of the variables involved in the constraints representing a test case.

Definition 10 (Relevant Constraints and Irrelevant Constraints for a test case) Let $\langle CSP\ \langle X, D, C_{ok} \cup C_{pf}\rangle, TC^+, TC^-\rangle$ be a CSP diagnosis problem, tc^+ a test case from TC^+ and Z the cells of the spreadsheet program P from which the CSP was derived. $RelevantConstraints(tc^+)$ is defined as the constraints of the union including all input cells appearing in each of the constraints of tc^+ without those constraints that are assumed to be correct, i.e. $RelevantConstraints(tc^+) = Constraints(\bigcup_{c \in tc^+} allInputs(cell(c, Z))) \setminus C_{ok}$. The set $IrrelevantConstraints(tc^+)$ is defined as $(C_{ok} \cup C_{pf}) \setminus RelevantConstraints(tc^+)$.

Definition 11 (Test-case pruned CDP) Let $\langle CSP\ \langle X, D, C_{ok} \cup C_{pf}\rangle, TC^+, TC^-\rangle$ be a CSP diagnosis problem and tc^+ a test case from TC^+. A *Test-Case Pruned CSP Diagnosis Problem* ($TCPDP$) for a CDP and tc^+ is defined as the tuple $\langle CSP\langle X, D, C'_{ok} \cup C'_{pf}\rangle, tc^+, TC^-\rangle$, where the irrelevant constraints are considered to be correct, i.e. $C'_{ok} = C_{ok} \cup IrrelevantConstraints(tc^+)$ and $C'_{pf} = C_{pf} \setminus C'_{ok}$.

[13] There also might be a wrong formula in D1 but this one will be independent of the other error and can only be identified through another test case.

In other words, given a certain test case, we consider all cells appearing "after" the cells mentioned in the test case to be correct and ignore them in the diagnosis process, which leads to a corresponding reduction of the search space. In Section 4, we will show how this reduction directly leads to a decrease in the required running times in an experimental evaluation without missing any of the diagnoses of the original problem.

Proposition 2 *Iff $S \subseteq C_{pf}$ is a minimal diagnosis for the CDP $\langle CSP\langle X, D, C_{ok} \cup C_{pf}\rangle, \{tc^+\}, TC^-\rangle$, S is also a minimal diagnosis for the pruned diagnosis problem TCPDP $\langle CSP\langle X, D, C'_{ok} \cup C'_{pf}\rangle, tc^+, TC^-\rangle$ with $C'_{ok} = C_{ok} \cup IrrelevantConstraints(tc^+)$ and $C'_{pf} = C_{pf} \setminus C'_{ok}$.*

Proof (sketch): In a spreadsheet program, formulas represent deterministic functions without circular dependencies and the value of a cell z_i is fully determined by the values of the constants and formulas in the cells of the set $allInputs(z_i)$. Cells that are not part of $allInputs(z_i)$ therefore cannot affect the value of z_i. If the observed value in a cell z_i of a spreadsheet deviates from the expected value, only a constant or formula in $allInputs(z_i)$ can be the cause.

According to our translation scheme, for every cell z_i with a formula in a spreadsheet program, exactly one CSP variable x_i and one corresponding constraint $c \in C$ with semantics that are equivalent to the formula in z_i is created. A test case tc^+ consists of a number of constraints on expected values for some variables in X. Each expected value for a variable $x_i \in X$ is however strictly determined by the constraints derived for the input cells of the corresponding spreadsheet formula $z_i \in Z$, i.e. $allInputs(cell(x_i, Z))$. The set of possible constraints that influence the expected values of the test case is thus the union of the inputs for the variables mentioned in tc^+, i.e., $\bigcup_{c \in tc^+} allInputs(cell(c, Z))$.

Any minimal diagnosis S for the $CDP\ \langle CSP\langle X, D, C_{ok} \cup C_{pf}\rangle, \{tc^+\}, TC^-\rangle$ can therefore only contain elements appearing in $\bigcup_{c \in tc^+} allInputs(cell(c, Z))$, which is the set of $RelevantCells$ according to Definition 10. According to Definition 11, the test-case pruned CDP restricts the set of error candidates to the set of relevant cells of the original CDP and thus no diagnosis can be missed. If, alternatively, there were additional constraints in S which are not part of the relevant cells for tc^+, they would be redundant and the diagnosis S thus not minimal. □

In our problem formalization, we made two basic assumptions that should be kept in mind. First, we assume that the test cases themselves are not contradictory and correctly reflect the desired calculations, i.e., our approach does not yet deal with "oracle errors" and unreliable inputs by the spreadsheet developer.

The other aspect is that we so far considered the values of input cells to be correct, assuming that all inputs are part of the test cases. However, there might also be cells with constant values in a spreadsheet that are independent from specific test cases, e.g., general parameters. Our MDB-approach can be easily modified to consider such parameters as possible sources of errors by adding such potentially faulty input values as unary constraints to the set C_{pf} of the CSP to be diagnosed[14].

[14] The set of input cells that should be considered by the debugging to be potentially faulty could be determined interactively by the spreadsheet developer.

3 Parallelized hitting set construction

In the previous section we have shown how the MBD-based debugging process can be focused with the help of test cases and assertions and how we can exploit the directedness of the constraints to narrow down the set of error candidates. Next, we will show how the capabilities of modern multi-threaded CPU architectures – these are nowadays standard for office computers and even tablet computers – can help us to further reduce the required computation times.

3.1 Example for hitting set construction

The most costly operation during the search for diagnoses is the search for an arbitrary solution for a CSP during the construction of the hitting set graph (HS-DAG). Therefore, we propose a parallelized version of the breadth-first HS-DAG construction strategy, in which multiple nodes at the same level are expanded in parallel in separate execution threads.

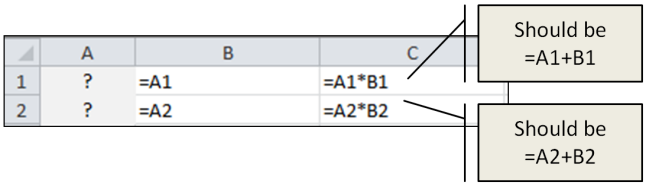

Fig. 3 A spreadsheet with two faults.

Figure 3 shows another faulty spreadsheet which we will use to illustrate the general idea. This time, the developer has probably misunderstood the underlying problem and formulated two multiplications instead of additions in the cells C1 and C2. The formulas in B1 and B2 look simple in this case, but they could be (faulty) references to other worksheets.

Let us assume that the spreadsheet developer tries to verify his solution using the following test case (tc_1): $(tc_1) : A1 = 3, A2 = 3, C1 = 6, C2 = 6$. The test case will obviously fail as the faulty program will output 9 instead of 6 in C1 and C2.

The search for diagnoses can be done efficiently based on the concept of conflicts as described in Section 2.2.1. In the context of diagnosing CSPs with test cases, a conflict can in analogy to Definition 4 be defined as follows.

Definition 12 (CSP Conflict) Given a CSP diagnosis problem $\langle CSP\langle X, D, C_{ok} \cup C_{pf}\rangle, TC^+, TC^-\rangle$ a conflict is a subset $F \subseteq C_{pf}$ for which there exists a test case $tc \in TC^+$ such that there exists no solution to the $CSP(X, D, C_{ok} \cup F \cup tc+)$. A conflict F is said to be minimal, if there exists no other conflict F' for $\langle CSP\langle X, D, C_{ok} \cup C_{pf}\rangle, TC^+, TC^-\rangle$ such that $F' \subset F$.

Finding a conflict within a set of constraints can be accomplished in an efficient way by using a conflict detection technique such as QuickXplain [35]. In our example there are two minimal conflicts sets (CS), i.e. sets of formulas that lead to inconsistencies between the test case and what is calculated by the spreadsheet

program, $CS_1 = \{B1, C1\}, CS_2 = \{B2, C2\}$. In other words, both CS_1 and CS_2 alone would lead to a contradiction with the test case and thus obviously make the CSP unsolvable.

The hitting sets of the two minimal conflicts CS_1 and CS_2 and therefore the set of diagnosis candidates correspondingly are $\{B1, B2\}$, $\{B1, C2\}$, $\{C1, B2\}$, and $\{C1, C2\}$.

Figure 4 shows a snapshot of the conflict-directed HS-DAG construction process. The numbers in the circles illustrate the sequence in which the DAG is constructed in a breadth-first manner. In the snapshot shown in the figure, the algorithm is about to explore the nodes marked with 6 and 7.

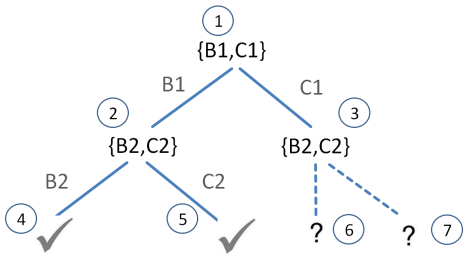

Fig. 4 Construction of the HS-DAG.

In the HS-DAG, each node in the tree is labeled with a conflict and outgoing edges are labeled with the elements of the conflict. The process starts by checking all positive test cases. If not all test cases pass, we create a root node ① and label it with an arbitrary conflict. At each new node, we then check if there is a solution for a relaxed CSP from which we remove all constraints that are used as path labels from the root to the current node. At the node labeled with ②, for example, we check if the set of constraints {B1}, which is the only label on the path so far, represents a diagnosis. This is done by searching a solution for the CSP from which we have removed B1 as we consider it to be faulty. If a solution is found, we know that {B1} is a diagnosis. If not, we search for a conflict in the remaining constraints and label the node with this conflict.

3.2 Parallelization approach

The most CPU-intensive part during the construction of the HS-DAG is the search for solutions of the underlying CSPs. The search for an arbitrary solution to various relaxed versions of the original CSP is required in two situations. First, when a new node is created, we need to check if we can find a solution for each test case. Furthermore, if no solution exists, a number of solution searches are usually required inside the conflict detection algorithm[15].

