Knowledge-Based Sowtftware Engineering
... (Eds.)
Publisher, 2010

Toward model-based debugging of spreadsheet programs

Dietmar JANNACH [a,1] and Ulrich ENGLER [b]
[a] *Department of Computer Science, TU Dortmund, Germany*
[b] *Department of Computer Science, TU Dortmund, Germany*

Abstract.
Spreadsheet programs are widely used in industrial practice. As they are often developed not by IT professionals but by end users, particular attention has to be paid to quality control and testing and appropriate methods for fault prevention, fault localization and repair for spreadsheet programs have to be developed. In this paper, we propose to apply model-based diagnosis techniques for the systematic localization of faults in spreadsheet programs. Departing from the idea that the relevant parts of a spreadsheet program, i.e., the formulas, can be transformed to a Constraint Satisfaction Problem, our method uses an extended Hitting-Set algorithm and user-specified or historical test cases and assertions to calculate possible error causes. The proposed method can be used in combination with previous, often heuristics-based methods for interactive testing and repair. First experiments on typical spreadsheets for financial calculations show the general applicability of our approach which extends the scope of AI-based software debugging methods to spreadsheet programs.

Keywords. Spreadsheet programs, debugging, model-based diagnosis

1. Introduction

Spreadsheets are the most successful example of the End-User Programming paradigm. Today, millions of people [24] are using software like Microsoft Excel as a programming tool for a variety of purposes. In particular, spreadsheets are often used for calculation and forecast tasks related to business- or financial aspects of an organization. Like any other software, spreadsheets programs may however contain errors, which in this context can easily lead to wrong business decisions that cost companies huge amounts of money [22]. As end-user programs, spreadsheets are usually not developed by IT professionals, which means that modern Software Engineering practices that aim to minimize the risk of undetected errors are often not applied. Since the mid-1980s, different proposals have therefore been made that aim to introduce better quality control procedures and algorithms into the spreadsheet development process. The set of proposals ranges from process-related aspects such as guidelines and defined development procedures, to

[1] Corresponding Author: Dietmar Jannach, TU Dortmund, Germany, dietmar.jannach@tu-dortmund.de

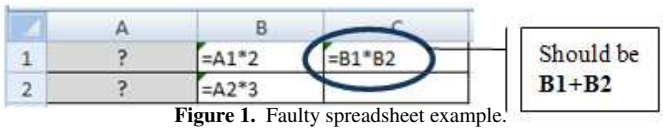

Figure 1. Faulty spreadsheet example.

systematic testing and test case generation, automated spreadsheet generation or the detection of particular types of problems such as type or unit errors; other techniques include the visualization debugging of potentially problematic errors or the generation of "repair" proposals in interactive debugging scenarios ([1],[3],[4],[6],[7],[8],[10], [21]).

One limitation of some existing approaches lies in the fact that they have to rely on heuristics to identify potentially problematic areas of spreadsheets, which are for instance based on fault-type probabilities. In this paper, we therefore propose to apply Model-based Diagnosis (MBD) techniques ([12], [23]) as a systematic method for fault localization in spreadsheets. MBD techniques were originally developed for the error localization in physical devices or electronic circuits; they have however also been successfully applied for debugging of software artifacts such as logical programs, VHDL designs, knowledge-based systems and Java programs ([11],[14],[19],[26]). The debugging process in our approach is based on sets of user-specified or historical test cases as well as additionally provided assertions entered by the end user. The outcome of the automatic analysis process is a set of diagnoses that point the end user to potentially problematic cells/formulas of the spreadsheet.

The paper is organized as follows. After giving an illustrating example in the next section we formally define the spreadsheet diagnosis problem based on the Constraint Satisfaction formalism and describe our diagnosis algorithm. Next, we discuss our prototype implementation and present first experimental results. An outlook on further steps and improvements in particular with respect to scalability are part of the final section.

2. Example

In classical MBD settings ([12],[23]), we are given a set of components (e.g., of an electronic circuit) including a characterization of their normal expected behavior as well as a description of how the components are interconnected. If the system does not behave as expected and the observed output of the system deviates from the predicted one, the diagnosis problem consists of finding those components which, when assumed to be faulty or behaving abnormally, explain this behavior.

