The Journal of Systems and Software 82 (2009) 1780–1792

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A practical evaluation of spectrum-based fault localization ☆

Rui Abreu [a,*], Peter Zoeteweij [a], Rob Golsteijn [b], Arjan J.C. van Gemund [a]

[a] Delft University of Technology, The Netherlands
[b] NXP Semiconductors, The Netherlands

ARTICLE INFO

Article history:
Available online 26 June 2009

Keywords:
Test data analysis
Software fault diagnosis
Program spectra
Real-time and embedded systems
Consumer electronics

ABSTRACT

Spectrum-based fault localization (SFL) shortens the test–diagnose–repair cycle by reducing the debugging effort. As a light-weight automated diagnosis technique it can easily be integrated with existing testing schemes. Since SFL is based on discovering statistical coincidences between system failures and the activity of the different parts of a system, its diagnostic accuracy is inherently limited. Using a common benchmark consisting of the Siemens set and the `space` program, we investigate this diagnostic accuracy as a function of several parameters (such as quality and quantity of the program spectra collected during the execution of the system), some of which directly relate to test design. Our results indicate that the superior performance of a particular similarity coefficient, used to analyze the program spectra, is largely independent of test design. Furthermore, near-optimal diagnostic accuracy (exonerating over 80% of the blocks of code on average) is already obtained for low-quality error observations and limited numbers of test cases. In addition to establishing these results in the controlled environment of our benchmark set, we show that SFL can effectively be applied in the context of embedded software development in an industrial environment.

1. Introduction

Testing, debugging, and verification represent a major expenditure in the software development cycle (Hailpern and Santhanam, 2002), which is to a large extent due to the labor-intensive task of diagnosing the faults (bugs) that cause tests to fail. Because under typical market conditions, only those faults that affect the user most can be solved before the release deadline, the efficiency with which faults can be diagnosed and repaired directly influences software reliability. Automated diagnosis can help to improve this efficiency.

Diagnosis techniques are complementary to testing in two ways. First, for tests designed to verify correct behavior, they generate information on the root cause of test failures, focusing the subsequent tests that are required to expose this root cause. Second, for tests designed to expose specific potential root causes, the extra information generated by diagnosis techniques can help to further reduce the set of remaining possible explanations. Given its incremental nature (i.e., taking into account the results of an entire sequence of tests), automated diagnosis alleviates much of the work of selecting tests in the latter category, and can hence have a profound impact on the test–diagnose–repair cycle.

An important part of diagnosis and repair consist in localizing faults, and several tools for automated debugging and systems diagnosis implement spectrum-based fault localization (SFL), an approach to diagnosis based on an analysis of the differences in *program spectra* (Harrold et al., 2000; Reps et al., 2000) for *passed* and *failed* runs. Passed runs are executions of a program that completed correctly, whereas failed runs are executions in which an error was detected. A program spectrum is an execution profile that indicates which parts of a program are active during a run. Spectrum-based fault localization entails identifying the part of the program whose activity correlates most with the detection of errors. Examples of tools that implement this approach are Pinpoint (Chen et al., 2002), which focuses on large, dynamic on-line transaction processing systems (Jones et al., 2002) whose implementation focuses on the analysis of C programs, and AMPLE (Dallmeier et al., 2005), which focuses on object-oriented software (see Section 9 for a discussion).

Spectrum-based fault localization does not rely on a model of the system under investigation. It can easily be integrated with existing testing procedures, and because of the relatively small overhead with respect to CPU time and memory requirements, it lends itself well for application within resource-constrained environments (Zoeteweij et al., 2007). However, the efficiency of SFL comes at the cost of a limited *diagnostic accuracy*. As an indication,

☆ This work has been carried out as part of the TRADER project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the BSIK03021 program.

* Corresponding author.
E-mail addresses: r.f.abreu@tudelft.nl (R. Abreu), p.zoeteweij@tudelft.nl (P. Zoeteweij), rob.golsteijn@nxp.com (R. Golsteijn), a.j.c.vangemund@tudelft.nl (A.J.C. van Gemund).

0164-1212/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2009.06.035

in one of the experiments described in Section 5, on average 20% of a program still needs to be inspected after the diagnosis due to a low number of failed runs.

In SFL, a *similarity coefficient* is used to rank potential fault locations. In earlier work (Abreu et al., 2006b), we obtained preliminary evidence that the Ochiai similarity coefficient, known from the biology domain (see, e.g., da Silva Meyer et al., 2004), can improve diagnostic accuracy over eight other coefficients, including those used by the Pinpoint and Tarantula tools mentioned above. Extending as well as generalizing this previous result, in this paper we investigate the main factors that influence the accuracy of SFL in a much wider setting. Apart from the influence of the similarity coefficient on the diagnostic accuracy, we also study the influence of the quality and quantity of the (pass/fail) observations used in the analysis.

Quality of the observations relates to the classification of runs as passed or failed. Since most faults lead to errors only under specific input conditions, and as not all errors propagate to system failures, this parameter is relevant because error detection mechanisms are usually not ideal. Quantity of the observations relates to the number of passed and failed runs available for the diagnosis. If fault localization has to be performed at run-time, e.g., as a part of a recovery mechanism, one cannot wait to accumulate many observations to diagnose a potentially disastrous error until sufficient confidence is obtained. In addition, quality and quantity of the observations both relate to test coverage. Varying the observation context with respect to these two observational parameters allows a much more thorough investigation of the influence of similarity coefficients. Our study is based on a widely-used set of benchmark faults (single faults) consisting of the Siemens set (Hutchins et al., 1994) and the `space` program, both of which are available from the Software-artifact Infrastructure Repository (Do et al., 2005).

While the benchmark problems are well-suited for studying the influence of the similarity coefficient and the quality and quantity of the observations, they give little indication on the accuracy of spectrum-based fault localization for large-scale codes, and the kind of problems that are encountered in practice. For this reason, we also report our experience with implementing SFL for an industrial software product, namely the control software of a particular product line of hybrid analog/digital LCD television sets.

The main contributions of our work are the following. We show that for the purpose of software fault diagnosis, the Ochiai similarity coefficient consistently outperforms several other coefficients used in fault localization and data clustering. Intuitively, this can be attributed to the Ochiai coefficient being more sensitive to activity of potential fault locations in failed runs than to activity in passed runs, which is well suited to software fault diagnosis because execution of faulty code does not necessarily lead to failures, while failures always involve a fault. We establish this result across the entire quality space, and for varying numbers of runs involved. Furthermore, we show that near-optimal diagnostic accuracy (exonerating over 80% of all code on average) is already obtained for low-quality (ambiguous) error observations, while, in addition, only a few runs are required. In addition to establishing these results in the controlled environment of our benchmark set, we show that SFL can effectively be applied in the industrial development of embedded software for resource-constrained systems.

The remainder of this paper is organized as follows. In Section 2 we introduce some basic concepts and terminology, and explain the diagnosis technique in more detail. In Section 3 we describe our experimental setup. In Sections 4–6 we describe the experiments on the similarity coefficient, and the quality and quantity of the observations, respectively. Preliminary results of Section 4 appeared in (Abreu et al., 2006b), and of Sections 5 and 6 in (Abreu et al., 2007). In Sections 7 and 8 we give an account of the industrial case study. Related work is discussed in Section 9. We conclude, and discuss possible directions for future work in Section 10.

2. Preliminaries

In this section we introduce program spectra, and describe how they are used in spectrum-based fault localization.

2.1. Failures, errors, and faults

As defined in Avižienis et al. (2004), we use the following terminology. A *failure* is an event that occurs when delivered service deviates from correct service. An *error* is a system state that may cause a failure. A *fault* is the cause of an error in the system.

In this paper we apply this terminology to computer programs, where faults are *bugs* in the program code. Specifically in our benchmark experiments, these programs transform an input file into an output file in a single run, and failures occur when the output for a given input differs from the expected output for that input.

To illustrate these concepts, consider the C function in Fig. 1. It is meant to sort, using the bubble sort algorithm, a sequence of `n` rational numbers whose numerators and denominators are stored in the parameters `num` and `den`, respectively. There is a fault (bug) in the swapping code within the body of the `if` statement (labeled as block 4): only the numerators of the rational numbers are swapped while the denominators are left in their original order. In this case, a failure occurs when `RationalSort` changes the contents of its argument arrays in such a way that the result is not a sorted version of the original. An error occurs after the code inside the conditional statement is executed, while $\texttt{den[j]} \neq \texttt{den[j+1]}$. Such errors can be temporary, and do not automatically lead to failures. For example, if we apply `RationalSort` to the sequence $\langle \frac{4}{1}, \frac{2}{2}, \frac{0}{1} \rangle$, an error occurs after the first two numerators are swapped. However, this error is "canceled" by later swapping actions, and the sequence ends up being sorted correctly.

Error detection is a prerequisite for the fault localization technique studied in this paper: we must know that something is wrong before we can try to locate the responsible fault. Failures constitute a rudimentary form of error detection, but many errors remain latent and never lead to a failure. An example of a technique that increases the number of errors that can be detected is array bounds checking. Failure detection and array bounds checking are both examples of *generic* error detection mechanisms, that can be applied without detailed knowledge of a program. Other

```
void RationalSort(int n, int *num, int *den){
   /* block 1 */
   int i,j,temp;

   for ( i=n-1; i>=0; i-- ) {
      /* block 2 */
      for ( j=0; j<i; j++ ) {
         /* block 3 */
         if (RationalGT(num[j], den[j],
                        num[j+1], den[j+1])) {
            /* block 4 */
            /* Bug: forgot to swap denominators */
            temp = num[j];
            num[j] = num[j+1];
            num[j+1] = temp;
         }
      }
   }
}
```

Fig. 1. A faulty C function for sorting rational numbers.