In order to better utilize the capabilities of modern CPUs and since the calculations at the different nodes of one tree level can in principle be done independently

[15] A discussion of the computational complexity of QuickXplain can be found in [35].

of each other, we propose an approach in which the calculations for the nodes at one level are done in parallel in different execution threads.

Reiter's hitting set algorithm includes a number of techniques to make the search process more efficient, including conflict reuse or tree pruning. In our parallelization approach, our goal is to retain these efficiency-enhancing techniques. Therefore, we propose a conservative strategy in which we limit the parallel computations to one level at a time and add a synchronization and pruning phase once all nodes of one level have been constructed. In the example in Figure 4, the node set {②,③} as well as the node set {④,⑤,⑥,⑦} would be processed in parallel.

As HS-trees can become broad and the search process requires no I/O operations, we suggest to use thread pools of limited size n, with n to be chosen based on the number of actually existing physical CPU cores and hardware threads. This way at most n nodes are processed at the same time, so that each calculation thread will utilize most of the available CPU capacity.

```
Input: A CDP consisting of a faulty CSP and testCases, a threadPool
Result: A set of minimal diagnoses
foundDiagnoses = ∅;
rootNode = calculateRootNode(CSP, testCases);
nodesForNextLevel = determineNodesToExpand({rootNode});
while nodesForNextLevel.size() > 0 do
    newNodes = ∅;
    for node in nodesForNextLevel do
        threadPool.execute(expand(node, newNodes));
    end
    threadPool.await();
    nodesForNextLevel = determineNodesToExpand(newNodes);
    pruneRedundantNodes(nodesForNextLevel);
end
return foundDiagnoses;
```

Algorithm 1: Sketch of HS-DAG parallelization

The main structure of the computation scheme, which works for general CSPs, is sketched in Algorithm 1. After the root node (and the corresponding conflict) have been determined, the procedure *determineNodesToExpand* is used to create a list of nodes to expand on the next level (Algorithm 2). As a parameter, the procedure takes the nodes of the last level and determines the next-level elements based on the conflicts associated with each node. Since for the root node the path label is still empty, the first call of *determineNodesToExpand* could use a slightly simpler method as the one shown in Algorithm 2. Back in the main function, the *while* loop contains the calculations for each subsequent level to be explored and continues until there are no more nodes to be expanded[16]. At each search level, the nodes to expand are queued for execution in the thread pool. The method *expand* itself works in the same way as in the non-parallelized version and is shown in Algorithm 3.

When expanding a node, a corresponding CSP for this node is created by removing those constraints from the problem which are on the edge labels from the root node to this node as they are considered to be faulty. The resulting

[16] Usual depth limitations could be applied here.

CSP is then analyzed with the QuickXplain algorithm, which either returns an empty set if all test cases are consistent with the relaxed problem formulation or a minimal conflict otherwise. To support the removal of constraints the call to the QuickXplain algorithm takes two arguments. The first is the complete set of positive test cases to check with all their constraints. The second parameter is a set of constraints to ignore while searching for a conflict. These are the constraints that are considered to be faulty and therefore should be removed from every test case. After all threads of the current level were executed (*await()*), the set of nodes to expand on the next level is determined.

```
Input: A set of nodes
Result: A set of new nodes to expand in the next level
nodesToExpand = ∅;
foreach node ∈ nodes do
    Let node.pathLabel be the set of conflicts that are used as path labels from
    rootNode to node and node.conflict the conflict found for this node;
    foreach constraint ∈ node.conflict do
        newPathLabel = node.pathLabel ∪ {constraint};
        nodesToExpand = nodesToExpand ∪ {new Node(newPathLabel)};
    end
end
return nodesToExpand;
```

Algorithm 2: determineNodesToExpand(nodes)

```
Input: A node to expand, a set of newNodes
Result: An extended set of newNodes or a new diagnosis in foundDiagnoses
Let node.examples be the set of examples associated to node and node.pathLabel the
set of conflicts that are used as path labels from rootNode to node;
node.conflict = QuickXplain(node.examples, node.pathLabel);
if node.conflict ≠ ∅ then
    newNodes = newNodes ∪ {node};
else
    foundDiagnoses = foundDiagnoses ∪ {node.pathLabel};
end
```

Algorithm 3: expand(node, newNodes)

```
Input: A set of nodes to expand in the next level
Result: A pruned set of nodes to expand in the next level
Let allPathLabels be the set of all path labels currently in nodesForNextLevel;
foreach node ∈ nodesForNextLevel do
    if node.pathLabel ∈ allPathLabels then
        nodesForNextLevel = nodesForNextLevel \ {node};
    end
end
```

Algorithm 4: pruneRedundantNodes(nodesForNextLevel)

Before entering the next level, the set of nodes to expand is examined for redundant elements (Algorithm 4). A node is redundant when it has the same set

of elements in the path label as another node to expand. In the sequential version of the algorithm, this is avoided by immediately closing nodes which have such a characteristic, see [49]. The other performance-enhancing techniques from [26] and [49], conflict reuse and tree-pruning can be applied without modifications also in this parallelized version[17]. For conflict reuse, it only has to be ensured that the global list of known conflicts is accessed in a thread-safe mode[18].

Overall, the proposed performance-enhancing technique was inspired by the fact that modern office computers – on which spreadsheet programs usually run – typically have CPUs with multiple cores and hardware threads. The proposed HS-DAG parallelization approach is however not limited in its applicability to spreadsheets and not even to CSPs and thus in our view represents generalizable algorithmic contribution of our work.

To the best of our knowledge, no comparable proposal has been put forward by the MBD research community so far. Over the years, various possible improvements to Reiter's scheme have been proposed to speed up the reasoning process. The proposals range form early works on using fault probabilities [36] over to recent approaches based on heuristics or on problem pre-compilation and re-coding [18, 6, 40]. Some of these methods are however incomplete or can only return one single diagnosis very quickly. Beside these optimizations for centralized diagnoses, recent works have discussed challenges in the area of *distributed* diagnoses [58, 57]. These works are however focusing more on knowledge representation aspects and theoretical problem properties than on parallelized computations.

4 Performance evaluation

We have evaluated our constraint-based spreadsheet debugging approach in two ways. In the following section, we will describe the results of a detailed performance analysis, in which we measured to which amount the algorithmic improvements proposed in this paper can speed up the diagnostic reasoning process. The results of an additional user study in which the participants interacted with an integrated debugging tool are presented later on in Section 5.

For the performance analysis, we selected a number of artificial and real-world spreadsheets in which we manually injected faults. As a measure, we used the CPU times that were required to calculate all diagnoses up to a given cardinality. Since all the compared algorithms are sound and complete, the sets of minimal diagnoses returned by the different algorithms for each problem setting are identical.

To measure the effects of the parallelization, we implemented two variants of the HS-DAG procedure, a single-threaded and a parallelized version. Both implementations share the same code base for solution search and conflict detection and the only difference lies in the way the nodes are expanded[19]. As an underlying constraint solver, we used the open source Java library Choco[20].

[17] We omitted these calls in Algorithm 1 for better readability.

[18] Some redundant calculations of conflicts can however in theory occur depending on how the threads are scheduled.

[19] We also made experiments in which we limited the thread-pool size to 1, thereby imitating a non-parallelized approach, leading to similar results. The additional overheads related to the creation of threads appear to be neglectable.

[20] `http://www.emn.fr/z-info/choco-solver/`

4.1 Experiments with artificial spreadsheets

In order to be able to compare our results with those from literature and to experiment with different problem sizes in a systematic way, we have re-constructed the parameterizable evaluation setting from [32] as shown in Figure 5. The spreadsheet represents a typical sales calculation, in which the input values are incrementally aggregated. To measure the efficiency for different problem sizes, the number of "products" (lines in the spreadsheet) can be increased which leads to an increase in inputs, problem variables and constraints in the underlying CSP. Figure 6 shows the characteristics of the CSPs that were derived from the different variations of the benchmark problem.

We used an evaluation procedure which is similar to that of [32]. In particular, we used a set of spreadsheet mutation operators [4] (e.g., exchanging a plus by a multiplication symbol) and mutated one or two of the spreadsheet formulas. For the first of the two scenarios, the mutation was done for cell `M13` as shown in Figure 5 for the following reasons. The formula in `M13` is fairly complex and has many dependencies, but is not an output cell so that the dependency analysis can prune a part of the succeeding formulas. Again, like in [32] we created a single positive test case for each problem to be diagnosed. Each test case was processed 100 times for each problem size to eliminate outside influences to the running times. In addition to the process in [32], we shuffled the order of the constraints before the diagnosis process in each iteration because QuickXplain's running times depend on the position of the conflict within the set of constraints.

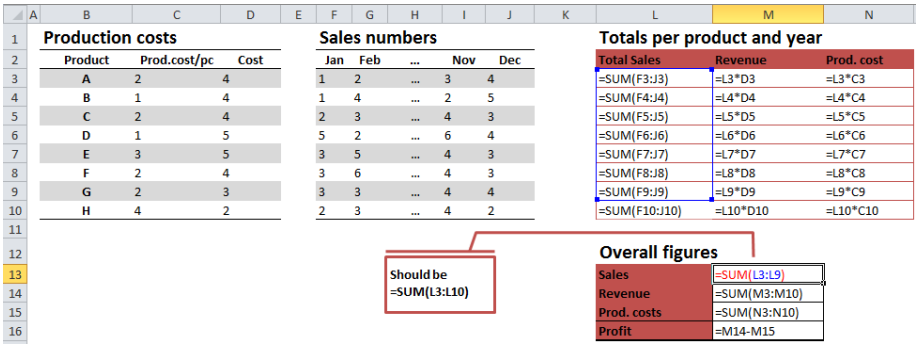

Fig. 5 The parameterizable benchmark problem from [32]. The example shows a typical range error in cell M13, which might have been introduced as new lines were added to the spreadsheet. A similar error was reported for the often-cited study by Reinhart and Rogoff [30].