This idea is transferred to the spreadsheet debugging problem in this work with the difference that instead of physical devices, the diagnosable components in our setting are the formulas in the spreadsheet, which are the most important sources of errors in spreadsheets [7] (as opposed to structural or semantic errors).

Consider the following simple spreadsheet in Figure 1 for integer calculations, where the input fields are $A1$ and $A2$, $B1$ and $B2$ are intermediate cells and $C1$ is the "output" field, which is not referenced by any other formula. Let us assume that the designer made a mistake and eventually used a multiplication in cell C1 instead of the intended plus symbol.

The first test case provided by the end user is $\{A1 = 1, A2 = 6, C1 = 20\}$), assuming that $C1$ should be properly calculated as $C1 = (1 * 2) + (6 * 3)$. The faulty spreadsheet will however output 36 by calculating (1*2) * (6*3) as the value of $C1$.

ID	Input A1	Input A2	Expected output C1	Observed output C1	Possible diagnoses
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. Using multiple examples to narrow down set of candidates.

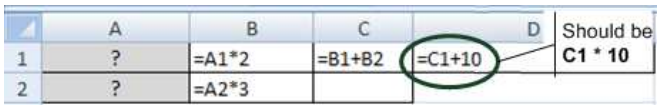

Figure 2. Using assertions to focus the diagnosis process.

Model-based diagnosis algorithms often work by identifying (minimal) conflicts in the system, i.e., minimal subsets of the full set of components which cause a problem. In our case, when we propagate the values of the test case through the spreadsheet, we can identify exactly one minimal conflict $\{B2, C1\}$. In other words, independent of whether or not the formula in $B1$ is correct, $B2$ and $C1$ cannot be correct at the same time as 6 is not a factor of 20, if we assume that only integer values are allowed in the cells. Note that we do not include the user input in A2 in the conflict as we assume the data from the test case to be correct. The corresponding minimal diagnoses are therefore either "B2 is incorrect" or "C1 is incorrect". Of course, supersets of these diagnoses would also explain the faulty behavior, but generally we are only interested in minimal diagnoses.

As an extension to the basic MBD scheme, we propose to extend the basic Hitting Set algorithm [23] in a way that multiple, simultaneous test cases can be utilized as to reduce the number of possible diagnosis [14]. Consider the set of four failing test cases in Table 1 based on which our proposed diagnosis algorithm is able to detect that only the formula in cell $C1$ can be faulty and "B2 is incorrect" is not a diagnosis. Table 1 shows the set of diagnoses for each test case. As it can be seen, only the diagnosis $\{C1\}$ explains the unexpected output for all examples. For instance, the diagnosis $\{B2\}$ does not explain the failure of the last test case.

In addition to basic input/output pairs (positive test cases), we also propose to allow the user to specify negative test cases, i.e., test cases that are supposed to fail. Furthermore, our approach also supports the usage of additional assertions as proposed for spreadsheets in [5] in order to narrow down the number of possible diagnoses.

Consider the extended example in Figure 2 where in cell $D1$ the value of $C1$ should be multiplied by 10 but unfortunately a plus was entered by the user. If we are using an unsuccessful and unluckily chosen test case $\{A1 = 4, A2 = 5, D1 = 230\}$, the basic MBD procedure would identify 4 singleton diagnosis, which is not helpful for the end user as it means that every single formula could be faulty. However, when designing the spreadsheet, the end user might already have an idea of possible values for different cells. In our example, the user could already know and explicitly state that $C1$'s value has to be definitely greater than 100 and $D1$ less than 1000. Applying these constraints $(C1 > 100 \wedge D1 < 1000)$ to our example above, value propagation and the MBD algorithm can immediately determine the only correct diagnosis $\{D1\}$.

3. Constraint-based diagnosis of spreadsheets

3.1. The spreadsheet diagnosis problem

In our work, the relevant parts, i.e., the formulas of the cells of a given spreadsheet are translated into a Constraint Satisfaction Problem (CSP) [25], in which cells correspond to problem variables and cell formulas to constraints. Test cases can be defined as sets of value assignments to variables (or as unary equality constraints). Since test cases can, on principle, be complete or incomplete with respect to the set of input and output fields, the end user is not forced to specify values for every field in a large spreadsheet when debugging it in the development phase. In the testing phase, complete sets of input and output pairs may however be often available, e.g., in the form of sales numbers from previous quarters.