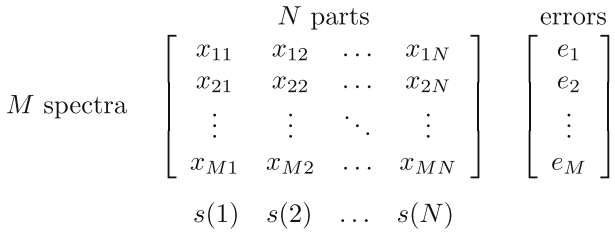

Fig. 2. The ingredients of spectrum-based fault localization.

examples are the detection of null pointer handling, `malloc` problems, and deadlock detection in concurrent systems. Examples of *program specific* mechanisms are precondition and postcondition checking, and the use of assertions.

2.2. Program spectra

A program spectrum (Reps et al., 2000) is a collection of data that provides a specific view on the dynamic behavior of software. This data is collected at run-time, and typically consist of a number of counters or flags for the different parts of a program. Many different forms of program spectra exist, see Harrold et al. (2000) for an overview. Although we work with so-called block-hit spectra, the approach studied in this paper easily generalizes to other types of program spectra (e.g., path-hit spectra, data-dependence-hit spectra).

A block hit spectrum contains a flag for every block of code in a program, that indicates whether or not that block was executed in a particular run. With a block of code we mean a C language statement, where we do not distinguish between the individual statements of a compound statement, but where we do distinguish between the cases of a switch statement.[1] As an illustration, we have identified the blocks of code in Fig. 1.

2.3. Spectrum-based fault localization

The hit spectra of M runs constitute a binary matrix, whose columns correspond to N different parts (blocks in our case) of a program (see Fig. 2). The information in which runs an error was detected constitutes another column vector, the error vector. This vector can be thought to represent a hypothetical part of the program that is responsible for all observed errors. Spectrum-based fault localization essentially consists in identifying the part whose column vector resembles the error vector most.

In the field of data clustering, resemblances between vectors of binary, nominally scaled data, such as the columns in our matrix of program spectra, are quantified by means of *similarity coefficients* (see, e.g., Jain and Dubes, 1988). Many similarity coefficients exist. As an example, below are three different similarity coefficients, namely the Jaccard coefficient s_J, which is used by the Pinpoint tool (Chen et al., 2002), the coefficient s_T, used in the Tarantula fault localization tool (Jones and Harrold, 2005), and the Ochiai coefficient s_O, used in the molecular biology domain (da Silva Meyer et al., 2004):

$$s_J(j) = \frac{a_{11}(j)}{a_{11}(j) + a_{01}(j) + a_{10}(j)}, \tag{1}$$

$$s_T(j) = \frac{\frac{a_{11}(j)}{a_{11}(j)+a_{01}(j)}}{\frac{a_{11}(j)}{a_{11}(j)+a_{01}(j)} + \frac{a_{10}(j)}{a_{10}(j)+a_{00}(j)}}, \tag{2}$$

$$s_O(j) = \frac{a_{11}(j)}{\sqrt{(a_{11}(j) + a_{01}(j)) \cdot (a_{11}(j) + a_{10}(j))}}, \tag{3}$$

[1] This is a different notion from a *basic block*, which is a block of code that has no branch.

Table 1
SFL applied on six runs of the `RationalSort` program.

Input	Block					Error
	1	2	3	4	5	
$I_1 = \langle\rangle$	1	0	0	0	0	0
$I_2 = \langle\frac{1}{4}\rangle$	1	1	0	0	0	0
$I_3 = \langle\frac{2}{1}, \frac{1}{1}\rangle$	1	1	1	1	1	0
$I_4 = \langle\frac{4}{1}, \frac{2}{2}, \frac{0}{1}\rangle$	1	1	1	1	1	0
$I_5 = \langle\frac{3}{1}, \frac{2}{2}, \frac{4}{3}, \frac{1}{4}\rangle$	1	1	1	1	1	1
$I_6 = \langle\frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{1}{1}\rangle$	1	1	1	0	1	0
s_J	0.17	0.20	0.25	0.33	0.25	
s_T	0.50	0.56	0.63	0.71	0.63	
s_O	0.41	0.45	0.50	0.58	0.50	

where $a_{11}(j)$ is the number of failed runs in which part j is involved, $a_{10}(j)$ is the number of passed runs in which part j is involved, $a_{01}(j)$ is the number of failed runs in which part j is not involved, and $a_{00}(j)$ is the number of passed runs in which part j is not involved, i.e., referring to Fig. 2,

$$a_{00}(j) = |\{i \mid x_{ij} = 0 \wedge e_i = 0\}|,$$
$$a_{01}(j) = |\{i \mid x_{ij} = 0 \wedge e_i = 1\}|,$$
$$a_{10}(j) = |\{i \mid x_{ij} = 1 \wedge e_i = 0\}|,$$
$$a_{11}(j) = |\{i \mid x_{ij} = 1 \wedge e_i = 1\}|.$$

Note that $a_{10}(j) + a_{11}(j)$ equals the number of runs in which part j is involved, and that $a_{10}(j) + a_{00}(j)$ and $a_{11}(j) + a_{01}(j)$ equal the number of passed and failed runs, respectively. The latter two numbers are equal for all j. Similarly, for all j, the four counters sum op to the number of runs M.

Under the assumption that a high similarity to the error vector indicates a high probability that the corresponding parts of the software cause the detected errors, the calculated similarity coefficients rank the parts of the program with respect to their likelihood of containing the faults.

To illustrate the approach, suppose that we apply the `RationalSort` function to the input sequences $I_1, \ldots, I_6$ shown in Table 1. The block hit spectra for these runs are shown in the central part of the table ('1' denotes a hit), where block five corresponds to the body of the `RationalGT` function, which has not been shown in Fig. 1. I_1, I_2, and I_6 are already sorted, and lead to passed runs. I_3 is not sorted, but the denominators in this sequence happen to be equal, hence no error occurs. I_4 is the example from Section 2.1: an error occurs during its execution, but goes undetected. For I_5 the program fails, since the calculated result is $\langle\frac{1}{1}, \frac{2}{2}, \frac{4}{3}, \frac{3}{4}\rangle$ instead of $\langle\frac{1}{4}, \frac{2}{2}, \frac{4}{3}, \frac{3}{1}\rangle$, which is a clear indication that an error has occurred. For this data, the calculated similarity coefficients $s_{x\in\{J,T,P\}}(1), \ldots, s_{x\in\{J,T,P\}}(5)$ listed at the bottom of Table 1 (correctly) identify block 4 as the most likely location of the fault.

3. Experimental setup

In this section we describe the benchmark set that we use in our experiments. We also detail how we extract the data of Fig. 2, and define how we measure diagnostic accuracy.

3.1. Benchmark set

In our study we work with two sets of faults that are available from the Software-artifact Infrastructure Repository (SIR, Do et al., 2005):

- the Siemens set (Hutchins et al., 1994), which is a widely-used collection of benchmark faults in seven small C programs, and
- a set of faults in a somewhat larger program called `space`.

Table 2
Set of programs used in the experiments.

Program	Faulty versions	Blocks	Test cases	Description
print_tokens	7	110	4130	Lexical analyzer
print_tokens2	10	105	4115	Lexical analyzer
replace	32	124	5542	Pattern recognition
schedule	9	53	2650	Priority scheduler
schedule2	10	60	2710	Priority scheduler
tcas	41	20	1608	Altitude separation
tot_info	23	44	1052	Information measure
space	38	777	13,585	Array definition language

The Siemens set and space are the only programs in SIR that are ANSI-C compliant, and that could therefore be handled by our instrumentation tool (see Section 3.2). Table 2 contains details about our benchmark set. For all eight programs, a correct version, and a number of faulty versions is available. Each faulty version contains a single fault, but this fault may span through multiple statements and/or functions. In addition, every program has a set of inputs (test cases) designed to provide full code coverage. For space, 1000 test suites are provided that consist of a selection of (on average) 150 test cases.

In our experiments we were not able to use all the faults offered by the Siemens set and space. Because we conduct our experiments using block hit spectra, we cannot use faults that are located outside a block, such as global variable initializations. Versions 4 and 6 of print_tokens contain such faults and were therefore excluded. Version 9 of schedule2, version 32 of replace, and versions 1, 2, 32, and 34 of space were not considered in our experiments because no test case fails and therefore the existence of a fault was never revealed. In total, we used 162 faulty versions in our experiments: 128 out of 132 faulty versions provided by the Siemens set, and 34 out of 38 faulty versions of space.

3.2. Data acquisition

3.2.1. Collecting spectra

To obtain block hit spectra, we automatically instrument the source code of all faulty versions of the programs in our benchmark set. A function call is inserted at the beginning of every block of code to log its execution. For the Siemens set, the spectra are generated for all test cases that are provided with the programs. For the faulty versions of space, we randomly choose one of the 1000 test suites. For instrumentation we use the parser generator Front (Augusteijn, 2002), which is part of the development environment of NXP Semiconductors (NXP), the main industrial partner in the TRADER project (Embedded Systems Institute, 2009). The overhead of the instrumentation on the execution time is measured to be approximately 6% on average (with standard deviation of 5%). The programs were compiled on a Fedora Core release 4 system with gcc-3.2. For details of the instrumentation process, see (Abreu et al., 2006a).