In Figure 7 and Figure 8, we report the average running times for two test scenarios using a current desktop computer[21] as well as the number of required constraint propagations and solution searches. Similar to [32], the first scenario contained one single fault. In the second scenario, two faults were injected. We furthermore varied the number of products in order to assess the scalability of the method. The largest spreadsheet in this test has 50 products and contains over 150 formulas. Figure 8 shows the numbers for the double-fault case.

[21] Intel Core i7-3770K, 3.5 GHz, 16GB RAM, 8 hardware threads.

#Products	#Vars	#Constraints
2	38	10
4	72	16
6	89	22
8	140	28
10	174	34
20	344	64
30	514	94
50	854	154

Fig. 6 Characteristics of the resulting CSPs for the benchmark problem in Figure 5.

In order to test the performance gains achieved by the analysis of the formula dependencies and the effects of parallelization, we ran the experiments in four different configurations consisting of the sequential or parallel approach each using the dependency analysis or not. Given the completeness of the MBD procedures and the given test cases, all injected errors were found in each problem setting. The number of variables and constraints for each problem setting are similar but not identical to those reported in [32], which is caused by a slightly different problem encoding. Also, the number of required propagations and solution searches is different, caused, e.g., by effects of randomly generating test cases and the shuffling mentioned earlier.

The detailed results of the analysis are shown in Figure 7 and Figure 8. In the right-most column we report the overall improvement that can be achieved through the two optimization techniques proposed in this paper, which corresponds to the difference between the shaded columns. What is shown is thus the relative improvement from the single-threaded method without dependency analysis – the approach from [32] – to the parallelized method that uses dependency-based pruning. Our observations can be summarized as follows.

- *Applicability of standard MBD-based debugging:* Already the basic MBD debugging approach is tractable for small to medium sized problems. The longest running time for the single-fault case on average was around 1 second and 2 seconds for the double-fault case, which we see as a realistic time frame for interactive debugging, in particular as the depth of the search tree was not limited in these experiments.
- *Effects of parallelization:* The performance gain obtained from parallelization depends on the complexity of the diagnosis problem and ranges from "no improvement" for the very tiny problems, which can be fully diagnosed in a couple of milliseconds, up to 30% for the most complex problem with 50 products and 150 constraints in the double-fault case. The results thus confirm that the parallelization of the diagnostic process for computationally intensive problems can help to significantly reduce the required running times.
- *Effects of dependency analysis:* An even larger effect can be observed when input dependencies between formulas are taken into account. Comparing the running times for the sequential versions with and without the dependency analysis, we can observe an improvement that ranges from 50 to 60 percent. Again, the relative improvements are higher for the more complex problems.
- *Overall improvement:* The numbers in the right-most column show that – again depending on the problem size – the required running times can be reduced by

more than 70%. For the most complex test case, we could for example decrease the running time from over 2.2 seconds to about 630 milliseconds.

Single fault case without dependency analysis					Single fault case with dependency analysis				
#Prods	#CSP prop	#CSP solved	Time(ms)		#CSP prop	#CSP solved	Time(ms)		Reduction in %
			Sequent.	Parallel			Sequent.	Parallel	
2	14	9	5	4	6	5	1	2	60%
4	25	17	9	8	12	11	4	3	67%
6	36	26	16	15	18	17	7	6	63%
8	47	34	25	24	24	23	11	10	60%
10	58	43	38	34	30	29	17	15	61%
20	111	85	136	123	60	59	59	52	62%
30	166	127	319	288	90	89	131	111	65%
50	274	211	1,000	890	150	149	379	341	66%

Fig. 7 Results for the single-fault case.

Double fault case without dependency analysis					Double fault case with dependency analysis				
#Prods	#CSP prop	#CSP solved	Time(ms)		#CSP prop	#CSP solved	Time(ms)		Reduction in %
			Sequent.	Parallel			Sequent.	Parallel	
2	26	16	7	7	12	9	3	3	57%
4	62	41	21	16	32	26	10	8	62%
6	86	59	38	27	46	39	18	13	66%
8	109	77	58	41	59	52	27	20	66%
10	131	94	83	60	72	64	39	28	66%
20	242	180	298	214	135	126	134	100	66%
30	356	266	700	501	197	187	293	222	68%
50	573	435	2,273	1,603	320	309	857	628	72%

Fig. 8 Results for the double-fault case.

In general, while the dependency analysis method is specific to the spreadsheet domain, the parallelization of the hitting set construction process can be seen as a generalizable contribution of our work. In order to demonstrate that the proposed parallelization can be beneficial also for general problems, we conducted a series of experiments on typical CSP benchmark problems. The results of this analysis are presented in Appendix A. Again, measurable performance improvements can be observed for these benchmark problems. At the same time, these experiments gave us further insights with respect to the choice of the size of the thread pool. In the following experiments, we therefore only used 4 of the 8 available parallel execution threads. This setting in our view is advantageous as it leads to high performance and at the same time ensures that there is enough CPU capacity left for the other tasks running on the computer of the spreadsheet user.

4.2 Experiments with real-world spreadsheets

Beside the parameterizable benchmark problem, we have evaluated our approach with a number of further real-world spreadsheets of different sizes and complexities,

where some of the spreadsheets are publicly available, some were taken from the EUSES corpus and some are from our own research institution. The spreadsheets included in this evaluation and their specifics are described in Table 2.

Selection of spreadsheet examples. In contrast, e.g., to [2], where the documents were sampled randomly from the EUSES corpus, we selected the documents in a manual process. Our goal was to experiment with a number of real-world spreadsheets that cover a range of typical applications and are not too similar in their structure. In addition, our current MBD-framework for spreadsheet debugging supports only a subset of the functionality of MS Excel including basic arithmetic, minimum and maximum operators, IF-statements, as well as logical operators. Therefore, we only considered spreadsheets that contained only such formulas and adapted some of the formulas accordingly. The general structure and the cell references which basically determine the complexity of the diagnosis process, were however left unchanged. Finally, in our current experiments so far, we restricted the examples to integer values. Although the presented approach is in principle applicable for real-valued numbers, we did not conduct detailed experiments so far due to the limitations of the used open source solver engine with respect to real number arithmetic.

Overall, despite these restrictions, the chosen document corpus shall help us to demonstrate that our MBD-approach is applicable for many classes of typical spreadsheets of small to medium size in different application domains.

Evaluation procedure, results and outlook. Regarding the evaluation procedure, we injected individual errors in these spreadsheets, created one artificial positive test case and measured the required CPU time in milliseconds. The results of this evaluation in which we used 4 execution threads, dependency-based pruning and no search-depth limitations are shown in Figure 9. More details about the individual test cases are given in Table 2.

Table 2: Notes on the additional experiments of Figure 9. .

ID	Comment
1, 2	These two tests are based on a typical sales forecasting spreadsheet, see `http://www.score.org/sites/default/files/Sales_Forecast_1yr_0.xls`. The spreadsheet has several hundred cells and over 140 formulas. We evaluated two scenarios, in which we varied the number and position of errors. In both scenarios, the diagnostic process took less than half a second. Notice however, that diagnosing the double fault case (2) was faster than the test case where only one fault was injected (1) even though there were also more candidates. The reason is that we additionally varied the position of the injected fault in the spreadsheet. In the single-fault case, the mutation was injected at an output cell, whereas in the other case two intermediate cells were changed so that the effects of dependency-based pruning were stronger. We include this example to illustrate that the number of errors (and the corresponding depth of the search tree) is only one among several parameters that can influence the diagnosis time.

3, 4, 5	These tests were made on different subsets of a real-world, comparably complex course planning and reporting spreadsheet from the university. For Cases (3) and (4) we injected the same error into two excerpts of different sizes of the spreadsheet. Case (4) shows that even when there are more than 500 variables and 400 constraints, the calculation of the diagnoses was done in less than a second. In general, comparing Cases (3) and (4) illustrates how the running times depend on the size of the spreadsheet. Case (5) is based on the same excerpt as Case (4), but this time we injected two errors, both of them at output cells, which are more or less dependent on all other cells in the spreadsheet. This pathological position of the errors combined with the fact that the spreadsheet is full of IF-THEN formulas, led to a huge number of possible candidates (over 3,000) and, correspondingly, running times which are beyond what is suitable for interactive settings (over 16 seconds). We include this measurement as another example to indicate that the running times strongly depend on the position of the errors, the general size of the spreadsheet, and the nature of the formulas. Perspectives on how to deal with spreadsheets, in which the number of candidates is huge, are discussed briefly in Section 5.3.
6	This test was made on another real-world department-internal spreadsheet, this time for project cost estimation. A range error was injected as in Figure 5 so that the sum of project costs only covered 3 of 4 project years and the faulty formula correspondingly was only dependent on 3/4 of the other formulas. As a result, the possible diagnoses were found within less than 100 milliseconds.
7, 8, 9	Cases (7) to (9) were based on an adapted and simplified[22] version of the spreadsheet at `http://discovery.ucl.ac.uk/14128/` for the estimation of life time preservation costs for digital objects. The particularity of these test cases is that the spreadsheet is comparably complex and contains more than 700 cells with formulas and that at the same time the formula complexity is comparably high in terms of number of inputs per formula. We used these test cases to measure the effect of the required search depth and correspondingly constructed cases with 1 to 3 errors. Therefore, in case (9), the HS-DAG has to be expanded to the third level before a diagnosis is found. The required running times of less than 4 seconds with the search depth limited to 3 levels are still acceptable in our view. Diagnoses of higher cardinality where, e.g., four or more formulas explain an unexpected observation, might also be too hard to understand and thus of limited value for the end user.