Assertions can be expressed as additional constraints of the CSP. A negative example, that is, a test case that is supposed to fail, can be expressed as a negated conjunct of unary constraints on input/output variables, which is added to the constraint network.

More formally, let a spreadsheet program SPR be defined as a tuple (IN, F, OUT) consisting of

- IN, a set of (initially empty) input cells whose values are to be specified by the end user;
- F, the set of cells of SPR containing formulas (functions) specifying how the cell value is to be calculated;
- OUT, the set of cells with formulas of SPR that are not referenced by other formulas of SPR, i.e., OUT represents the result or output cells.

Technically, the differentiation between F and OUT cells is not necessary as the formulas of both types of cells are transformed to constraints in the same way. We leave this distinction in the tradition of testing approaches that are based in input/output pairs. Note however that in our approach we are not limited to input/output pairs as test cases may also contain values for intermediate cells.

Given a spreadsheet $SPR \quad (IN, F, OUT)$, a Spreadsheet Diagnosis Problem (SDP) can be defined as a tuple $(SCSP(V, C, Dom), E^+, E^-, ASSRT)$, where

- $SCSP(V, C, Dom)$ is a Constraint Satisfaction Problem derived from a spreadsheet program as follows: V is a set of variables corresponding to spreadsheet cells in IN, OUT, and F; C is the set of constraints derived from the formulas in F and OUT; Dom as usual describes the domains for the variables in V;
- E^+ and E^- are sets of (positive and negative) test cases expressed as conjunctions of unary constraints on variables of $SCSP$;
- $ASSRT$ is a set of additional constraints specifying further assertions on allowed variable assignments.

A diagnosis D for a spreadsheet diagnosis problem $(SCSP(V, C, Dom), E^+, E^-, ASSRT)$ is a subset of the constraints in C such that there exists an extension EX, where EX is a set of constraints, such that

- a solution to the $CSP(V, C - D \cup ASSRT \cup EX \cup \{e^+\}, Dom)$ exists $\forall e^+ in E^+$
- no solution to the $CSP(V, C - D \cup ASSRT \cup EX \cup \{e^+\}, Dom)$ exists $\forall e^- in E^-$.

Informally speaking, we aim to find a set of faulty constraints (corresponding to spreadsheet formulas) which explain the discrepancy between the expected test case values and the observed one. In the example in Figure 1 we can see that the set $\{C1\}$ is a diagnosis, because if the constraint is removed from the CSP, all the positive test cases can be extended to a CSP solution. $\{B1\}$ in contrast is not a diagnosis because even when removed, the conflict of $\{B2, C1\}$ with the first test case will persist. Note that we are typically only interested in minimal diagnoses, i.e., sets of constraints of a spreadsheet diagnosis problem SDP for which there are no subsets which are also diagnoses.

A diagnosis D for the spreadsheet diagnosis problem $(SCSP(V, C, Dom), E^+, E^-, ASSRT)$ will always exist, if the positive and the negative examples do not contradict each other, i.e. *iff* $\forall e^+ in E^+ : e^+ \cup \bigwedge_{e^- \in E^-}(e-)$ is consistent. In the worst case D will contain all elements of C. In the following, we will refer to the conjunction of negated test cases $\bigwedge_{e^- \in E^-}(e-)$ as NTC.

Based on the above observation, diagnoses can be finally characterized without the explicit specification of EX (compare also [14]) and we can rewrite our diagnosis definition as follows.

D is a diagnosis for $(SCSP(V, C, Dom), E^+, E^-, ASSRT)$ *iff* $\forall e^+ \in E^+$, the $CSP(V, C - D \cup ASSRT \cup NTC \cup$ e+, $Dom)$ has at least one solution.