3.2.2. Error detection

As for each program our benchmark set provides a correct version, we use the output of the correct version of each program as error detection reference. We characterize a run as 'failed' if its output differs from the corresponding output of the correct version, and as 'passed' otherwise.

3.3. Evaluation metric

As spectrum-based fault localization creates a ranking of blocks in order of likelihood to be at fault, we can retrieve how many blocks we still need to inspect until we hit the faulty block. If other blocks have the same similarity coefficient as the fault location, we use the average ranking position for these blocks. In those cases where the fault spans multiple locations, we verified that there is one block that is involved in the fault, and that is executed in all failed runs. This is the block that our evaluation metric is based on for the multiple-location faults. Repairing the fault at just this location would lead to iterative testing and debugging, but in our experiments we assume that the program is bug-free after the first iteration.

For all $j \in \{1,\ldots,N\}$, let $s(j)$ denote the similarity coefficient calculated for block j. Specifically, let f be the index of the block that is known to contain the fault, and let $s(f)$ denote the similarity coefficient calculated for this block. Then, assuming that on average, half of the blocks j with $s(j) = s(f)$ are inspected before block f is found, the number of blocks that need to be inspected in total is given by

$$\tau = \frac{|\{j|s(j) > s(f)\}| + |\{j|s(j) \geqslant s(f)\}| - 1}{2}. \quad (4)$$

We define accuracy, or quality of the diagnosis as the effectiveness to pinpoint the faulty block. This metric represents the fraction of all blocks that need not be considered when searching for the fault by traversing the ranking. It is defined as

$$q_d = \left(1 - \frac{\tau}{N-1}\right). \quad (5)$$

In the remainder of this paper, values for q_d will be expressed as percentages.

4. Similarity coefficient impact

At the end of Section 2.3 we reduced the problem of spectrum-based fault localization to finding resemblances between binary vectors. The key element of this technique is the calculation of a similarity coefficient. Many different similarity coefficients are used in practice, and in this section we investigate the impact of the similarity coefficient on the diagnostic accuracy q_d.

For this purpose, we evaluate q_d on all faults in our benchmark set, using nine different similarity coefficients. We only report the results for the Jaccard coefficient of Eq. (1), the coefficient used in the Tarantula fault localization tool as defined in Eq. (2), and the Ochiai coefficient of Eq. (3). We experimentally identified the latter as giving the best results among all eight coefficients used in a data clustering study in molecular biology (da Silva Meyer et al., 2004). Table 3 contains the details of the coefficients that are involved in this study, and that have not already been introduced in Section 2.3. For brevity, the block index j as an argument to the counter functions $a_{11},\ldots,a_{00}$ has been omitted.

In addition to the coefficient s_T of Eq. (2), the Tarantula tool uses a second coefficient, which amounts to the maximum of the two fractions in the denominator of Eq. (2). This second coefficient is interpreted as a *brightness* value for visualization purposes, but the experiments in (Jones and Harrold, 2005) indicate that s_T can

Table 3
Additional similarity coefficients evaluated; see da Silva Meyer et al. (2004) for references.

Sorensen–Dice	$\frac{2 \cdot a_{11}}{2 \cdot a_{11} + a_{01} + a_{10}}$
Anderberg	$\frac{a_{11}}{a_{11} + 2 \cdot (a_{01} + a_{10})}$
Simple-matching	$\frac{a_{11} + a_{00}}{a_{11} + a_{01} + a_{10} + a_{00}}$
Rogers and Tanimoto	$\frac{a_{11} + a_{00}}{a_{11} + a_{00} + 2 \cdot (a_{01} + a_{10})}$
Ochiai II	$\frac{a_{11} \cdot a_{00}}{\sqrt{(a_{11} + a_{01}) \cdot (a_{11} + a_{10}) \cdot (a_{00} + a_{01}) \cdot (a_{00} + a_{10})}}$
Russel and Rao	$\frac{a_{11}}{a_{11} + a_{01} + a_{10} + a_{00}}$

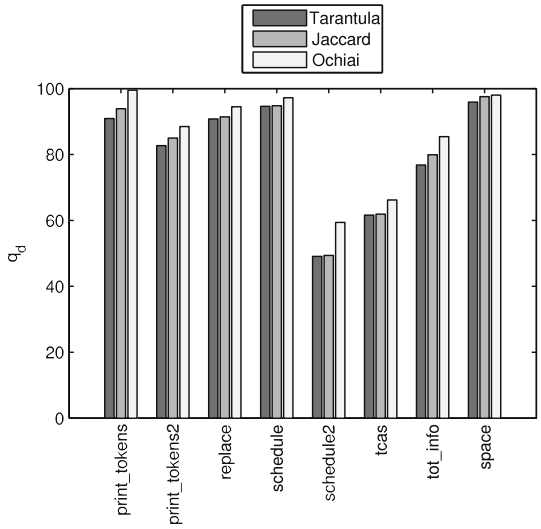

Fig. 3. Diagnostic accuracy q_d.

be studied in isolation. For this reason, we have not taken the brightness coefficient into account.

Fig. 3 shows the results of this experiment. It plots q_d, as defined by Eq. (5), for the Tarantula, Jaccard, and Ochiai coefficients, averaged per program of our benchmark set. See (Abreu et al., 2006b) for more details on these experiments.

An important conclusion that we can draw from these results is that under the specific conditions of our experiment, the Ochiai coefficient gives a better diagnosis: it always performs at least as good as the other coefficients, with an average improvement of 4% over the second-best case, and improvements of up to 30% for individual faults. This effect can be explained as follows.

First, note that if $a_{11} = 0$, all three coefficients evaluate to 0, so blocks that are not executed in failed runs rank lowest. For the case that $a_{11} > 0$, the rankings produced by the Tarantula, Jaccard, and Ochiai coefficients are the same as the ranking produced by the respective coefficients below, where the block index j as an argument to the coefficients and counter functions a_{11} and a_{10} has been omitted for brevity

$$s'_T = \frac{1}{1 + c_T \cdot \frac{a_{10}}{a_{11}}}, \quad s'_J = \frac{a_{11}}{c_J + a_{10}}, \quad s'_O = \frac{a_{11}}{1 + \frac{a_{10}}{a_{11}}}.$$

Here $c_T = \frac{a_{11}+a_{01}}{a_{10}+a_{00}}$ and $c_J = a_{11} + a_{01}$, both of which are constant for all blocks, and do not influence the ranking. Coefficient s'_T is derived from s_T by dividing the numerator and denominator by $\frac{a_{11}}{a_{11}+a_{01}}$. Coefficient s'_O is derived by squaring s_O, dividing the denominator of the resulting fraction by the constant $a_{11} + a_{01}$, and dividing the numerator and denominator by a_{11}. Note that for $a_{11} > 0$, none of these operations modify the rankings implied by the Tarantula and Ochiai coefficients. The expression for s'_J is identical to that for s_J except for the introduction of c_J.

By thus rewriting the coefficients, it becomes apparent that the rankings implied by the Tarantula, Jaccard, and Ochiai coefficients depend only on a_{11} and a_{10}, i.e., the involvement of a block in passed and failed runs. It can also be seen that for $a_{10} = 0$, it follows that $s'_T = 1$, which implies that all blocks that are exclusively active in failed runs rank with the same, and highest s_T. This explains the improvement of Jaccard and Ochiai over Tarantula, because these coefficients both take a_{11} into account for ranking the blocks that have $a_{10} = 0$. The improved performance of the Ochiai coefficient over the Tarantula and Jaccard coefficients can be explained by observing that increasing a_{11} both increases the numerator, and decreases the denominator of s'_O, whereas to s'_T and s'_J, only one of these effects applies. As a result, compared to the other coefficients, Ochiai is much more sensitive to presence in failed runs than to presence in passed runs. This is well-suited to fault diagnosis because the execution of faulty code does not necessarily lead to a failure, while failures always involve a fault.

5. Observation quality impact

Before reaching a definitive decision to prefer one similarity coefficient over another, as suggested by the results in Section 4, we want to verify that the impact of this decision is independent of specific conditions in our experiments. Because of its relation to test coverage, and to the error detection mechanism used to characterize runs as passed or failed, an important condition in this respect is the quality of the error detection information used in the analysis.

In this section we define a measure of quality of the error observations, and show how it can be controlled as a parameter if the fault location is known, as is the case in our experimental setup. Thus, we verify the results of the previous section for varying observation quality values. Investigating the influence of this parameter will also help us to assess the potential gain of more powerful error detection mechanisms and better test coverage on diagnostic accuracy.

5.1. A measure of observation quality

Correctly locating the fault is trivial if the column for the faulty part in the matrix of Fig. 2 resembles the error vector exactly. This would mean that an error is detected if, and only if the faulty part is active in a run. In that case, any coefficient is bound to deliver a highly accurate diagnosis. However, spectrum-based fault localization suffers from the following phenomena.