[22] We did for example only use the main worksheet, adapted functions in formulas that our translation process does not support yet, but kept the general structure.

10	Case (10) is a real-world spreadsheet for wage calculation from `http://ogs.ny.gov/bu/dc/forms/Docs/Excel/bdc63.xlsx`. While the spreadsheet is comparably small, we injected the error on the worst possible place at the final sum calculation, meaning that nearly no dependency-based pruning can be applied and all formulas can contribute to the problem. Still, the measurement shows that the diagnoses could be determined in less than a second. At the same time, while the dependencies between the formulas required an increased calculation effort, they helped us to narrow down the set of candidates to two formulas in this case.
11, 12	These test cases were based on a hospital cost form from a health-care company, see `http://www.tmhp.com/TMHP_File_Library/HealthIT/TXMedicaidHospitalPaymentCalcWksheet.xls`. The difference between the test cases is again the position of the injected error. In Case (11), the error was injected in the "middle" of the spreadsheet, whereas in Case (12), the error was injected close to the final sum. In both cases, the running times were acceptable for interactive debugging and comparable to the Cases (6) and (10), which have similar size but partially use different operators in the formulas, which induce different computational complexity.
13, 14, 15	These are parts of spreadsheets taken from the EUSES corpus. (13) and (14) are typical calculations (`financial/03-3xx-en.xls`, `financial/02rise.xls`). In each case, we took one of the worksheets for our experiments. We injected a triple-fault – all off-by-one range errors – into (13) and a single-fault into (14). Test case (15) finally is an excerpt of a huge spreadsheet from the molecular biology domain from Filby's [21] corpus (`filby/CHARGED.xls`). The particularity of the spreadsheet is that it nearly only consists of IF-statements.

The absolute running times can depend – as mentioned in Table 2 – on a number of factors, including the size and constrainedness of the underlying CSP, the type and complexity of the operators in the formulas, the domain sizes of the variables, the position of the error in the spreadsheet or the number of the diagnoses. Some of these aspects are directly related to "phase-transition" phenomena that appear in general CSPs and that depend, e.g., on the density and tightness of constraints [52] or simply on how effective the constraint propagators are implemented in the underlying solver.

The reported running times in Figure 9 should thus be interpreted as general indicators of the applicability of our proposed method and different running times can be obtained when some of the above mentioned parameters change.

We have conducted a number of additional experiments to measure some of these effects. We report a few anecdotal examples in the following paragraphs. An in-depth analysis of the individual effects is part of our future work.

Effect of domain sizes: We have varied the domain sizes for the financial calculation in test case (14), a financial calculation from the EUSES corpus, in which we injected a triple-fault for this experiment. The required running time with a

ID	Application domain	#Vars	#Cts	#Injected Faults	#Diags.	Time (ms)
1	Sales forecasting	367	143	1	89	390
2	Sales forecasting	367	143	2	144	234
3	Course planning 1	205	163	1	20	126
4	Course planning 2	583	457	1	63	768
5	Course planning 3	583	457	2	3024	16518
6	Project calculation	132	64	1	24	69
7	Lifetime Cost estimation	803	701	1	22	241
8	Lifetime Cost estimation	803	701	2	22	1037
9	Lifetime Cost estimation	803	701	3	22	3930
10	Wage calculation	123	17	1	2	865
11	Hospital cost form	76	38	1	13	726
12	Hospital cost form	76	38	1	15	1414
13	Revenue calculation (EUSES)	154	110	3	200	243
14	Financial calculation (EUSES)	220	112	1	54	131
15	Molecular biology (EUSES)	112	210	1	53	633

Fig. 9 Results for a number of other mutated real-world spreadsheets.

default domain size of 0 to 100,000 was 285ms[23]. If we however restrict the domain sizes to 0 to 10,000, the running times increase to about 580ms. A further (unnecessary) reduction to domain sizes that were smaller than 1,000 caused the running times to go up to about 3 seconds. Note that these changes did not affect the solution space, i.e., in all settings the same set of diagnoses was returned. The only difference was that it became harder and harder for the constraint solver to find arbitrary solutions during the diagnostic process because the number of solution was drastically reduced when the domain sizes were reduced. Restricting domain sizes can thus represent a trade-off. On the one hand, restricting the possible values for a cell can in some cases help to focus the diagnosis process as shown in the examples in Section 2. On the other hand, it can lead to increased computation times.

Effects of the position of the error: For this experiment, we again used a variant of the parameterizable problem setting from Figure 5. We injected one double-fault into it, but varied which formulas were affected. In one case, the sum formulas for sales and production costs were faulty (`T19, T21`) and the required time to find the 7 possible diagnoses was around 45ms. If the affected cells were however `T19` and `T20` (the revenue calculation), the running times required to identify 16 diagnoses went up to about 250ms. This effect is of course to some extent expected, as the revenue calculation depends on a larger number of inputs than the production costs.

4.3 Discussion

Overall, we see our results of our experimental analysis with small and medium-sized spreadsheets as strong indicators that the required computation times for

[23] This is about twice the time needed for the single fault problem reported in Table 9.

the proposed constraint-based debugging approach are suitable for interactive debugging scenarios, in which the allowed response times are in the range of a few seconds.

Technically, the approach can also be applied for huge spreadsheets with thousands of cells and formulas. The required running times can then, however – again depending on the problem setting – exceed the time frames that are acceptable for interactive settings. In that context, we see a number of possible improvements that are part of our current and future work. Obviously, using an optimized commercial constraint solver will lead to some performance gain. There are, however, also a number of algorithmic extensions which we are investigating. In particular, we are currently examining if hierarchical decomposition approaches as done, e.g., in [13] or [19] can be applied to our spreadsheet debugging problem. Since spreadsheets often consist of logically connected blocks, a "natural" decomposition structure might be given. Furthermore, given that the constraints derived from the spreadsheet are directed and have a functional character, we are currently analyzing if the specific diagnostic procedures proposed in [55] for tree-structured systems and in [54] for functional programs can be applied or extended for our problem.

5 Toward interactive spreadsheet debugging - the Exquisite framework

The experimental analysis in the previous section was based on an offline experimental design in which existing spreadsheets were mutated and then analyzed. This form of experimental evaluation is common in the literature on automated spreadsheet testing and debugging and helped us to confirm the general feasibility of our constraint-based and model-based debugging approach.

We are, however, aware that more research is required on how an interactive and integrated debugging environment should be designed that is usable by typical or at least advanced spreadsheet developers. For example, the question arises how the debugging user interface should be designed, how complex such an interface may be and how the user interaction mechanisms should look like within the context of a spreadsheet environment. This is in particular important as our and also previous spreadsheet debugging approaches require new interaction types that do not exist so far in commercial tools. Both in the GoalDebug system [3] and our approach, the user may specify expected values for individual cells. Another example is the concept of "test cases", which are fundamental both in our work and in previous proposals for automation support in spreadsheet testing such as [1].

In our current and future work, we aim to address these research questions in more detail and have developed a first prototype of an integrated spreadsheet debugging framework called Exquisite. The detailed discussion of the framework, which is described in [33], goes beyond the scope of this paper. In order to paint a broader picture of the challenges of algorithmic spreadsheet debugging in practice, we will first give an overview of the design considerations of the framework and discuss possible extensions to our algorithms which may become possible when the user is actively involved in the process.

5.1 The Exquisite framework

5.1.1 Designing the end user view

The user interface component of the Exquisite framework is designed as an add-in to the wide-spread MS Excel spreadsheet environment. An annotated screenshot is shown in Figure 10, which visualizes some of the user interface elements with which the user can interact. The purpose of the screenshot and the following discussions is to exemplify the challenges of finding proper and easy-to-use interaction mechanisms that are understandable for a spreadsheet developer.

UI integration: While modern general-purpose integrated software development environments like Eclipse comprise comparably complex debug "perspectives", one of the main design goals was to stay within the visual environment that the user already knows. Therefore, when the user launches the debugging process by navigating to the Exquisite ribbon, the spreadsheet under inspection remains visible all the time. One first challenge in that context however is that the spreadsheet formulas cannot be edited as long as the debugging view is active. This is necessary because the cognitive load may get too high, if the user had to think about putting in test values for a specific test case and correcting formulas for all test cases at the same time. While this restriction is a standard behavior in usual debugging environments, it is something untypical during the spreadsheet development process in which there are usually no limitations regarding what and when the user can enter data into cells.

Visual indicators: To give the user visual feedback on being in the debug mode, we change the cell background colors in a way that input, output, and intermediate cells get different colors. Some types of errors like incomplete ranges can directly be spotted through this visualization.

Test case management: Through the menu, the user can create, store, load, modify or delete test cases. Defining a test case is done directly in the spreadsheet. The user can enter values into the input cells as these remain editable, observe the immediate effects and then interactively indicate if some values are unexpected or what an expected value in an output cell or an intermediate cell should be. The specification of expected values is thus done similar to the way it was done in [1]. A somewhat open question in that context is whether or not typical spreadsheet users are actually willing to invest time in the specification of test cases. While current spreadsheet environments do not support the explicit management of test cases, recent insights reported by Hermans in [27] suggest that there are quite many users who add additional test conditions into their spreadsheets.

Cell & formula information: In that area, the user can provide further information about allowed cell values. These specifications can either be global definitions (e.g., assertions) which have to hold in any case, or test case specific additional constraints as described in Section 2.1. Furthermore, the user can explicitly state the correctness of some of the formulas or individual values in the context of a test case. Both pieces of information can help to reduce the complexity of the diagnosis problem.