Regarding the representation of NTC, consider a set of negative test cases $E^- = \{(A1 = 1, A2 = 1, C1 = 1), (A1 = 1, A2 = 1, C1 = 3)\}$ given for the example in Figure 1. The constraints to be added to the CSP that represent the negative test cases would be: $\neg(A1 = 1 \wedge A2 = 1 \wedge C1 = 1) \wedge \neg(A1 = 1 \wedge A2 = 1 \wedge C1 = 3)$

3.2. Diagnosis algorithm

We propose to apply the conflict-based diagnosis algorithm from [14] which extends Reiter's Hitting Set algorithm [23] with the possibility to use multiple test cases to the Spreadsheet Diagnosis Problem. In addition, we use QuickXPlain [20] for fast computation of minimal conflicts.

In the context of spreadsheet diagnosis, a conflict in an $SDP(SCSP(V, C, Dom), E^+, E^-, ASSRT)$ is defined as a subset $X \subseteq C$ for which there exists at least one $e^+ \in E^+$ such that there exists no solution to the $CSP(V, X \cup \{e^+\} \cup ASSRT \cup NTC, Dom)$. A minimal conflict M for an SDP is a conflict where there exists no other conflict $M' \subset M$ which is also a conflict for the SDP.

In order to support a diagnosis process based on multiple test cases, we extend the original Hitting Set Directed Acyclic Graph (HS-DAG) labeling where nodes are labeled with conflicts and edges leading away are labeled with individual elements of the node's conflict. In our algorithm, every node is also labeled with a set $ST \subseteq E^+$, the subset of successful test cases at that node. Test cases that were found to be successful at some node in the tree do not have to be re-checked at a deeper tree level, because at deeper levels the problem is always further relaxed as additional constraints are removed.

In the algorithm below, we use the function $ST(preds(n))$ that returns the union of all positive examples of all predecessor nodes of n. The function $H(n)$ denotes the set of edge labels from the root node to node n (see [23]).

Figure 3 shows this additional labeling for the example spreadsheet diagnosis problem from Figure 1. For demonstration purposes, we will use the test cases 2, 3, and 4 from Table 1 in this example. As usual, the tree is constructed in breadth-first manner.

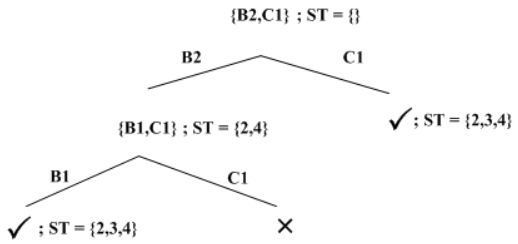

Figure 3. Hitting Set construction with multiple test cases.

The root node is labeled with the conflict $\{B2, C1\}$ returned by the theorem prover (a constraint solver and a conflict detection technique in our setting). The set of consistent examples at the root node is $ST = \{\}$. The root node is then expanded with two outgoing edges labeled $B2$ and $C1$. When $B2$ is assumed to be faulty, there still remains the conflict $\{B1, C1\}$ for test case 3. Thus, a new node $\{B1, C1\}$ is added. The set of successful test cases ST is $\{2, 4\}$, which means that test case 3 has not to be re-checked at a lower level of the tree. When on the other hand $C1$ is assumed to be faulty, no conflict remains and the node can be labeled with $\surd$. Expanding the node $\{B1, C1\}$ at the second level with the outgoing edge $B1$ corresponds to testing whether the edge labeling on the path from the root ($\{B2, B1\}$) is a diagnosis. As no conflict remains and all test cases were successful, $\{B2, B1\}$ is also a diagnosis. The other branch at the second level would be again labeled with $C1$. Since $C1$ was however already identified as a diagnosis at the

Algorithm 1 Spreadsheet diagnosis algorithm with support for multiple test cases.

In: An SDP $(SCSP(V, C, Dom), E^+, E^-, ASSRT)$,
searchdepth: maximum size of diagnoses to be retrieved.
Out: A set of diagnoses S.

(1) Use the Reiter's Hitting Set algorithm [23] to generate a pruned HS-DAG T for the collection F of minimal conflict sets for $(SCSP(V, C, Dom), E^+, E^-, ASSRT)$ in a breadth-first manner until the maximum search-depth is reached.