- Most faults lead to an error only under specific input conditions. For example, if a conditional statement contains the faulty condition $\mathtt{v} < \mathtt{c}$, with $\mathtt{v}$ a variable and $\mathtt{c}$ a constant, while the correct condition would be $\mathtt{v} \leqslant \mathtt{c}$, no error occurs if the conditional statement is executed, unless the value of $\mathtt{v}$ equals $\mathtt{c}$.
- Similarly, as we have already seen in Section 2.1, errors need not propagate all the way to failures (Morell, 1990; Voas, 1992), and may thus go undetected. This effect can partially be remedied by applying more powerful error detection mechanisms, but for any realistic software system and practical error detection mechanism there will likely exist errors that go undetected.

As a result of both phenomena, the set of runs in which an error is detected will only be a subset of the set of runs in which the fault is activated.[2] We propose to use the ratio of the size of these two sets as a measure of observation quality for a diagnosis problem. Using the notation of Section 2.3, we define

[2] In our experimental setup, we do not consider effects that carry over from one run to another, so conversely, if an error is detected, the fault is always active.

$$q_e = \frac{a_{11}(f)}{a_{11}(f) + a_{10}(f)}, \quad (6)$$

where f is the known location of the fault, as in Section 3.3. This value can be interpreted as the unambiguity of the passed/failed data in relation to the fault being exercised, which may be loosely referred to as "error detection quality", hence the symbol q_e. In the remainder of this paper, values for q_e will be expressed as percentages.

A problem with the q_e measure is that no information on undetected errors is available: $a_{10}(f)$ counts both the undetected errors, and the number of times the fault location was activated without introducing an error. This can be summarized as follows, where X, E, and D denote activation of the fault location, the occurrence of an error, and detection of an error, respectively:

X	E	D	
0	0	0	$a_{00}(f)$
1	0	0	$a_{10}(f)$
1	1	0	
1	1	1	$a_{11}(f)$

Even though the ratio of the two contributions to $a_{10}(f)$ is unknown, it can still be influenced in our experimental setup. We will now describe our procedure for doing so.

5.2. Varying q_e

Subject to various factors such as the nature of the fault, the similarity coefficient used in the diagnosis, the design of the test data, but also the compiler and the operating system, each faulty version of a program in our benchmark set has an inherent value for q_e, which can be evaluated by collecting spectra and error detection information for all available test cases, and performing the diagnosis of Section 2.3. For the Siemens set, this inherent value for q_e ranges from 1.4% for `schedule2` to 20.3% for `tot_info`, whereas for `space` this value is measured to be 50.9% on average for our selection of test suites.

We can construct a different value for q_e by excluding runs that contribute either to $a_{11}(f)$ or to $a_{10}(f)$ as follows.

- Excluding a run that activates the fault location, but for which no error has been detected lowers $a_{10}(f)$, and will *increase* q_e.
- Excluding a run that activates the fault location and for which an error has been detected lowers $a_{11}(f)$, and will *decrease* q_e.

Excluding runs to achieve a certain value of q_e raises the question of which particular selection of runs to use. For this purpose we randomly sample passed or failed runs from the set of available runs to control q_e within a 99% confidence interval. We verified that the variance in the values measured for q_d is negligible.

Note that for decreasing q_e, i.e., obscuring the fault location, we have another option: setting failed runs to 'passed.' In our experiments we have tried both options, but the results were essentially the same. The results reported below are generated by excluding failed runs. Conversely, setting passed runs that exercise the fault location to 'failed' is not a good alternative for increasing q_e: this may obstruct the diagnosis as we cannot be certain that an error occurs for a particular data input. Moreover, it may allocate blame to parts of the program that are not related to the fault. Thus, excluding runs is always to be preferred as this does not compromise observation consistency. This way, we were able to vary q_e from 1% to 100% for all programs.

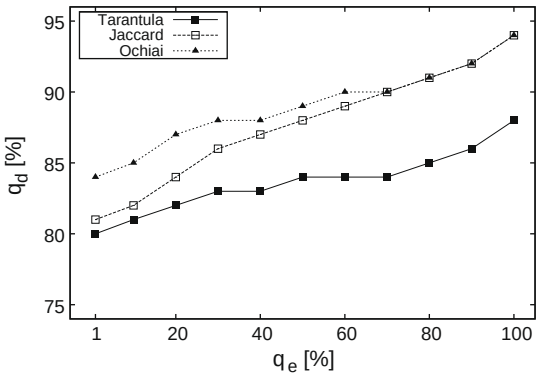

Fig. 4. Observation quality impact.

5.3. Similarity coefficients revisited

Using the technique for varying q_e introduced in Section 5.2 we revisit the comparative study of similarity coefficients in Section 4. Fig. 4 shows q_d for the three similarity coefficients, and values of q_e ranging from 1% to 100%. In this case, instead of averaging per program in our benchmark set, as we did in Fig. 3, we arithmetically averaged q_d over all 162 faulty program versions to summarize the results (this is valid because q_d is already normalized with respect to program size). As in Fig. 3, the graphs for the individual programs are similar, only having different offsets.

These results confirm what was suggested by the experiment in Section 4. The Ochiai similarity coefficient leads to a better diagnosis than the other eight, including the Jaccard coefficient and the coefficient of the Tarantula tool. Compared to the Jaccard coefficient the improvement is greatest for lower observation quality. As q_e increases, the performance of the Jaccard coefficient approaches that of the Ochiai coefficient. The improvement of the Ochiai coefficient over the Tarantula coefficient appears to be consistent.

Another observation that can be made from Fig. 4 is that all three coefficients provide a useful diagnosis (q_d around 80%) already for low q_e values ($q_e = 1\%$ implies that only around 1% of the runs that exercised the faulty block actually resulted in a failed run). The diagnostic accuracy increases as the quality of the error detection information improves, but the effect is not as strong as we expected. This suggests that more powerful error detection mechanisms, or test sets that cover more input conditions will have limited gain. In the next section we investigate a possible explanation, namely that not only the quality of observations, but also their quantity determines the diagnostic accuracy.

6. Observation quantity impact

To investigate the influence of the number of runs on the accuracy of spectrum-based fault localization, we evaluated q_d while varying the numbers of passed (N_P) and failed runs (N_F) that are involved in the diagnosis, across the benchmark set. Since all interesting effects appear to occur for small numbers of runs, we have focused on the range of 1–100 passed and failed runs. Although the number of available runs in the Siemens set ranges from 1052 (`tot_info`) to 5542 (`replace`), the number of runs that fail is comparatively small, down to a single run for `tcas` version 8. The situation is comparable for `space`, with only seven out of 13,585 runs failing for version 33. For this reason, even in the range 1–100, some selections of failed runs are not possible for some of the faulty versions.

Fig. 5 shows two representative examples of such evaluations, where we plot q_d according to the Ochiai coefficient for N_P and

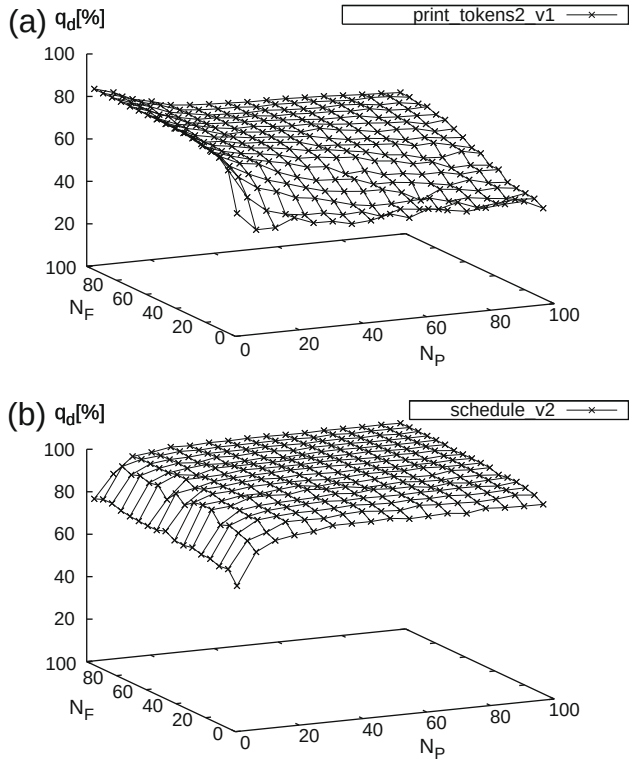

Fig. 5. Observation quantity impact.

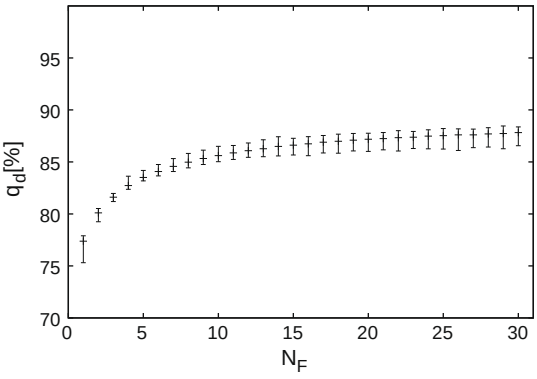

Fig. 6. Impact of N_F on q_d, on average.

N_F varying from 1 to 100. For each entry in these graphs, we averaged q_d over 50 randomly selected combinations of N_P passed runs and N_F failed runs, where we verified that the variance in the measured values of q_d is negligible. Apart from the apparent monotonic increase of q_d with N_F, we observe that for version 1 of `print_tokens2`, q_d decreases when more passed runs are added (Fig. 5a), while q_d increases for version 2 of `schedule` (Fig. 5b).