Completeness indicator: The more information about the spreadsheet under consideration is available, the easier it is to focus the diagnosis process. The speed gauges in the right lower corner shall help to motivate the user, for example, to

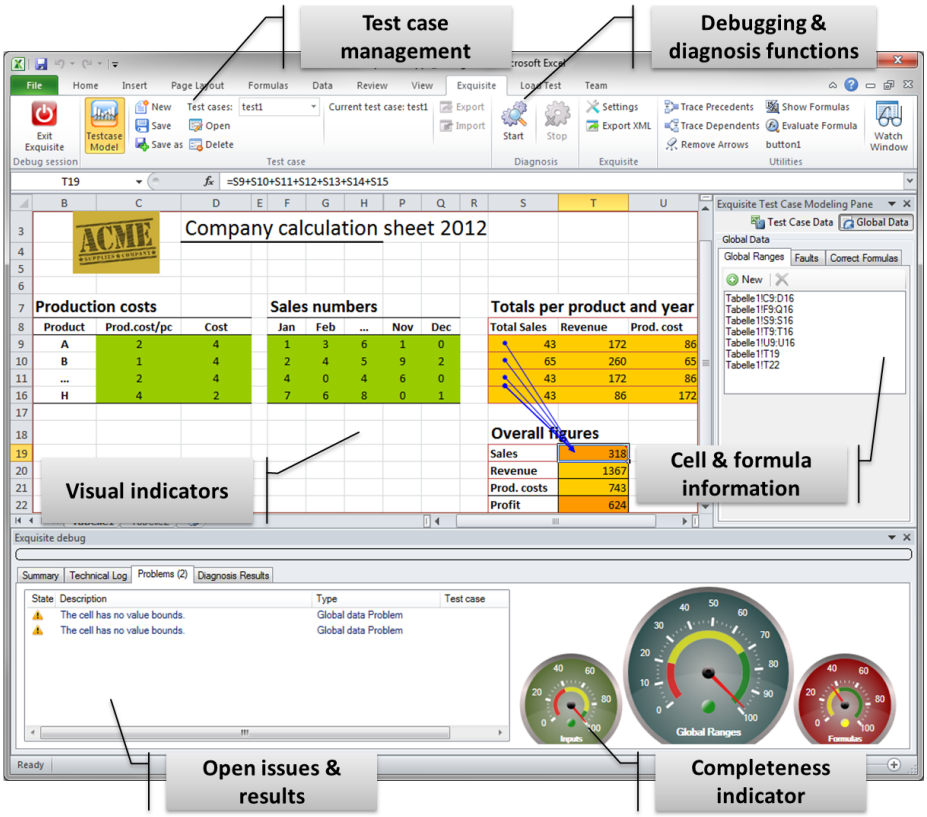

Fig. 10 Interacting with the EXQUISITE add-in.

explicitly specify – at least roughly – the range of possible values for the different cells or fill in values for all input cells of a test case.

Open issues & results area: This area is used to give the user various types of feedback and, for example, list suggestions to the user with respect to missing pieces of information. This area is also used to display the outcomes of the debugging process, i.e., the list of diagnosis candidates, which we currently rank based on their cardinality. When a user clicks on one of the diagnoses, its components are highlighted visually as shown in the figure.

5.1.2 Technical architecture of the framework

We call EXQUISITE a framework, because it consists of a number of modular components as described in detail in [33]. In particular, we have designed a client-server architecture and have correspondingly de-coupled the client-side add-in from the diagnosis module and the constraint reasoner as shown in Figure 11.

The communication between the client and server components is based on an XML protocol which allows us in principle also to use a different spreadsheet environment. As the diagnostic reasoning algorithm runs in its own process and is connected to the client via the TCP protocol, the computation-intensive diagno-

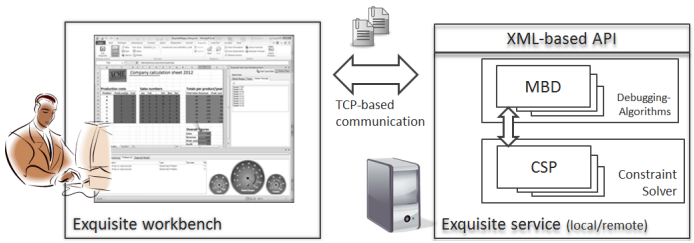

Fig. 11 Overview of the architecture of EXQUISITE.

sis process can also be run on a remote high-performance server. Regarding the constraint reasoner, we are currently using the open source Choco solver. This component could however also be exchanged by an optimized commercial library.

5.2 Experimental evaluation

In order to assess if the developed EXQUISITE tool is actually helping users during debugging in the sense of [44], we conducted a laboratory user study involving 24 subjects. To that purpose, we designed an error detection experiment, where users had to locate a fault within a spreadsheet. With our experiment, we primarily aim to answer the question if users are more efficient (faster) when locating the error with the help of EXQUISITE and if the users are capable to deal with the tool without in-depth training.

5.2.1 Design of the debugging exercise

Profit calculation

Free bananas

Sold	127
Free items	10

Product information

Product	Prod. cost	Price
Bananas	1	4
Apples	2	4
Cherries	2	3

Sales numbers

Jan	Feb	Mar	Apr	May	Jun
16	25	29	19	21	17
12	22	23	18	16	10
34	42	45	43	37	28

Totals per product

Revenue	Prod. costs	Sales
508	137	127
496	248	124
687	458	229

Tax calculation rules

Base limit	150
Add-on limit	400
Fees	5

Production costs should be below 800

The profit value should be 757

Overall figures

Sales	480
Revenue	1691
Tax	45
Prod. costs	843
Profit	803

Fig. 12 Spreadsheet used for the error detection exercise.

Design of the faulty spreadsheet. As a test case, we used a spreadsheet that is structurally similar to our benchmark spreadsheet shown in Figure 5. We extended

the spreadsheet slightly in order to be able to integrate different types of formula expressions. The resulting spreadsheet, which is slightly more complex than our benchmark problem, is shown in Figure 12[24]. The spreadsheet comprises 16 cells that contain formulas.

We injected one mechanical error – a duplicate cell reference – in one of the sales summations (cell `O9`). The task of the participants was to locate the error based on one defined test case. During the experiment, one group consisting of half of the participants used the EXQUISITE tool; the other group could use all the standard mechanisms of MS Excel. In each case, the data of the given test case were already inserted in the spreadsheet as shown in Figure 12. In addition, for both groups the information was provided that the calculated outcomes for the test case deviated from what was expected. These hints are shown as annotations in Figure 12 and include one expected value (the final outcome of the calculation in cell `N17`) and one range restriction (the value of cell `N16` is expected to be below some threshold).

Experimental procedure. Each subject completed the exercise individually using a pre-configured computer at our department. Depending on whether the EXQUISITE tool was used or not, different preparatory instructions were given to the participants by the experimenter. In order to avoid any differences among the subjects or potential biases introduced by the different experimenters in this phase, detailed scripts were prepared. Participants of both groups were given a detailed description of the intended semantics of the spreadsheet to be debugged (including the description of which outcomes are expected for the given test case). Also, participants with little or no knowledge of Excel formulas were briefly introduced to the general usage of formulas in Excel and the set of functions needed for the exercise.

The group using the EXQUISITE system was in addition given a tutorial document describing the usage of the tool. In the tutorial, an example was given on how to invoke the tool, specify a test case, start the diagnosis process and how to interpret the output. The participants could also actually step through the tutorial example using the EXQUISITE tool. The tutorial example was not related to the spreadsheet to be debugged in the subsequent experiment.

After these preparations, all participants were instructed to search for exactly one error in the given spreadsheet. The participants of the group that used the EXQUISITE tool had to start the debug add-in in Excel and enter the expected value for the profit (Cell `N17`) and the upper bound for the production costs (Cell `N16`) for the test case through the tool's user interface. After saving the test case, the participants could start the diagnosis process. The corresponding diagnosis candidates were then presented in the "Open issues & results" area of the tool as shown in Figure 10. The users could then click on the individual candidates and inspect the corresponding cells in the spreadsheet.

One experimenter was present during this phase, monitored the behavior of the subjects and took time measurements. The experimenter was allowed to provide assistance in case there were technical problems with the tool or when there were questions regarding the procedure. In case a participant could not correctly locate the error, the experiment was stopped after 15 minutes. After the error detection

[24] We conducted the experiment in German and all material used in the study was prepared in German. Here the English translation is shown.

Group	Tool usage	Error detection	Total time	Success rate
With EXQUISITE	66 (σ=31)	97 (σ=69)	163 (σ=78)	100%
Without EXQUISITE	-	547 (σ=269)	547 (σ=269)	67%

Table 3 Time measurements for the user study in seconds.

exercise, the participants filled out a questionnaire. The participants who did not use the tool during the experiment, were instructed to go through the EXQUISITE tutorial before they filled out the questionnaire.

5.2.2 Study participants

We recruited 24 students of our university for the study. All of them were Computer Science students. The average age of the participants was about 26, about 75% were male. While the participants might thus not be representative for the average population of (advanced) spreadsheet users, they are comparable with respect to their general IT education, which we think is important given the comparably small size of our preliminary study.

The participants were however quite different with respect to their self-reported expertise regarding spreadsheets. In our questionnaire, the participants had to report on a 1-to-5 scale[25] on (a) how often they use spreadsheets and (b) if they have ever used complex expressions in these spreadsheets (e.g., IF statements). Regarding the usage frequency, both the mean and the median value of the answers were about 3 and our sample contained both frequent spreadsheet users as well as users who only rarely use spreadsheets. When asked about the experience with complex formulas, the average value was even lower (2.1), the median value was 2, and 7 participants reported that they never had used such formulas at all.

5.2.3 Results of the debugging exercise

Quantitative analysis. As quantitative measures we will report the time needed to find the error after having read the instruction and intended semantics. Furthermore, we report if the participant was able to find the error within the given time frame at all[26]. The measurements are summarized in Table 3.