- During HS-DAG construction, at node n, the theorem prover call $TP(V, C - H(n) \cup ASSRT \cup E^+ - ST(preds(n)) \cup NTC)$ corresponds to a check, whether there exists an $e^+ \in E^+$ for which the $CSP(V, C - H(n) \cup ASSRT \cup \{e^+\} \cup NTC, Dom)$ has no solution.
  IF such an e^+ exists
  Use QuickXPlain to compute a minimal conflict C for one of the failing test cases in $E^+ - ST(preds(n))$;
  return C as result of the call to TP;
  ELSE
  return "ok" as the result of the call to TP;
- set $ST(n) = ST(preds(n)) \cup E_{CONS}$ where E_{CONS} is the set of all positive examples e^+ for which a CSP solution was found at node n.

(2) Return $\{H(n) |$ n is a node of T labeled by ok$\}$.)

	B	C	D	E	F	G	O	P	Q	R	S	T
1	**Sales numbers**									**Totals per product and year**		
2	Product	Prod. cost/pc	Cost Price	Jan	Feb	Mar	...	Nov	Dec	**Total Sales**	**Revenue**	**Prod. cost**
3	A	200	400	?	?	?	?	?	?	=SUM(E3:Q3)	=R3*D4	=R3*C3
4	B	150	420	?	?	?	?	?	?	=SUM(E4:Q4)	=R4*D4	=R4*C4
5	C	220	450	?	?	?	?	?	?	=SUM(E5:Q5)	=R5*D5	=R5*C5
6	D	100	500	?	?	?	?	?	?	=SUM(E6:Q6)	=R6*D6	=R6*C6
7	E	300	510	?	?	?	?	?	?	=SUM(E7:Q7)	=R7*D7	=R7*C7
8	F	200	400	?	?	?	?	?	?	=SUM(E8:Q8)	=R8*D8	=R8*C8
9	G	250	300	?	?	?	?	?	?	=SUM(E9:Q9)	=R9*D9	=R9*C9
10	H	400	250	?	?	?	?	?	?	=SUM(E10:Q10)	=R10*D10	=R10*C10
12										**Totals**		
13										Sales	=SUM(R3:R9)	
14										Revenue	=SUM(S3:S10)	
15										Production cost	=SUM(T3:T10)	
16										Profit	**=S14-S15**	

Figure 4. Parameterizable spreadsheet structure used for experimental evaluation, errors indicated.

first level, the node can be closed – indicated by the X in Figure 3 – because $\{B2, C1\}$ would be a superset of an existing diagnosis and thus not minimal. The algorithm schema is summarized in Algorithm 1.

4. Implementation and first evaluation

In order to evaluate the general practicability of our approach, a first Java-based prototype system has been developed. The system is capable of processing MS Excel documents using the POI library[2]. In an initial phase, a parser module transforms the spreadsheet instructions into the representation of the open source constraint library Choco[3]. At the moment, the parser is capable of translating a subset of the most common MS Excel arithmetic functions, constant numbers, as well as the IF-THEN-ELSE construct. Fields with text constants are currently not translated in the prototype as we do not support string operations in constraints yet; moreover, such constant fields are mostly used as labels for cells and are irrelevant for the calculations in most spreadsheets we analyzed.

Beside the adapted HS-DAG algorithm described above, our system – as mentioned – implements Junker's QuickXPlain algorithm [20] for the fast, on-demand detection of minimal conflicts that steer the diagnosis process. In our prototype system, the test cases and optional further assertions are stored in text files.

In order to evaluate our method with respect to running times, we used – among other manual experiments with different spreadsheets – a typical spreadsheet for financial calculations in which we inserted common errors. An example of such a spreadsheet which was used with variations in the subsequently described experiments is shown in Figure 4. In cell $S3$, a reference to the wrong cost field ($D4$ instead of $D3$) is made. Such errors can easily be caused by copying and pasting cell contents. Cell $S13$ contains a range error (should be "R3:R10"), which might be caused by a later addition of product E. In addition, we also made use of assertions, which are viewed as a valuable tool for end users in spreadsheet debugging in this experiment [5]. The assertions – which could

[2] http://poi.apache.org/

[3] http://www.emn.fr/x-info/choco-solver/

#Prods.	#Vars	#Constrs.	#CSP prop.	#CSP solved	Diag. time (ms)
2	40	16	18	10	920
3	55	19	23	13	1404
4	70	22	32	18	4555
5	85	25	31	18	6162
6	100	28	46	26	8814
7	115	31	49	28	15242
8	130	34	54	31	22761

Table 2. Experiment 1 measurements (single fault diagnosis).

be easily specified by an end user – were that the total revenue must be higher than the total production cost for each product.