Given a set of faulty program versions that all allow failed runs to be selected up to a given value for N_F, we can average the measured values for q_d again over these versions. This summarizes several graphs of the kind shown in Fig. 5. This way, in Fig. 6 we plot the average q_d using the Ochiai coefficient for $1 \leqslant N_F \leqslant 30$ and $1 \leqslant N_P \leqslant 100$, projected on the $N_F \times q_d$ plane. The ticks on the vertical bars in the graph indicate the minimum, maximum, and average observed for the 100 values for N_P. With this limited range for N_F we can still use 110 of the 162 versions in the benchmark set, whereas for $N_F \leqslant 100$, we can only use 60. We verified that for $N_F \leqslant 15$, for which we can use 128 versions, the results are essentially the same.

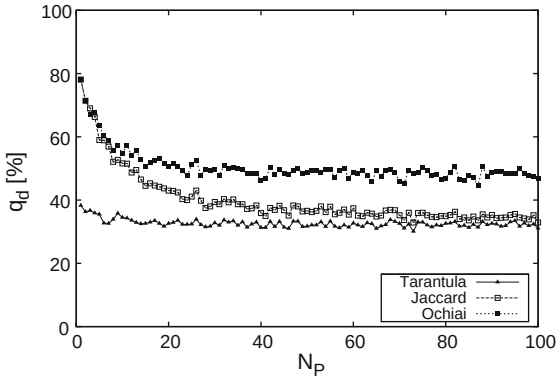

Fig. 7. Impact of N_P on q_d for `print_tokens2` v1, and $N_F = 6$.

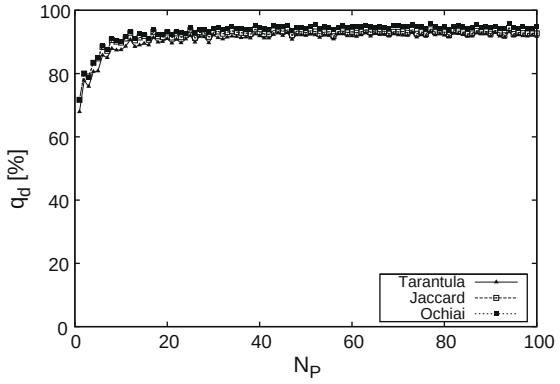

Fig. 8. Impact of N_P on q_d for `schedule` v2, and $N_F = 6$.

A first conclusion that we draw from Fig. 6 is that overall, adding failed runs improves the accuracy of the diagnosis. However, the benefit of having more than around 10 runs is marginal on average. In addition, because the measurements for varying N_P show little scattering in the projection, we can conclude that on average, N_P has little influence.

Inspecting the results for the individual program versions confirms our observation that adding failed runs consistently improves the diagnosis. However, although the effect does not show on average, N_P can have a significant effect on q_d for individual runs. As shown in Fig. 5, this effect can be negative or positive. This shows more clearly in Figs. 7 and 8, which contain cross sections of the graphs in Fig. 5 at $N_F = 6$. To factor out any influence of N_F, we have created similar cross sections at the maximum number of failed runs. Across the entire benchmark set, we found that the effect of adding more passed runs stabilizes around $N_P = 20$.

Returning to the influence of the similarity coefficient once more, Figs. 7 and 8 further indicate that the superior performance of the Ochiai coefficient is consistent also for varying numbers of runs. We have not plotted q_d for the other coefficients in Fig. 5, but we verified this observation for all program versions, with N_P and N_F varying from 1 to 100.

From our experiments on the impact of the number of runs we can draw the following conclusions. All of these are in the context of our benchmark set, which has the important characteristic that most faults involve a single location in the source code. First, including more failed runs is safe because the accuracy of the diagnosis either improves or remains the same. This is observed due to the fact that failed runs add evidence about the block that is causing the program to fail, and hence causing it to move up in the ranking. Our results show that the optimum value for N_F is in the order of 10 runs. To what extent this result depends on

characteristics of the fault or program is subject to further investigation. Second, while stabilizing around $N_P = 20$, the effect of including more passed runs is unpredictable, and may actually decrease q_d. In fact, q_d decreases only if the faulty block is touched often in passed runs, as spectrum-based fault localization works under the assumption that if a block is touched often in passed runs, it should be exonerated. Besides, a large number of runs can apparently compensate weak error detection quality: even for small q_e, a large amount of runs provides sufficient information for good diagnostic accuracy, as shown in Fig. 4. Lastly, the number of runs has no influence on the superiority of the Ochiai coefficient.

7. An industrial case study

While the benchmark problems are well suited for studying the influence of parameters such as the similarity coefficient, and the quality and quantity of the observations, they give little indication on the accuracy of spectrum-based fault localization for large-scale codes, and the kind of problems that are encountered in practice. For this reason, in this section and the next we report our experience with implementing SFL for an industrial software product, namely the control software of a particular product line of hybrid analog/digital LCD television sets. These experiments are done in the context of the TRADER project (Embedded Systems Institute, 2009), whose goal is to improve the user-perceived reliability of consumer electronics systems with embedded software. In this section we describe the experimental platform, and our implementation of SFL for it. The actual experiments are described in Section 8.

7.1. Platform

One of the products of the main industrial partner in the TRADER project, NXP Semiconductors, is the TV520 platform for building hybrid analog/digital LCD television sets, which in turn serves as the basis for televisions sets manufactured by NXP's customers. The TV520 platform comprises one or two proprietary CPUs for audio and video signal processing, plus a MIPS CPU running the control software (under Linux). More details on TV520 can be found on the NXP website (NXP).

Our experiments are performed on development versions of television sets based on TV520. All problems that we diagnosed are in the control software of the sets, which is responsible for tasks such as

- decoding the remote-control input,
- navigating the on-screen menu,
- coordinating the hardware (e.g., the tuner),
- coordinating the audio and video processing on the proprietary CPUs, based on an analysis of the signals,
- teletext[3] decoding, viewing, and navigation.

The control software comprises roughly one million lines of C code (configured from a much larger code base of software components), and 150,000 blocks, as defined in Section 2.2. Porting the SFL tooling to the proprietary CPUs is part of our ongoing work.

7.2. Space efficiency

In a regular computing environment, storing all program spectra for a series of test cases is no problem, and this is how we implemented spectrum-based fault localization for the experiments reported in the previous sections. In the embedded domain, however, memory is typically a scarce resource, and storing all spectra to be post-processed at diagnosis time is usually not an option. As an indication, the 64 MB that is available for our experiments is shared with the application binaries and variables, and depending on the usage scenario, only approximately 10 spectra can be stored in the remaining space, using a byte for each flag. Bytes are the smallest data unit that can be transferred to/from memory directly. There is a potential problem when different bytes in the same word are updated from within different threads. However, due to an extra layer of virtual threads running on top of the operating systems threads, the probability that two blocks that are mapped on the same word are executed in parallel is negligible in practice, and we chose to ignore this issue.

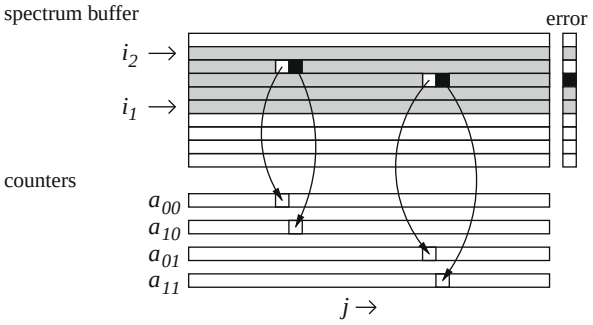

Fig. 9. Data structures.

Fortunately, although the available storage space is quite limited, the set of spectra that a diagnosis is based on contains much more information than needed, and can easily be compacted at run-time. In the end, the only information that is needed to generate the ranking are the four sets of counters $a_{00}(j), \ldots, a_{11}(j)$, introduced in Section 2.3, and the space required to store these is linear in the size of the program, not in the number of test cases. To avoid having to store the actual spectra, we can update the counters right after a run has finished, and the passed/failed verdict has become available:

- For a passed run, and all blocks j: if block j has been active, increment $a_{10}(j)$, otherwise increment $a_{00}(j)$.
- For a failed run, and all blocks j: if block j has been active, increment $a_{11}(j)$, otherwise increment $a_{01}(j)$.

After thus having processed the program spectrum of a passed or failed run, the spectrum itself can be discarded. Any time after processing at least one failed run, the diagnosis can be performed by evaluating the similarity coefficient of choice for all blocks, and by ranking the blocks based on their calculated coefficients.

In our implementation, we use a small circular buffer to cache recently recorded spectra until they can be processed on a low priority thread (see Fig. 9). Two pointers cycle through this buffer: i_1, pointing to the current spectrum, where the system activity is being recorded, and i_2, pointing to the first spectrum whose contributions must still be added to the sets of counters $a_{00}(j), \ldots, a_{11}(j)$. While in theory, spectra can be overwritten if insufficient idle time is available for processing spectra of previous runs, this is not a problem in our experiments, and spectra are cleaned up almost immediately after they are cached by advancing i_1. However, if runs are delimited automatically, it may be necessary to tune the rate at which spectra are generated to the size of the buffer.

7.3. Implementation

As we describe in Section 3.2, the spectra are obtained via instrumentation. Compared to the experiments on the benchmark set, additional, but nonfundamental difficulties that are encountered in the NXP development environment are the following.