In the first two columns, we report how much time the participants on average needed to use and instrument the debugging tool, before they could focus on the inspection of the reported diagnoses and the involved cells. On average, the participants needed about 66 seconds to start the debugging environment, enter the expected values from the test case description, save the test case and run the diagnosis algorithm. The computation time needed by the system for the actual diagnosis task was below one second and neglectable. Once the diagnoses were returned – the system identified two single-fault diagnoses – the subjects on average needed 97 more seconds to find the problem. The overall time needed was therefore below 3 minutes. The group of participants that only relied on MS Excel

[25] The value 1 corresponds to "never"; 5 means "very often".

[26] We made additional measurements regarding the time needed by the subjects to study the description of the intended semantics or go through the EXQUISITE tutorial. A detailed analysis of possible correlations of these measurements with the error-detection performance is however beyond the scope of this paper.

on average needed more than 9 minutes (547 seconds) for the task, not counting those that did not complete the task. The observed differences are thus quite large and statistically significant ($p<0.05$).

Finding the true cause of the error was obviously much easier for the group using the EXQUISITE tool, because they had to examine a much smaller set of formulas provided that they had understood how to interpret the output of the tool correctly. Interestingly, the participants, which did *not* use the tool, exhibited quite different problem solving strategies even though they all had a similar IT education background[27]. Some participants started examining the spreadsheet from the result cell, where the error was observed; others started checking the formulas from the inputs. Some participants ruled out cells as the cause based on the formulas' dependencies. The times needed to find the error strongly varied among the participants. The fastest participant of this group, for example, needed only 2.5 minutes, which is even below the average of those who used the EXQUISITE tool.

Finally, since only 67% of the participants of the group without tool support could find the error within the specified limit, our study can be interpreted as an indicator regarding the *effectiveness* of our method, i.e., users do not only find the errors faster, but they can assumedly even find more errors contained in a spreadsheet.

Questionnaire results and other observations. After the debugging exercise, the participants filled out a 9-item questionnaire. The detailed questions are listed in Appendix B. The questionnaire covered different aspects related to (a) the general usefulness of tool support, (b) the role of test cases and (c) an assessment of the usability of the EXQUISITE tool. For all questions a 1-to-5 rating scale was used where 5 means "completely agree/yes" and 1 means "completely disagree/no".

Given that the goal of the user study was to mainly provide a first assessment of the general usability and value of the tool, we will only summarize a few key observations here.

- *Tool support for debugging:* The average feedback on questions in this block was very high and around 4.5 for the first three questions for both groups. The participants thus stated that (i) better tool support for fault localization is desirable, (ii) that the EXQUISITE tool is helpful and (iii) that they would be able to find errors faster with the tool. Interestingly, the "willingness" to use the tool again (question item (iv)) was on average even higher for the participants who had not used the tool during the experiment (4.7 vs. 4.2 on average). The observed differences were however only modestly significant ($p < 0.1$).
- *The role of test cases:* Regarding the questions on the usage of test cases for fault localization, we could not observe statistically significant differences between the different groups. The participants stated that they (i) considered test cases can be helpful for fault localization in general and that (ii) the EXQUISITE tool can be helpful in that respect. The rating feedback for both questions was at about 4.4 on average. The self-reported willingness to define additional test cases to find errors was also relatively high and about 4.1 on average. Again, the group that had not used the tool was slightly more willing

[27] We encouraged the participants to comment on their strategies during the experiment in the sense of a think-aloud protocol.

than the group that has used the tool (4.3 vs. 4.0). The differences were however not statistically significant.
- *Tool and usability:* Both groups of participants stated that the integration of a debugging tool within the spreadsheet environment is important (4.4 on average). The usability of the EXQUISITE tool was finally rated to be high with an average rating of about 4.1 for both groups.

Summary and limitations. Overall, we see the results of our preliminary user study to be encouraging. In particular we can observe that the participants did not only consider the tool and our specific solution approach to be valuable, but also reported that they in general would appreciate better tool support and would actually be willing to invest in the definition of test cases. This is in some sense in line with the observations from Hermans [27], who reports on additional indications that users actually perform certain forms of test activities during spreadsheet development.

The reported findings have of course to be interpreted with care. While the participants had different expertise in the usage of spreadsheets, they were all Computer Science students and thus should have an increased awareness of the importance of systematic quality assurance and testing. At the same time, even though the self-reported spreadsheet expertise of most participants was comparably low, we assume that the participants are to some extent used to interact with more complex software applications. Therefore the findings regarding the usability of the tools might not fully generalize to different types of spreadsheet users and additional studies are thus required. Finally, we have so far used only one specific and comparably small spreadsheet for the user study.

5.3 Toward interactive spreadsheet debugging and incremental focusing

Our future work will focus on two aspects. First, we will further focus on the empirical evaluation of the user interface design based on studies with real users as described in the previous section. Overall, the number of such studies in the literature is however comparably low according to [46] and a number of specific methodological aspects have to be considered [8]. The EXQUISITE framework will represent a solid basis for such future studies and comprises a logging component with the help of which we can record all user interactions during a spreadsheet creation and debugging exercise. Overall, with the help of such studies we hope to be able to gain insights not only with respect to the user interface design but also more generally about typical errors made by users.

Our second aim is the design of new techniques and the application of existing algorithms for interactive debugging. One of the challenges in most application domains of model-based diagnosis is that the number of diagnosis candidates can be comparably large and overwhelms the end user. In our current implementation, we rank the diagnoses based on their cardinality and present diagnoses that involve fewer formulas first. Alternatively, one could also take the complexity of the formulas or external statistics about error type frequencies into account. In particular, we also plan to investigate to which extent the various types of spreadsheets smells – for example those recently proposed in [7, 16, 28] – can be used for diagnosis ranking.

Another approach to differentiate among a number of possible diagnoses, which was proposed already in the early years of model-based diagnosis and on which we want to focus in our future work, is to use additional "measurement points" [37]. The idea is to try to obtain additional information about the system's internal behavior given some inputs. Applied to our spreadsheet diagnosis problem, this would for example mean to explicitly ask the user for the expected value for some intermediate cell for some test case. With the help of such an additional observation, hopefully a number of possible causes of unexpected values at the output variables can be ruled out. This process of asking for additional values can be done in an incremental process, where the determination of a good measurement point in each step can for example be based on information-theoretic considerations [37]. We plan to include a similar approach in the EXQUISITE framework and will in particular evaluate if the recent technique interactive ontology debugging proposed in [51] can be applied or extended for the spreadsheet domain.

Finally, for debugging standard software artifacts, spectrum-based fault localization (SFL) was proposed to localize faulty code lines by examining their occurrence in failing test cases, see, e.g., [34] and [5]. Recently, Campos et al. in [11] have shown how such techniques can be smoothly integrated into the popular Eclipse IDE to support interactive debugging and regression testing. In our future work we will follow a similar line of research and investigate how such techniques can be integrated into our interactive spreadsheet debugging environment and, e.g., be used to refine the ranking of diagnosis candidates.

6 Relation to previous works

The work presented in this paper is related to a number of research fields including software testing and debugging, end-user programming, algorithmic and AI-based software engineering approaches, or model-based diagnosis algorithms and applications.

In fact, the literature on debugging of traditional software artifacts like procedural programs debugging alone is huge and comprises a number of popular techniques such as slicing, spectrum-based fault localization (SFL), algorithmic and genetic debugging, "code smells", or hybrids thereof. We assume that many of these techniques are also applicable for spreadsheet programs and could complement the current model- and constraint-based approach of EXQUISITE. Examples of recent works in that direction are the paper on spreadsheet smells by Hermans et al. [29] or the comparative analysis of different techniques for spreadsheet debugging by Hofer et al. [31]. In the following, however, we will focus our discussion on methods which were specifically designed for spreadsheet testing and debugging.

Regarding the general nature of the debugging approach, the work in this paper extends our own previous work reported in [32], in which we introduced the general idea of translating spreadsheet programs into CSPs and apply Reiter's MBD algorithm. In our current work, we have not only developed a precise problem characterization but also proposed novel techniques to increase the scalability of the approach and conducted a systematic offline evaluation of different aspects which revealed that our enhancements led to a significant scalability improvement when compared to the basic approach. In addition, we performed a user study to assess the value of our method for the end user.

A similar idea of encoding a spreadsheet to a CSP was proposed later on by Abreu et al. in [6]. In their work they however used a slightly different strategy to locate the errors by encoding the "abnormal" predicate into the constraint problem and not using a hitting-set algorithm. In a case study, they could show that small-scaled spreadsheet problems can be efficiently solved using the MINION constraint solver. Experiments with larger spreadsheets or specific enhancements for spreadsheets were however not reported in [6].

A constraint-based but slightly different system for spreadsheet debugging called GoalDebug was proposed by Abraham et al. in [3]. GoalDebug is a user-oriented spreadsheet debugging method, which calculates a set of "repair" proposals for faulty spreadsheets [3] based on user-provided expected values for certain cells. The ranking of possible repairs that lead to the desired result is based on domain-specific heuristics. The technique has been implemented into a real spreadsheet environment. The evaluation was based on the mutation of real-world spreadsheets through the injection of artificial errors using spreadsheet-specific mutation operators, see also [4]. As evaluation metrics both the percentage of actually identified correct repairs as well as the ranking of the correct repair within a set of alternatives were used.

Similar to the above mentioned MBD-approaches for spreadsheets, GoalDebug focuses on a certain type of spreadsheet errors – errors in formulas[28] – and uses a constraint-based representation of the spreadsheet on which the internal inferences are based upon. The proposed change inference rules are however spreadsheet-specific and GoalDebug's effectiveness therefore depends on the quality of these rules.

Our work is similar to GoalDebug in that we aim to develop a system which is integrated into a real spreadsheet environment. This allowed us to conduct a first study with spreadsheet users. Furthermore, the offline experimental evaluation reported for the Exquisite framework follows a comparable experimental protocol for offline analysis based on automatically generated program mutants using spreadsheet specific operators. In contrast to GoalDebug, our framework however currently focuses on the error localization process. Clearly, the ultimate goal of the debugging process is to fix the problem. The investigation of MBD-based methods that also support the repair process as suggested for example in [23] or [53] is thus part of our future work.