In order to evaluate different problem sizes, we varied various parameters of the problem setting such as the number of cells and the number of formulas by adding more products. The experiment was also conducted with different numbers of errors. In Experiment 1 reported below, we for example limited the search to single-element diagnosis and thus included only one of the errors shown in Figure 4. In Experiment 2, we also looked for double faults[4]. All experiments were run with one single (positive) test case, i.e., we did not rely on the focusing effect of multiple test cases, negative test cases or lots of additional assertions. In real-world settings, the validated sales numbers of one of the previous years could simply be used. Also in practical settings, the end user might (interactively) specify additional knowledge about definitely correct formulas, which have not to be taken into account during the diagnostic process. Such knowledge can be easily into the Hitting Set algorithm and into QuickXPlain and thus help to further reduce the number of possible diagnoses.

Because of the completeness of the used test cases, the given structure of the spreadsheet and the assertion used in Experiment 2, only one single correct diagnosis candidate $\{S3, S13\}$ exists in the setting, i.e., no superfluous diagnosis candidates were reported by the algorithm. Note that the existence of more diagnosis candidates would not significantly influence the reported running times (but might in practice however make some intelligent form of ranking of the diagnoses necessary, which is beyond the scope of this paper).

The average running times for Experiment 1 using different problem sizes are shown in Table 2. All tests were made on a standard desktop computer with an Intel CoreDuo CPU and 3 GB RAM running Windows Vista and Java 1.6.

Beside the averaged computation times from several runs of the program, which range from below 1 second for the two-product case to up to 22.7 seconds in Experiment 1, we also report the number of constraint propagations and CSP solving steps required to determine the minimal conflicts, as this is the most costly operation in the diagnostic procedure. Note that in some situations, constraint propagation is sufficient to identify inconsistencies between the test cases and the spreadsheet definition. However, to determine whether a set of formulas represents a diagnosis, one arbitrary solution to the Constraint Satisfaction Problem has to be calculated.

[4]Note that diagnoses of higher cardinality might be hard to understand for end users.

#Prods.	#Vars	#Constrs.	#CSP prop.	#CSP solved	Diag. time (ms)
2	40	16	67	37	1981
3	55	19	80	43	4477
4	70	22	105	55	12028
5	85	25	126	65	21778
6	100	28	141	72	38470
7	115	31	152	77	49277
8	130	34	161	81	59062

Table 3. Experiment 2 measurements (double fault diagnosis).

Overall, we interpret the running times for diagnosis to be promising, in particular as they were achieved using a non-optimized proof-of-concept prototype built with standard, general-purpose algorithms, which did not make use of any domain-specific heuristics and only used one single test case and very few assertions, i.e., one assertion per product that specifies that the total production costs must be lower than the total revenue for the product.

Regarding the analyzed problem sizes, the example described above was actually inspired by a real-world sales forecast tool developed in MS Excel. With respect to the number of formulas contained in our experiments, note that a major fraction of the EUSES corpus of spreadsheets [16] contains far less formulas than we used in our evaluation setting. The complexity of our benchmark problems is also at least comparable or even higher than the complexity of the programs used for evaluation in recent repair-oriented approaches ([1], [2])

However, for some complex, real-world spreadsheets containing several hundreds of formulas with which we also experimented, the running times exceed the time frames that are acceptable for interactive debugging sessions. In such situations, automated offline batch testing can be one of the possible options; a future enhancement of the approach to scalability to very large spreadsheet programs is sketched in the next section.

5. Summary and current work

In this paper, we have presented a new constraint- and model-based fault localization approach for spreadsheet programs using model-based diagnosis techniques. In contrast to previous, often heuristics-based approaches, our method relies on the systematic analysis of possible problem causes using positive and negative test cases as well as user-specified assertions. A first evaluation of the method demonstrates the general feasibility of the approach.