[3] A standard for broadcasting information (e.g., news, weather, TV guide) in text pages, popular in Europe.

- Although the code base is ANSI-C compliant, several GNU extensions that are inserted by the preprocessor cannot be handled by the `Front` parser, which is not normally applied to preprocessed code, and require a workaround.
- Parallel build threads must be disabled, to ensure that unique numbers are assigned to blocks.
- The possibilities for incremental builds are limited because we have to set a maximum number of blocks.
- In addition to constraints on the available memory, discussed in the previous section, CPU time is also a scarce resource, and the TV520 architecture imposes various timing constraints on different activities.

Regarding the last point, the sorting task that is involved in ranking the approximately 150,000 blocks does not violate any of the timing constraints. Although the block-level instrumentation noticeably slows down the operation of the TV, enough CPU idle time is available to support the extra load on the MIPS. This is not the case for the proprietary CPUs though, and we have not yet found a practicable solution to instrumenting the signal processing software.

The spectrum bookkeeping illustrated in Fig. 9 is implemented in a small software component that is added to the control software. Communication with this component is via standard I/O, using a PC and terminal emulator connected to the TV set. On the terminal we can enter commands such as

- start a new run, and mark the previous run as passed,
- start a new run, and mark the previous run as failed,
- select a particular similarity coefficient,
- calculate the diagnosis, and print the n locations at the top of the ranking,

where we consider a "run" to be any given period of activity of the system. We used the Jaccard and Ochiai coefficients, introduced in Section 2.3, but because of the highly accurate (manual) error detection information involved in these experiments, the diagnoses were essentially the same. This confirms the observation made in Section 5.3, that the performance of the former coefficient approaches that of the latter as the quality of the error detection information improves (see Fig. 4). Because of the superior performance of these two coefficients, we have not included the Tarantula coefficient in this case study.

8. Experiments

We diagnosed four problems that were encountered during the development of television sets based on the TV520 platform. In the same way as the known location of a fault can be used to evaluate the quality of the diagnosis for the benchmark experiments, the location of the repairs that were made can be used as an indication of diagnostic quality here.

Selecting the problems to use for this case study was more difficult than we anticipated when planning the experiment. Enough problem reports and repairs are available, but in many cases we could not reproduce the problem, for reasons such as

- the source tree having been removed from the version repository,
- the version of the hardware for which the problem manifested itself no longer being available,
- the problem residing in the streaming code on the proprietary CPUs, for which our tooling is not yet available, and
- the problem being hard to reproduce in itself.

In Sections 8.1–8.4 below we give a description of the four problems, our approach to diagnose them, and the result delivered by SFL. The quality of these diagnoses is discussed in Section 8.5.

8.1. NVM corrupted

8.1.1. Problem description

The TV sets that we used in our experiments contain a small amount of non-volatile memory (NVM), whose contents are retained without the set being powered. In addition to storing information such as the last channel watched, and the current sound volume, the NVM contains several parameters of a TV's configuration, for example to select a geographical region. These parameters can be set via the so-called service menu, which is not normally accessible to the user.

A subset of the parameters stored in NVM are so important for the correct functioning of the set, that it has been decided to implement them with triple redundancy and majority voting. This provides a basic protection against memory corruption, since at least two copies of a value have to be corrupted to take effect. The problem that we analyze here entails that two of the three copies of redundant NVM parameters are not updated when changes are made via the service menu.

8.1.2. Approach

To diagnose this problem, we extend our diagnosis component such that once per second, it starts a new run. Knowing that the problem manifests itself in NVM, we add a consistency check on the redundant items to characterize the runs as passed or failed. The runs are taken from a simple scenario where we first activate the general menu-browsing functionality, to exonerate that part of the code. Then we make several changes to nonredundant NVM parameters, before changing the value of a redundant parameter, and performing the diagnosis based on a single failed run. The number of passed runs depends on the time to run the scenario, which is in the order of one or 2 min.

8.1.3. Diagnosis

In the ranking produced by SFL, 96 blocks have the highest similarity to the error vector. These blocks are in 10 files, one of which is part of the NVM stack, making this the obvious place to continue the diagnosis. Inside its component, this files' functions access modules for normal, and redundant access to NVM, which confirms that the problem is in this area. The bug, however, resides in a routine that is called at system initialization time to retrieve the status (redundant, or not), of the individual NVM items to populate a table describing the NVM layout. Since this routine is always used at initialization, while the problem does not yet manifest itself, there is no way that SFL can associate it with the failures that occur later on, so in this case, the actual diagnosis is indirect at best. In general, SFL based on block-hit spectra cannot be expected to directly locate data-dependent faults, or faults in code that is always executed. However, debugging is usually an iterative process, and in this sense, zooming in on the code that accesses the table describing the NVM layout can still be seen as a valuable suggestion for where to look next. In general, hit spectra of definition-use pairs (Harrold et al., 2000) may provide more relevant information to diagnose data-dependent problems, but in this particular case the diagnosis is hindered because the fault occurs at initialization, and is always executed.

8.2. Scrolling bug

8.2.1. Problem description

The TV has several viewing modes to watch content with different aspect ratios. In 16:9 viewing mode, only part of a 4:3 image is

displayed on screen, and the "window" through which the image is watched can be positioned using the directional buttons on the remote control (scrolling). The problem considered here entails that after scrolling in a vertical direction, switching to dual-picture mode and back re-centers the screen. Continuing to scroll after this re-centering has occurred makes the screen jump back to the position that it had before entering dual-picture mode, and scrolling continues from that position. It should be noted that in dual-picture mode, one of the two screen halves displays the original picture, and the other half displays teletext.

8.2.2. Approach

To diagnose this problem, we rerun the above scenario as follows:

1. enter 4:3 mode, and switch to dual-picture and back,
2. enter 16:9 mode, and scroll up and down,
3. demonstrate the problem in both vertical scrolling directions,
4. switch to teletext and back.

The runs are defined by the various actions such as scrolling, selecting the viewing modes, etc., leading to approximately 20 runs, two of which are marked as failed. Because we do not know where exactly the problem occurs, the two failed runs both involve two key-presses: one to switch back from dual-picture mode (which re-centers the picture), and another to scroll (which makes the picture jump).

8.2.3. Diagnosis

The repair of this problem involves three locations, and one of these is right on top of the ranking produced by SFL, sharing the first place with four other blocks. The second location is in the top 13 of the ranking, with the second-highest similarity, but the third location is much further down: so many other blocks have the same similarity that effectively, SFL cannot find it. However, all three fixes are in the same file, and the third fix is a natural extension of the other two.

Given the small number of locations that have to be examined before we hit two of the locations where this problem is repaired, we consider this diagnosis quite accurate. However, the last step of the scenario, where we exonerate the teletext functionality, appears to be essential for getting a good result consistently. Why the first step of the scenario, which also activates teletext in one of the two screen halves, is not sufficient, is still subject of further investigation.

8.3. Pages without visible content

8.3.1. Problem description

In the particular product line where this problem manifests itself, it is possible to highlight a word on a teletext page, and then search the whole database of teletext pages for the current channel for other occurrences of that word. However, the teletext standard provides for pages with invisible content, through which, for example, certain control messages can be broadcast: the characters are there, but a special flag marks them invisible to the user. The problem that we investigate here entails that the word search function also finds occurrences of a word on invisible pages, and that hitting such an occurrence locks up the search functionality.

8.3.2. Approach

To diagnose this problem we use a scenario where we activate the relevant teletext browsing functionality, including the word search, and where we start new runs after, for example, changing the page, navigating to words of interest, and finding new occurrences of those words. We manually mark runs as passed or failed depending on whether the TV enters the locked-up state, or not. In the end, we could not improve the diagnosis by using more than a single failed run, and around 10 passed runs.

8.3.3. Diagnosis

Because this particular problem is still under investigation at NXP, it is not possible to evaluate the quality of the diagnosis based on the locations of the fixes. However, several code locations at the top of the ranking generated by SFL involve statements whose execution depend on whether a page contains invisible content. We expect that this could well serve as a reminder that pages can have invisible content, and that this information provides a good suggestion on the nature of a possible fix.

8.4. Repeated tuner settings

8.4.1. Problem description

Some broadcasters' signals contain regional information in a protocol that is recognized by many television sets, and which specifies, for example, a preferred order for the television channels. The problem that we investigate here entails that after an installation (finding all channels) is performed in presence of this regional information, tuning twice in a row to an analog signal at the same frequency results in a black screen.

8.4.2. Approach

There are two ways in which the same frequency can be set repeatedly: by entering the same channel number on the remote-control twice, and by switching from an analog channel to an external video source (which does not change the tuner frequency) and back. We run a scenario where we demonstrate the problem in both ways, on both a single-digit and a two-digit channel, and where we also include several examples of changing the channel without triggering the problem. The general strategy is to start a new run after each channel change, and to mark the previous run as passed or failed depending on whether the problem manifests itself, or not, resulting in 4 failed runs, and depending on the exact scenario, around 15 passed runs.

8.4.3. Diagnosis

The repairs for this problem involve modifications in 13 code blocks, all in the same file. Although none of the exact locations appears at a high position in the ranking generated by SFL, depending on the exact scenario, typically 11 other blocks are found at the highest level of similarity, 10 of which are from the file where the problem has been repaired, making this the obvious place to start debugging. Given the fact that over 1800 C files are involved in the build, with approximately a dozen files related to low-level tuner functionality, this can be considered a reasonably accurate diagnosis. We have not been able to exploit the information that the problem only occurs after an installation in presence of the regional information.