Software *testing* and debugging are closely related. Spreadsheet-specific methods for testing have been proposed, e.g., in [1], [50], or [9]. These approaches for example include techniques for automatic test-case generation or a visual ("What you see is what you test") method for end users (WYSIWYT), which allows users to interactively test a spreadsheet in a fully visual way and gives the user feedback about the overall "testedness" of the program.

Test cases also play a crucial role in the model-based calculation of diagnoses in our work. While the focus of our framework is on error localization, the Exquisite framework discussed in Section 5 comprises corresponding tools for test case management. Similar to the WYSIWYT approach, our framework allows the user to specify that a certain value is correct. Instead of using this information to calculate definition-use associations, we use this information to narrow down the set of possible error candidates in a model-based approach. In contrast to the WYSIWYT

[28] Other types of errors could be semantic or structural errors.

method, the EXQUISITE framework also allows the user to explicitly state that a cell value, formula, or the spreadsheet given a certain test case is faulty.

With respect to automated test case generation as proposed in [1], our EXQUISITE framework does not comprise such a functionality so far. However, we believe that this will be a valuable functionality to include as one of our next steps to support the end user in the specification test cases in particular for large spreadsheets.

Regarding our future work on studies with real users, only few examples of such studies exist in the literature. The work by [25] represents an early example of an experimental study with real users. The goal of the study was to understand the role of domain expertise and spreadsheet expertise on the error-finding performance of spreadsheet users in terms of effectiveness and efficiency. The exercise consisted of letting users find manually-injected errors in spreadsheets and it turned out that domain experts usually find more errors than non-experts. Still, even the experts did not detect all errors. Similar code-inspection experiments also showed the limitations of manual error localization procedures, as reported for example in [41].

In this work we use Reiter's MBD algorithm to find diagnoses for the translated CSPs. Another approach to find a diagnosis to a CSP would be to search for a minimal relaxation for it (Max-CSP [22]). The Max-CSP problem is defined as finding a partial solution to a CSP which satisfies the maximum possible number of constraints. Those constraints that could not be satisfied can be interpreted as a diagnosis. An efficient algorithm for the Max-CSP problem is given in [38]. Although this approach can be used to find one minimal diagnosis, the basic technique does not support the identification of all minimal diagnoses. A further analysis of how such algorithms can be integrated in our our framework, for example to quickly locate one first diagnosis for complex problem settings, is part of our future work.

7 Summary

Spreadsheet programs are in widespread use in companies and in many cases, important business decisions are based on spreadsheet calculations. In this work, we have shown how model-based diagnosis techniques can be applied to help the developer detect the possible causes when a spreadsheet does not behave as expected.

Technically, we propose to translate the spreadsheet to a constraint-based representation to perform further reasoning, e.g., about input propagations or arithmetically impossible constellations. We have shown that the directedness of spreadsheet formulas can be utilized to further speed up and focus the diagnostic process. As an additional performance-enhancing contribution, we presented a parallelized algorithm scheme for the construction of the hitting-set DAG, which makes it possible to better utilize the capabilities of modern computer hardware. An experimental evaluation based on artificial and real-world examples using an open source constraint solver showed the general practicability of the method for realistic interactive debugging scenarios.

Our future work includes on the one hand additional laboratory studies with real users to validate the interaction design of a prototypical add-in for a popular

spreadsheet environment as well as the incorporation of techniques for a more fine-grained ranking of candidates and more interactive debugging strategies.

Acknowledgements

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

References

1. Abraham R, Erwig M (2006) AutoTest: A Tool for Automatic Test Case Generation in Spreadsheets. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2006), Brighton, United Kingdom, pp 43–50
2. Abraham R, Erwig M (2006) Inferring Templates from Spreadsheets. In: Proceedings of the 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, pp 182–191
3. Abraham R, Erwig M (2007) GoalDebug: A Spreadsheet Debugger for End Users. In: Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, pp 251–260
4. Abraham R, Erwig M (2009) Mutation Operators for Spreadsheets. IEEE Transactions on Software Engineering 35(1):94–108
5. Abreu R, Zoeteweij P, van Gemund AJC (2008) An Observation-based Model for Fault Localization. In: Proceedings of the 6th International Workshop on Dynamic Analysis (WODA 2008), New York, NY, USA, pp 64–70
6. Abreu R, Riboira A, Wotawa F (2012) Constraint-based Debugging of Spreadsheets. In: Proceedings of the 15th Ibero-American Conference on Software Engineering (CIbSE 2012), Buenos Aires, Argentina, pp 1–14
7. Asavametha A (2012) Detecting Bad Smells in Spreadsheets. Master's thesis, School of Electrical Engineering and Computer Science, Oregon State University, USA
8. Brown PS, Gould JD (1987) An Experimental Study of People Creating Spreadsheets. ACM Transactions on Information Systems 5(3):258–272
9. Burnett M, Sheretov A, Ren B, Rothermel G (2002) Testing Homogeneous Spreadsheet Grids with the "What You See Is What You Test" Methodology. IEEE Transactions on Software Engineering 28(6):576–594
10. Burnett M, Cook C, Pendse O, Rothermel G, Summet J, Wallace C (2003) End-User Software Engineering with Assertions in the Spreadsheet Paradigm. In: Proceedings of the 25th International Conference on Software Engineering (ICSE 2003), Portland, Oregon, pp 93–103
11. Campos J, Riboira A, Perez A, Abreu R (2012) Gzoltar: An eclipse plug-in for testing and debugging. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012), Essen, Germany, pp 378–381

12. Chadwick D, Knight B, Rajalingham K (2001) Quality Control in Spreadsheets: A Visual Approach using Color Codings to Reduce Errors in Formulae. Software Quality Control 9(2):133–143
13. Chittaro L, Ranon R (2004) Hierarchical Model-Based Diagnosis Based on Structural Abstraction. Artificial Intelligence 155(1–2):147–182
14. Console L, Friedrich G, Dupré DT (1993) Model-Based Diagnosis Meets Error Diagnosis in Logic Programs. In: Proceedings of the 13th International Joiunt Conference on Artificial Intelligence (IJCAI 1993), Chambéry, France, pp 1494–1499
15. Creeth R (1985) Micro-Computer Spreadsheets: Their Uses and Abuses. Journal of Accountancy 159(6):90–93
16. Cunha J, Fernandes JaP, Ribeiro H, Saraiva Ja (2012) Towards a Catalog of Spreadsheet Smells. In: Proceedings of the 12th International Conference on Computational Science and Its Applications (ICCSA 2012), Salvador de Bahia, Brazil, pp 202–216
17. Ditlea S (1987) Spreadsheets can be hazardous to your health. Personal Computing 11(1):60–69
18. Felfernig A, Schubert M (2010) FastDiag: A Diagnosis Algorithm for Inconsistent Constraint Sets. In: Proceedings of the 21st International Workshop on the Principles of Diagnosis (DX 2010), Portland, OR, USA, pp 31–38
19. Felfernig A, Friedrich G, Jannach D, Stumptner M, Zanker M (2001) Hierarchical Diagnosis of Large Configurator Knowledge Bases. In: Proceedings of the Joint German/Austrian Conference on Artificial Intelligence (KI 2001), Vienna, Austria, pp 185–197
20. Felfernig A, Friedrich G, Jannach D, Stumptner M (2004) Consistency-based diagnosis of configuration knowledge bases. Artificial Intelligence 152(2):213–234
21. Filby G (ed) (1998) Spreadsheets in Science and Engineering. Springer
22. Freuder EC, Wallace RJ, Heffernan R (1992) Partial Constraint Satisfaction. Artificial Intelligence 58(1-3):278–283
23. Friedrich G, Nejdl W (1992) Choosing Observations and Actions in Model-Based Diagnosis/Repair Systems. In: Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR 1992), Cambridge, MA, USA, pp 489–498
24. Friedrich G, Stumptner M, Wotawa F (1999) Model-Based Diagnosis of Hardware Designs. Artificial Intelligence 111(1-2):3–39
25. Galletta DF, Abraham D, Louadi ME, Lekse W, Pollalis YA, Sampler JL (1993) An empirical study of spreadsheet error-finding performance. Accounting, Management and Information Technologies 3(2):79–95
26. Greiner R, Smith BA, Wilkerson RW (1989) A Correction to the Algorithm in Reiter's Theory of Diagnosis. Artificial Intelligence 41(1):79–88
27. Hermans F (2013) Improving Spreadsheet Test Practices. In: Proceedings of the 23rd Annual International Conference on Computer Science and Software Engineering (CASCON 2013), Markham, Ontario, Canada, pp 56–69
28. Hermans F, Pinzger M, van Deursen A (2012) Detecting and Visualizing Inter-Worksheet Smells in Spreadsheets. In: Proceedings of the 34th International Conference on Software Engineering (ICSE 2012), Zurich, Switzerland, pp 441–451