Our work is both related to previous work in the areas of model-based software debugging and spreadsheet debugging from the software engineering field. Applying MBD techniques for logic programs was for example proposed in the early 1990s in [11]. Later on, model-based debugging was used to find errors in hardware designs specified in the VHDL language [19] and finally also to imperative languages (Java) [26]. Furthermore, similar approaches have been taken to find errors in ontologies [18], [17], special-purpose knowledge-bases [14] or declarative navigation specifications of intelligent user interfaces [13]. With respect to these AI-based software debugging methods, our work

continues previous research in the area and thus further broadens the application scope of these techniques to spreadsheet programs, a class of end user programs widely used in industrial practice.

Considering existing approaches to spreadsheet debugging, we see our work as a first step toward an additional method that can be used in combination with existing methods for error prevention based on assertions [5], "What You See Is What You Test" (WYSIWYT) techniques [5] or automatic test-case generation [1]. In addition, it can be seen as a complementary means to support the user in semi-automatic and interactive approaches which are also capable of generating repair suggestions based on heuristics and a set of predefined mutation operators [2].

Our current work covers the following improvements of the basic method developed in this paper.

- Scalability: Beside algorithmic optimizations, we are currently evaluating hierarchical problem decomposition as a means to improve the scalability of the approach. In particular, we focus on an iterative diagnosis refinement strategy, which was developed in the context of the diagnosis of complex telecommunication switch configurations [15]; see also [9]. The idea is to diagnose the system on different abstraction levels, where whole blocks of cells are treated as one constraint thus reducing the number of potentially faulty components. These blocks could be either defined by the end user or detected automatically based on heuristics such as the contingency of cells or the organization of formulas on different sheets.
- Formula analysis and repair: Our debugging approach is currently focused on the identification of problematic formulas. The combination of repair methods as described in [2] or [3] as well as the extension of the diagnostic procedure to formula contents as proposed in [17] are part of our current work.
- In many applications of model-based diagnosis, one problem is that the number of possible diagnosis (candidates) quickly increases. Thus, methods for ranking the diagnosis and in particular algorithms for determining an optimal set of additional measurements as for example described in [12] can be integrated in our method.
- Integration into the spreadsheet system: In order to fully support the typical interactive debugging process of spreadsheet, a set of user-friendly annotation tools and plug-ins for spreadsheet environments such as MS Excel have to be developed. In addition, we are currently extending the set of supported MS Excel functions and and data types in our prototype system.

Overall, we view our work in the line of previous works in the area of model-based software debugging and as a first step toward the application of these techniques to spreadsheets programs. The experiments reported in this paper are aimed at demonstrating the general feasibility of a the approach, which represents another example of how AI technology can be applied to support software engineering processes. In general, we also believe that the proposed transformation of the function-oriented spreadsheet program to the constraint satisfaction formalism is particularly advantageous because Constraint Satisfaction Problems are well-understood and many of the highly-efficient algorithms developed in this field can be applied.

References

[1] Robin Abraham and Martin Erwig, 'Autotest: A tool for automatic test case generation in spreadsheets', in *Proceedings of the Visual Languages and Human-Centric Computing (VLHCC'06)*, pp. 43–50, Brighton, (2006).

[2] Robin Abraham and Martin Erwig, 'Goaldebug: A spreadsheet debugger for end users', in *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*, pp. 251–260, Minneapolis, MN, (2007).

[3] Robin Abraham and Martin Erwig, 'Mutation operators for spreadsheets', *IEEE Transactions on Software Engineering*, **35**(1), 94–108, (2009).

[4] Tudor Antoniu, Paul A. Steckler, Shriram Krishnamurthi, Erich Neuwirth, and Matthias Felleisen, 'Validating the unit correctness of spreadsheet programs', in *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, pp. 439–448, Edinburgh, Scotland, (2004).

[5] Margaret Burnett, Curtis Cook, Omkar Pendse, Gregg Rothermel, Jay Summet, and Chris Wallace, 'End-user software engineering with assertions in the spreadsheet paradigm', in *Proceedings of the International Conference on Software Engineering (ICSE'03)*, pp. 93–103, Portland, Oregon, (2003).