8.5. Evaluation

These experiments demonstrate that the integration of SFL in an industrial software development process is feasible: although much more time was invested in these experiments, the estimated costs for an analysis are 2 h for building the application binary of the control software with instrumentation enabled, plus another few hours for running the experiments, and analyzing the data. If automated error detection is required, as we described for the NVM corruption in Section 8.1, some more time must be reserved for writing the special-purpose error detectors. In any case, running the analysis within a working day is feasible, and in some debugging scenario's this can be a sensible investment. In addition

Table 4
Diagnostic accuracy for the industrial test cases; total numbers of blocks and files are 150,000 and 1800, respectively.

Case	Estimated q_d (%)	Inspect
NVM corrupt	99.96*	96 blocks, 10 files
Scrolling bug	>99.99	5 blocks
Invisible pages	>99.99	12 blocks
Tuner problem	99.97	2 files

* Indirect, see Section 8.1.

to that, opportunities for integrating SFL with automated testing schemes still have to be explored.

Of the four cases that we have considered thus far, one diagnosis is quite good (the scrolling bug), and in the other three cases, SFL provides a useful suggestion, where in the case of the NVM corruption, this suggestion is indirect because of the data dependencies involved. In Table 4 we give an estimate of the quality of the diagnosis in terms of q_d, as defined in Section 3.3. For the NVM corruption, this is based on the 96 blocks and 10 files on top of the ranking, as described in Section 8.1. For the scrolling bug we use the highest-ranking location where a repair has been made. In case of the teletext lock-up at invisible pages, we use the rank of the block that directs our attention to the flag for invisible content. For the tuner problem, the estimate is based on two out of approximately 1800 files, instead of blocks, as discussed in Section 8.4. Because of the high percentages involved, we have also included an indication of the amount of code that must be investigated, based on the number of blocks with an equal or higher calculated similarity coefficient.

While the estimates in Table 4 are debatable, the experiments demonstrate that spectrum-based fault localization scales well, and that it can be applied as a practicable tool in industrial software development. Note that while the estimated q_d values in Table 4 clearly indicate the power of SFL on large codes, these numbers are not indicative for the added value for an experienced developer. For example, as we discussed in Section 8.4, an NXP developer would immediately concentrate on the dozen of files related to low-level tuner functionality, lowering q_d to just over 95% of files that do not have to be investigated. Nevertheless, SFL confirms such a decision as well as improves on it in terms of q_d.

These experiments are part of the ongoing work to transfer the techniques that have been developed in the TRADER project to NXP Semiconductors. They were initiated after a first successful trial (Zoeteweij et al., 2007), with the purpose of demonstrating the technique on a number of problems that have been filed in recent development history. In the end, we expect that the transfer will lead to instrumentation for SFL being included as a standard option in the build scripts of NXP Semiconductors, accompanied by guidelines for when and how to use the technique.

9. Related work

Program spectra themselves were introduced in (Reps et al., 2000), where hit spectra of intra-procedural paths are analyzed to diagnose year 2000 problems. The distinction between count spectra and hit spectra is introduced in (Harrold et al., 2000), where several kinds of program spectra are evaluated in the context of regression testing.

In Section 1, we already mentioned three practical diagnosis/debugging tools (Chen et al., 2002; Dallmeier et al., 2005, 2002) that are essentially based on spectrum-based fault localization. Pinpoint (Chen et al., 2002) is a framework for root cause analysis on the J2EE platform and is targeted at large, dynamic Internet services, such as web-mail services and search engines. The error detection is based on information coming from the J2EE framework, such as caught exceptions. The Tarantula tool (Jones et al., 2002) has been developed for the C language, and works with statement hit spectra. AMPLE (Dallmeier et al., 2005) is an Eclipse plug-in for identifying faulty classes in Java software. However, although we have recognized that it uses hit spectra of method call sequences, we did not include its weight coefficient in our experiments because the calculated values are only used to collect evidence about classes, not to identify suspicious method call sequences.

Diagnosis techniques can be classified as white box or black box, depending on the amount of knowledge that is required about the system's internal component structure and behavior. An example of a white box technique is model-based diagnosis (see, e.g., de Kleer and Williams, 1987), where a diagnosis is obtained by logical inference from a formal model of the system, combined with a set of run-time observations. White box approaches to software diagnosis exist (see, e.g., Wotawa et al., 2002), but software modeling is extremely complex, so most software diagnosis techniques are black box. Since the technique studied in this paper requires practically no information about the system being diagnosed, it can be classified as a black box technique.

Examples of other black box techniques are Nearest Neighbor (Renieris and Reiss, 2003), dynamic program slicing (Agrawal et al., 1993; Liu et al., 2005), Delta Debugging (Zeller, 2002), and Δ-slicing (Groce, 2004). The Nearest Neighbor technique first selects a single failed run, and computes the passed run that has the most similar code coverage. Then it creates the set of all statements that are executed in the failed run but not in the passed run. Dynamic program slicing narrows down the search space to the set of statements that influence a value at a program location where the failure occurs (e.g., an output variable). Sober is a statistical debugging tool which analysis traces fingerprints and produces a ranking of predicates by contrasting the evaluation bias of each predicate in failing cases against those in passing cases. Delta Debugging compares the program states of a failing and a passing run, and actively searches for failure-inducing circumstances in the differences between these states. In Gupta et al. (2005) Delta Debugging is combined with dynamic slicing in four steps: (1) Delta Debugging is used to identify the minimal failure-inducing input; step (2) computes the forward dynamic slice of the input variables obtained in step 1; (3) the backward dynamic slice for the failed run is computed; (4) finally it returns the intersection of the slices given by the previous two steps. This set of statements is likely to contain the faulty code. Δ-slicing is a model-based automated approach, based on distance metrics for program executions traces, for assisting developers to isolate faulty parts of a program. Basically, with the help of a (correct) counter-example the technique tries to isolate the minimal difference that induces a transition from a correct to a faulty execution trace.

Except for Pinpoint (Chen et al., 2002), all other researchers used a reference program as error detector to study their techniques. Although this program is typically available during regression testing, there are several situations where a reference program is simply not there. This is the case when a new system or new functionality is developed. The experiments reported in Section 8 are representative for this situation. In this case, errors were indicated manually, or by an assert-like check that was added to the software under diagnosis. Another example of a situation where no reference system is available is when a product has passed the testing stage, and is released on the market. For this situation, in (Abreu et al., 2008a; Abreu et al., 2008b) generic program invariants were successfully used as error detectors for spectrum-based fault localization. The authors concluded that the diagnostic quality of this fully automated approach equals the one obtained when the reference program is used. The investigated program invariants are the bitmask (Abreu et al., 2008a), range (Abreu et al., 2008a;

Abreu et al., 2008b), and the Bloom filter (Abreu et al., 2008b) invariants. Given that the performance overhead of generic program invariants is small, they are attractive to be used in the context of resource constraint systems, as the one we used in our industrial experiments, allowing to create self-diagnosing systems.

Regarding our observation that for the benchmark faults in the Siemens set and `space` program, the diagnostic quality does not change significantly when using more than 20 passed runs and 10 failed runs, in (Yu et al., 2008) the effect of several test-suite reduction strategies on the accuracy of spectrum-based fault localization is studied. The SFL variants taken into account in this study include Tarantula, and the Jaccard and Ochiai coefficients. To our knowledge, no other evaluations of the diagnostic quality of similarity coefficients in the context of varying observation quality and quantity exist. Furthermore, we are not aware that any of the other techniques mentioned above have successfully been applied for diagnosing software faults in resource-constrained systems.

10. Conclusions and future work

Reducing fault localization effort greatly improves the test–diagnose–repair cycle. In this paper, we have investigated the influence of different parameters on the accuracy of the diagnosis delivered by spectrum-based fault localization. Our starting point was a previous study on the influence of the similarity coefficient, which indicated that the Ochiai coefficient, known from the biology domain, can give a better diagnosis than eight other coefficients, including those used by the Pinpoint (Chen et al., 2002) and Tarantula (Jones and Harrold, 2005) tools.

By varying the quality and quantity of the observations on which the fault localization is based, we have established this result in a much wider context. We conclude that the superior performance of the Ochiai coefficient in diagnosing single faults in the Siemens set and the `space` program is consistent, and does not depend on the quality or quantity of observations. We expect that this result is also relevant for the Tarantula tool, whose analysis is essentially the same as ours.

In addition, we found that even for the lowest quality of observation that we applied ($q_e = 1\%$, corresponding to a highly ambiguous error detection), the accuracy of the diagnosis is already quite useful: around 80% for all the programs in our benchmark set, which means that on average, only 20% of the code remains to be investigated to locate the fault. Furthermore, we conclude that while accumulating more failed runs only improves the accuracy of the diagnosis, the effect of including more passed runs is unpredictable. With respect to failed runs we observe that only a few (around 10) are sufficient to reach near-optimal diagnostic performance. Adding passed runs, however, can both improve or degrade diagnostic accuracy. In either case, including more than around 20 passed runs has little effect on the accuracy. The fact that a few observations can already provide a near-optimal diagnosis enables the application of spectrum-based fault localization methods within continuous (embedded) processing, where only limited observation horizons can be maintained.