29. Hermans F, Pinzger M, van Deursen A (2012) Detecting Code Smells in Spreadsheet Formulas. In: Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM 2012), Riva del Garda, Trento, Italy, pp 409–418
30. Herndon T, Ash M, Pollin R (2013) Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff. Working Paper 322, Political Economy Research Institute, University of Massachusetts, Amherst
31. Hofer B, Riboira A, Wotawa F, Abreu R, Getzner E (2013) On the Empirical Evaluation of Fault Localization Techniques for Spreadsheets. In: Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE 2013), Rome, Italy, pp 68–82
32. Jannach D, Engler U (2010) Toward model-based debugging of spreadsheet programs. In: Proceedings of the 9th Joint Conference on Knowledge-Based Software Engineering (JCKBSE 2010), Kaunas, Lithuania, pp 252–264
33. Jannach D, Baharloo A, Williamson D (2013) Toward an integrated framework for declarative and interactive spreadsheet debugging. In: Procedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2013), Angers, France, pp 117–124
34. Jones JA, Harrold MJ, Stasko J (2002) Visualization of Test Information to Assist Fault Localization. In: Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), Orlando, FL, USA, pp 467–477
35. Junker U (2004) QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems. In: Proceedings of the 19th National Conference on Artificial Intelligence (AAAI 2004), San Jose, CA, USA, pp 167–172
36. de Kleer J (1990) Using Crude Probability Estimates to Guide Diagnosis. Artificial Intelligence 45(3):381–391
37. de Kleer J, Williams BC (1987) Diagnosing Multiple Faults. Artificial Intelligence 32(1):97–130
38. Larrosa J, Meseguer P, Schiex T (1999) Maintaining reversible DAC for Max-CSP. Artificial Intelligence 107(1):149–163
39. Mateis C, Stumptner M, Wieland D, Wotawa F (2000) Model-Based Debugging of Java Programs. In: Proceedings of the Fourth International Workshop on Automated Debugging (AADEBUG 2000), Munich, Germany
40. Metodi A, Stern R, Kalech M, Codish M (2012) Compiling Model-Based Diagnosis to Boolean Satisfaction. In: Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI 2012), Toronto, Canada
41. Panko RR (1998) What We Know About Spreadsheet Errors. Journal of End User Computing 10(2):15–21
42. Panko RR, Halverson RP (1996) Spreadsheets on Trial: A Survey of Research on Spreadsheet Risks. In: Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS 1996), Wailea, HI, USA, pp 326–335
43. Panko RR, Port DN (2012) End User Computing: The Dark Matter (and Dark Energy) of Corporate IT. In: Proceedings of the 45th Hawaii International Conference on System Sciences (HICSS 2012), Wailea, HI, USA, pp 4603–4612
44. Parnin C, Orso A (2011) Are Automated Debugging Techniques Actually Helping Programmers? In: Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA 2011), Toronto, Canada, pp 199–209

45. Pemberton J, Robson A (2000) Spreadsheets in business. Industrial Management & Data Systems 100(8):379–388
46. Powell SG, Baker KR, Lawson B (2008) A critical review of the literature on spreadsheet errors. Decision Support Systems 46(1):128–138
47. Reichwein J, Rothermel G, Burnett M (1999) Slicing Spreadsheets: An Integrated Methodology for Spreadsheet Testing and Debugging. In: Proceedings of the 2nd Conference on Domain-Specific Languages (DSL 1999), Austin, Texas, pp 25–38
48. Reinhart CM, Rogoff KS (2010) Growth in a Time of Debt. American Economic Review 100(2):573–578
49. Reiter R (1987) A Theory of Diagnosis from First Principles. Artificial Intelligence 32(1):57–95
50. Rothermel G, Li L, Dupuis C, Burnett M (1998) What You See Is What You Test: A Methodology for Testing Form-Based Visual programs. In: Proceedings of the 20th International Conference on Software Engineering (ICSE 1998), Kyoto, Japan, pp 198–207
51. Shchekotykhin K, Friedrich G, Fleiss P, Rodler P (2012) Interactive ontology debugging: Two query strategies for efficient fault localization. Journal of Web Semantics 12-13:788–103
52. Smith BM (1994) Locating the Phase Transition in Binary Constraint Satisfaction Problems. Artificial Intelligence 81:155–181
53. Stumptner M, Wotawa F (1996) Model-Based Program Debugging and Repair. In: Proceedings of the 9th International Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE 1996), Fukuoka, Japan, pp 155–160
54. Stumptner M, Wotawa F (1999) Debugging Functional Programs. In: Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI 1999), Stockholm, Sweden, pp 1074–1079
55. Stumptner M, Wotawa F (2001) Diagnosing Tree-Structured Systems. Artificial Intelligence 127(1):1–29
56. Tsang E (1993) Foundations of Constraint Satisfaction. Academic Press
57. Wotawa F, Pill I (2013) On Classification and Modeling Issues in Distributed Model-Based Diagnosis. AI Communications 26(1):133–143
58. Wotawa F, Weber J (2010) Challenges of Distributed Model-Based Diagnosis. In: Trends in Applied Intelligent Systems, Springer Lecture Notes in Computer Science, pp 711–720

Appendix

A Performance measurements for regular CSPs

The parallelization approach described in Section 3 is not limited to spreadsheet debugging problems but can also be applied to general CSPs. In order to demonstrate the generalizability of the approach we have conducted a series of experiments to measure the performance gains that can be obtained for a number of benchmark problems taken from the 2008 CSP Solver competition[29].

[29] Available at `http://www.cril.univ-artois.fr/CPAI08/`

A.1 Problem selection and experimental procedure

When selecting CSP instances from the solver competition, we only included solvable instances and picked constraint problems that could be solved in relatively short time (i.e., below 2 seconds), since the diagnosis process can require a number of solution searches and we had to repeat all experiments several times to factor out random effects.

For each tested problem instance, we first generated a number of random solutions. From each solution, we randomly picked a fraction of the variables (e.g. 10%) and stored their values. These variable/value-combinations thus represent our partial test cases to be used by both algorithm variants. Next, we manually inserted errors (mutations) in the constraint problem formulations, e.g., by changing a $\leq$ operator to a $<$ operator, which corresponds to a mutation-based approach as proposed for example in [4].

The different scenarios are summarized in Table 13, in which we give details about the constraint problem, the number and characteristics of diagnoses induced by the test cases as well as information about the number of corresponding test cases. "TC fail" is the number of test cases which were inconsistent with the mutated CSP. "TC ok" is the number of non-failing partial test cases.

Scenario	#Cts	#Var	#Diags	AvgDiagSize	Search depth	#TC ok	#TC fail
costasArray-13	87	88	2	2.5	3	17	3
graph2	2245	400	72	3	3	2	3
graceful--K3-P2	60	15	117	2.9	3	12	8
e0ddr1-10-by-5-1	265	50	33	3	3	1	2
e0ddr1-10-by-5-8	265	50	210	3	3	2	3
bibd-10-30-9-3-2_glb	1435	1650	8	4	4	0	3
mknap-1-5	6	39	4	1.75	3	0	5

Fig. 13 General CSP diagnosis scenarios.

To measure the performance gains for parallelization, we used the absolute running times ("wall time") to compute all diagnoses up to a certain search depth. The experiments were repeated 10 times and average running times were reported. Experiments with more runs did not lead to different results.

A.2 Effects of Parallelization

The results of our first experiment using the scenario costasArray-13 shows how the computation time depends on the number of parallel threads. For these experiments, we used a laptop computer with an Intel Core i7-3610QM processor with 8 hardware threads.

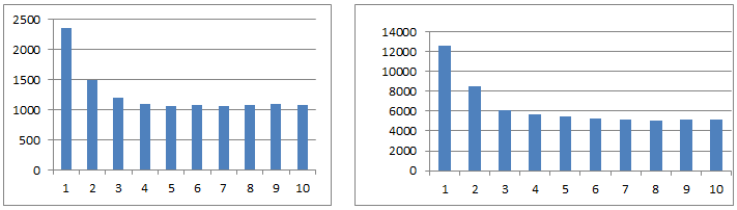

Fig. 14 Results for scenario CostasArray-13. Running times on the Y-axis are given in milliseconds. X-axis: Number of parallel threads.

Figure 14 shows running times for the CostasArray-13 scenario with a varying number of threads. On the left hand part of the figure, we used the original CSP instance and observed

that the running times strongly decrease for this problem setting when more than one thread is used. After about 4 to 5 threads, however, no further improvement is obtained and performance slightly degrades when the number of threads is higher than the number of CPU threads. Overall, the running times can be decreased by more than 50% from 2.36 seconds to about 1 second. Since on average it only takes a few milliseconds to find a solution for this problem, we made another experiment shown on the right hand side of the figure. In this experiment, we artificially extended the search time by keeping the CPU busy in a no-operation loop for ten milliseconds before actually starting search. The numbers show that the same trend can be observed also when the solution search lasts longer. The achieved improvement here was even at about 60%, which might be caused by the fact that the additional thread management is neglectable when compared with the search time in this case.

The relative performance improvements for the other analyzed scenarios from Figure 13 using 6 parallel threads are given in Figure 15 below. Note that some of them were comparably hard, with running times above one minute to compute the diagnoses. A detailed analysis of the relation of problem characteristics and the observed run-time improvements is part of our current work. Obviously, however, the size of the conflicts and thus the breadth of the search tree can be considered as one of the most relevant factors.

Scenario	Performance gain
costasArray-13	54.9%
graph2	50.9%
graceful--K3-P2	33.1%
e0ddr1-10-by-5-1	26.9%
e0ddr1-10-by-5-8	80.2%
bibd-10-30-9-3-2_glb	30.1%
mknap-1-5	46.1%

Fig. 15 Experiment results for general CSPs.

B Detailed questionnaire items used in user study

The post-experiment questionnaire to be filled out by the participants of the user study consisted of the following items. The questions and answers are translated into English; the original questionnaire was in German. A 5-point rating scale was used for all questions ranging from "completely agree/yes" (5) to "completely disagree/no" (1).

On debugging tools

- Do you think that better tool support for fault localization is on principle desirable for spreadsheets?
- Do you think that the Exquisite tool is helpful for fault localization?
- Do you think that faults can be located faster with this tool support?
- Would you use the tool (again) to find an error in a spreadsheet application?

On testing

- Do you think that relying on explicit test cases is advantageous for the detection of faults?
- Do you think that the usage of test cases through the Exquisite tool is helpful for the detection of faults?
- Would you be willing to define test cases, if you can detect faults with them?

Tool assessment

- Do you think it is important that an error-detection tool is embedded within a system like MS Excel?
- Was it clear to you how to use the *Exquisite* tool?