[6] Margaret Burnett, Andrei Sheretov, Bing Ren, and Gregg Rothermel, 'Testing homogeneous spreadsheet grids with the "what you see is what you test" methodology', *IEEE Transactions on Software Engineering*, **28**(6), 576–594, (2002).

[7] David Chadwick, Brian Knight, and Kamalasen Rajalingham, 'Quality control in spreadsheets: A visual approach using color codings to reduce errors in formulae', *Software Quality Control*, **9**(2), 133–143, (2001).

[8] Chris Chambers and Martin Erwig, 'Automatic detection of dimension errors in spreadsheets', *Journal of Visual Languages and Computing*, **20**(4), 269–283, (2009).

[9] Luca Chittaro and Roberto Ranon, 'Hierarchical model-based diagnosis based on structural abstraction', *Artificial Intelligence*, **2004**, 1–2, (2004).

[10] Markus Clermont, 'Heuristics for the automatic identification of irregularities in spreadsheets', in *Proceedings of the 1st workshop on End-user software engineering (WEUSE I)*, pp. 1–6, St. Louis, MO, (2005). ACM.

[11] Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré, 'Model-based diagnosis meets error diagnosis in logic programs', in *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, pp. 1494–1499, Chambery, France, (1993).

[12] J de Kleer and B C Williams, 'Diagnosing multiple faults', *Artifical Intelligence*, **32**(1), 97–130, (1987).

[13] Alexander Felfernig, Gerhard Friedrich, Klaus Isak, Kostyantyn Shchekotykhin, Erich Teppan, and Dietmar Jannach, 'Automated debugging of recommender user interface descriptions', *Applied Intelligence*, **31**(1), 1–14, (2009).

[14] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Stumptner, 'Consistency-based diagnosis of configuration knowledge bases', *Artifical Intelligence*, **152**(2), 213–234, (2004).

[15] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, Markus Stumptner, and Markus Zanker, 'Hierarchical diagnosis of large configurator knowledge bases', in *Proceedings of the 12th Intl. Workshop on Principles of Diagnosis (DX'01)*, pp. 55–62, Via Lattea, Italy, (2001).

[16] Marc Fisher and Gregg Rothermel, 'The EUSES spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms', in *WEUSE I: Proceedings of the first workshop on End-user software engineering*, pp. 1–5, St. Louis, Missouri, (2005).

[17] G. Friedrich, S. Rass, and K. Shchekotykhin, 'A general method for diagnosing axioms', in *Proceedings of the 17th International Workshop on Principles of Diagnosis (DX'06)*, pp. 101–108, Penaranda de Duero, Burgos, Spain, (2006).

[18] Gerhard Friedrich and Kostyantyn M. Shchekotykhin, 'A general diagnosis method for ontologies', in *Proceedings of the 4th International Semantic Web Conference (ISWC 2005)*, pp. 232–246, Galway, Ireland, (2005).

[19] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa, 'Model-based diagnosis of hardware designs', *Artificial Intelligence*, **111**(1-2), 3–39, (1999).

[20] Ulrich Junker, 'Quickxplain: Preferred explanations and relaxations for over-constrained problems', in *Proceedings of the The Nineteenth National Conference on Artificial Intelligence (AAAI'04)*, pp. 167–172, (2004).

[21] J. Paine, 'Excelsior: bringing the benefits of modularisation to excel', in *Proceedings of the International Conference of EuSpRIG'05*, Greenwich, UK, (2005).

[22] Raymond R. Panko, 'What we know about spreadsheet errors', *Journal of End User Computing*, **10**(2), 15–21, (1998).

[23] Raymond Reiter, 'A theory of diagnosis from first principles', *Artifical Intelligence*, **32**(1), 57–95, (1987).

[24] Christopher Scaffidi, Mary Shaw, and Brad Myers, 'Estimating the numbers of end users and end user programmers', in *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC'05)*, pp. 207–214, Washington, DC, USA, (2005).

[25] E.P.K Tsang, *Foundations of Constraint Satisfaction*, Academic Press, 1993.

[26] Franz Wotawa, 'On the relationship between model-based debugging and program slicing', *Artificial Intelligence*, **135**(1-2), 125–143, (2002).