In addition to our benchmark studies on the Siemens set and `space`, we have also evaluated spectrum-based fault localization on a large-scale code in the area of embedded software in consumer electronics. These experiments have convinced us that SFL scales well, and that it can be applied as a useful tool in an industrial software development environment.

In future work, we plan to study the influence of the granularity (statement, function level) of program spectra on the diagnostic accuracy of spectrum-based fault localization. Furthermore, we intend to investigate the accuracy improvement of integrating static and dynamic program slicing (see, e.g., Agrawal et al., 1993) within our technique. Finally, our study was conducted using C programs with a single fault. Regarding the latter restriction, Spectrum-based fault localization naturally extends to the multiple-fault case by iteratively applying the technique until all faults are repaired (see, e.g., Jones et al., 2002). However, we recently applied model-based diagnosis techniques to derive multiple-fault explanations directly from program spectra and pass/fail information (see Abreu et al. (2008c) for an account of the approach), and further investigation is in progress to compare the debugging effort implied by the two approaches on multiple-fault programs. Regarding the restriction to the C programming language, we expect the technique to work well on other programming paradigms and languages. From our current set of tools, the primary task would be to port the instrumentation step, and work is currently underway to implement it for the LLVM platform (Lattner and Adve, 2004), which has front-ends for various programming languages. This tool will be made available from http://www.fdir.org/sfl.

Acknowledgements

We gratefully acknowledge the fruitful discussions with our TRADER project partners from Design Technology Institute, Embedded Systems Institute, IMEC, Leiden University, NXP Semiconductors, Philips TASS, Philips CE, and Twente University.

References

Abreu, R., Zoeteweij, P., van Gemund, A.J.C., 2006a. Program spectra analysis in aembedded systems: a case study. Technical Report TUD-SERG-2006-007, Delft University of Technology. <http://arxiv.org/abs/cs.SE/0607116>.

Abreu, R., Zoeteweij, P., van Gemund, A.J.C., 2006b. An evaluation of similarity coefficients for software fault localization. In: Proceedings of PRDC'06, Riverside, CA, USA, December 2006, IEEE Computer Society, pp. 39–46.

Abreu, R., Zoeteweij, P., van Gemund, A.J.C., 2007. On the accuracy of spectrum-based fault localization. In: Proceedings TAIC PART'07, Windsor, UK, September 2007, IEEE Computer Society, pp. 89–98.

Abreu, R., González, A., Zoeteweij, P., van Gemund, A.J.C., 2008a. Automatic software fault localization using generic program invariants. In: Proceedings SAC'08, Fortaleza, Ceará, Brazil, March 2008, ACM Press, pp. 712–717.

Abreu, R., González, A., Zoeteweij, P., van Gemund, A.J.C., 2008b. On the performance of fault screeners in software development and deployment. In Proceedings ENASE'08, Funchal, Madeira, Portugal, May 2008, INSTICC Press, pp. 123–130.

Abreu, R., Zoeteweij, P., van Gemund, A.J.C., 2008c. An observation-based model for fault localization. In: Proceedings of WODA'08, Seattle, WA, USA, July 2008, ACM Press, pp. 64–70.

Agrawal, H., DeMillo, R.A., Spafford, E.H., 1993. Debugging with dynamic slicing and backtracking. Software – Practice and Experience 23 (6), 589–616.

Augusteijn, L, 2002. Front: a front-end generator for Lex, Yacc and C. Release 1.0. <http://front.sourceforge.net/>.

Avižienis, A., Laprie, J.-C., Randell, B., Landwehr, C.E., 2004. Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing 1 (1), 11–33.

Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E., 2002. Pinpoint: problem determination in large, dynamic internet services. In: Proceedings of DSN'02, Washington, DC, USA, June 2002, IEEE Computer Society, pp. 595–604.

Dallmeier, V., Lindig, C., Zeller, A., 2005. Lightweight defect localization for Java. In: Proceedings of ECOOP'05, Glasgow, UK, July 2005, LNCS 3568, Springer-Verlag, pp. 528–550.

da Silva Meyer, A., Franco Garcia, A.A., Pereira de Souza, A., Lopes de Souza Jr., C., 2004. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (*Zea mays* L). Genetics and Molecular Biology 27 (1), 83–91.

de Kleer, J., Williams, B.C., 1987. Diagnosing multiple faults. Artificial Intelligence 32 (1), 97–130.

Do, H., Elbaum, S.G., Rothermel, G., 2005. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. Empirical Software Engineering: An International Journal 10 (4), 405–435.

Embedded Systems Institute, 2009. Trader project website. <http://www.esi.nl/trader/>.

Groce, A., 2004. Error explanation with distance metrics. In: Proceedings TACAS'04, Barcelona, Spain, March 2004, Springer-Verlag, pp. 108–122.

Gupta, N., He, H., Zhang, X., Gupta, R., 2005. Locating faulty code using failure-inducing chops. In: Proceedings of ASE'05, Long Beach, CA, USA, November 2005, ACM Press, pp. 263–272.

Hailpern, B., Santhanam, P., 2002. Software debugging testing and verification. IBM Systems Journal 41 (1), 4–12.
Harrold, M.J., Rothermel, G., Sayre, K., Wu, R., Yi, L., 2000. An empirical investigation of the relationship between fault-revealing test behavior and differences in program spectra. Journal of Software Testing Verification and Reliability 10 (3), 171–194.
Hutchins, M., Foster, H., Goradia, T., Ostrand, T., 1994. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In: Proceedings ICSE'94, Sorrento, Italy, May 1994, IEEE Computer Society, pp. 191–200.
Jain, A.K., Dubes, R.C., 1988. Algorithms for Clustering Data. Prentice-Hall.
Jones, J.A., Harrold, M.J., 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of ASE'05, Long Beach, CA, USA, November 2005, ACM Press, pp. 273–282.
Jones, J.A., Harrold, M.J., Stasko, J., 2002. Visualization of test information to assist fault localization. In: Proceedings of ICSE'02, Orlando, Florida, USA, May 2002, ACM Press, pp. 467–477.
Lattner, C., Adve, V., 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of CGO'04, San Jose, CA, USA, March 2004, IEEE Computer Society, pp. 75–88.
Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S.P., Sober: statistical model-based bug localization. In Proc. ESEC/FSE'05, Lisbon, Portugal, September 2005, pp. 286–295, ACM Press, 2005.
Morell, L.J., 1990. A theory of fault-based testing. IEEE Transactions on Software Engineering 16 (8), 844–857.
NXP Semiconductors website. <http://www.nxp.com>.
Renieris, M., Reiss. S.P., 2003. Fault localization with nearest neighbor queries. In: Proceedings of ASE'03, Montreal, Canada, October 2003, IEEE Computer Society, pp. 30–39.
Reps, T., Ball, T., Das, M., Larus. J., 1997. The use of program profiling for software maintenance with applications to the year 2000 problem. In: Proceedings of ESEC/FSE'97, Zurich, Switzerland, September 1997, LNCS 1301, Springer-Verlag, 1997, pp. 432–449.
Voas, J., 1992. PIE: a dynamic failure based technique. IEEE Transactions on Software Engineering 18 (8), 717–727.
Wotawa, F., Stumptner, M., Mayer, W., 2002. Model-based debugging or how to diagnose programs automatically. In: Proceedings of IEA/AIE'02, Cairns, Australia, June 2002, LNCS 2358, Springer-Verlag, pp. 746–757.
Yu, Y., Jones, J.A., Harrold, M.J., 2008. An empirical study of the effects of test-suite reduction on fault localization. In: Proceedings of ICSE'08, Leipzig, Germany, May 2008, ACM Press, pp. 201–210.
Zeller, A., 2002. Isolating cause-effect chains from computer programs. In: Proceedings of FSE'02, Charleston, SC, USA November 2002, ACM Press, pp. 1–10.
Zoeteweij, P., Abreu, R., Golsteijn, R., van Gemund, A.J.C., 2007. Diagnosis of embedded software using program spectra. In: Proceedings of ECBS'07, Tucson, AZ, USA, March 2007, IEEE Computer Society, pp. 213–218.

Rui Abreu is a PhD. student at the Embedded Software Lab within the Software Engineering Research Group at Delft University of Technology. He holds an MSc. in Computer Science and Systems Engineering from Minho University, Portugal. Through his thesis work at Siemens R&D Porto, and professional internship at Philips Research, he acquired industrial experience in the area of embedded systems.

Peter Zoeteweij works in the Software Engineering Research Group at Delft University of Technology. He holds an MSc. from Delft University of Technology, and a PhD. from the University of Amsterdam, both in computer science. Before his PhD., Peter worked for several years as a software engineer for Logica (now LogicaCMG), mainly on software for the oil industry.

Rob Golsteijn holds an MSc. in Computing Science from Eindhoven University of Technology and completed the two years' post-graduate Software Technology program from the Stan Ackermans Institute. Rob now works for NXP, formerly known as Philips Semiconductors, and has experience in embedded software development of television platforms and products. Rob is currently working as a member of an industrial research project focusing on reliability of resource-constrained consumer devices.

Arjan J.C. van Gemund holds a BSc. in physics, and an MSc. (cum laude) and PhD. (cum laude) in computer science, all from Delft University of Technology. He has held positions at DSM and TNO, and currently serves as a full professor at the Electrical Engineering, Mathematics, and Computer Science Faculty of Delft University of Technology, heading the Embedded Software Lab within the Software Engineering Research Group.