Software Fault Diagnosis

Peter Zoeteweij*, Rui Abreu, and Arjan J.C. van Gemund

Embedded Software Lab,
Faculty of Electrical Engineering, Mathematics, and Computer Science,
Delft University of Technology,
P.O. Box 5031, 2600 GA Delft, The Netherlands

*corresponding author: p.zoeteweij@tudelft.nl

Abstract. This tutorial paper gives an overview of automated diagnosis applied to software faults. The emphasis is on a particular technique called *spectrum-based fault localization*, which is well-suited for diagnosing software systems, and which can easily be integrated with existing testing schemes. We discuss applications of the technique, including the specific application domain of embedded software, and provide pointers to recent research on factors that influence its diagnostic accuracy. In addition, we give instructions for quickly getting started with applying spectrum-based fault localization to existing projects.

1 Introduction

The amount of source code underlying the systems that we use on a day-to-day basis is constantly growing. Combined with a practically constant rate of faults per line of code, this implies that system dependability is decreasing. Automated diagnosis techniques are an important means to counter this trend. They serve to localize the faults that are the root causes of discrepancies between expected and observed behavior of systems, and as such they are a natural companion to testing efforts in the software development cycle, which aim at exposing such discrepancies. In this context, automated diagnosis can reduce the effort spent on manual debugging, which shortens the test-diagnose-repair cycle, and can hence be expected to lead to more reliable systems, and a shorter time-to-market.

Outside the software development cycle, automated diagnosis techniques can be used in maintenance, and can serve as the basis for (automated) recovery strategies. As such, they are a vital ingredient of dependable autonomic systems.

This tutorial paper aims to give an overview of automated diagnosis applied to software faults. The emphasis is on a particular technique called *spectrum-based fault localization* (SFL), which is well-suited for diagnosing software systems, and which can easily be integrated with existing testing schemes. We start by taking a high-level view on the diagnosis problem. Central to this view is the notion of a model of a system, which serves to define its intended behavior, and may contain additional information about its composition and operation. In the context of this high-level view, we compare SFL to model-based diagnosis (MBD). While MBD has successfully been applied for diagnosing complex

mechanical systems, its application to software has proven to be difficult. Comparing SFL and MBD is useful for understanding the possibilities and limitations of both methods.

The plan for the paper is as follows. In Sec. 2 we introduce the principles underlying the diagnosis problem. In Sec. 3 we introduce SFL. In Sec. 4 we provide some suggestions for quickly getting started with SFL experiments. In Sec. 5 we give an overview of recent research on several factors that influence the diagnostic accuracy of SFL. In Sec. 6 we discuss the applicability of SFL for in the area of embedded software. In Sec. 7 we discuss related work. This includes both existing diagnosis / debugging systems that apply SFL, and other approaches to software fault diagnosis. We conclude in Sec. 8.

2 Principles

Central to a discussion of diagnosis are the notions of failure and fault. A *failure* is a discrepancy between expected and observed behavior, and a *fault* is a property of the system that causes such a discrepancy. The notion of an *error* is sometimes used to indicate a system state that potentially leads to a failure, but for our purposes the distinction between failure and error is largely artificial, and depends only on what can be observed and what has been specified. See [5] for an in-depth discussion of these concepts.

As an illustration, consider the digital circuit in Fig. 1, which is composed of three logical inverters i_1, i_2, and i_3. If it operates correctly, an inverter changes an input signal 0 to an output signal 1, and an input signal 1 to an output signal 0. The expected behavior of the system is that the input signal x and the output signals y_1 and y_2 are equal. This allows us to detect a failure if any of these three signals differs from the other two. An example of a fault in this system is that the inverter i_2 is broken, and always produces an output signal 0 (commonly referred to as *stuck at zero*). This fault will only manifest itself as a failure for input $x = 1$: for $x = 0$ the circuit still behaves as expected.

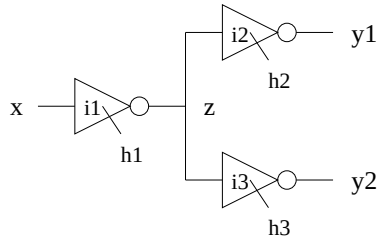

Fig. 1. Example circuit with three inverters

The purpose of diagnosis is to identify the fault that is the root cause of a failure. As such, any diagnosis is based on a behavioral model that captures the

expected behavior. The diagram of Fig. 1 is an example of such a model, but for the purpose of defining the expected behavior, it may well be replaced by the equations $x = y_1 = y_2$. Although many forms of formal models exist, the model of the expected behavior need not be formal, nor does it have to be made explicit: it may simply exist in the mind of a human user of a system.

2.1 Model-Based Diagnosis

In addition to identifying failures, a model of a system may allow us to reason about possible causes of these failures. In this case, information on the internal composition of a system is essential. Returning to the three-inverter example, the expected behavior of the system is fully captured by the equations $x = y_1 = y_2$, but the additional information that the system is composed of three inverters will help us identify possible causes for observed deviations from this expected behavior. Suppose, for example, that we have $x = 1$, $y_1 = 0$, and $y_2 = 1$. One possible explanation for this unexpected behavior is that inverter i_2 does not perform its function correctly. Another explanation would be that i_2 performs it function correctly, while the other two inverters are broken. However, based on the diagram in Fig. 1 we can rule out the possibility that i_1 is broken while the other two inverters operate correctly. In total, there are five combinations of one or more inverters failing that explain the observed behavior:

i_1	i_2	i_3
healthy	broken	healthy
broken	healthy	broken
broken	broken	healthy
healthy	broken	broken
broken	broken	broken

Based on the probability that a single inverter fails, the explanation with a single broken inverter is most likely, and the explanation where all inverters are broken is least likely. Note that for this example, we make no assumption about how an inverter behaves if it is broken.

The above reasoning captures the essence of *model-based diagnosis* (MBD, [9]): with every component we associate a variable that captures its health state (healthy or broken), and we search for an assignment of values to these variables that logically explains the observed behavior. Note that even for diagnosis the model need not be made explicit. Model-based diagnosis is close to human reasoning, and the model may simply exist in the mind of a service engineer. However, if the model is made explicit, it becomes amenable to automated reasoning. For the above example, the following model can be used, where $h_1, \ldots, h_3$ are the (propositional) health variables associated with the inverters $i_1, \ldots, i_3$. The symbol $\Rightarrow$ denotes logical implication.

$$h_1 \Rightarrow z = \neg x$$
$$h_2 \Rightarrow y_1 = \neg z$$
$$h_3 \Rightarrow y_2 = \neg z$$

Together with a set of observations, this model can directly serve as input to a truth maintenance system to automatically derive a sequence of assignments to the health variables that explains the observations. This also covers the correct behavior, for which the most likely explanation simply entails that all components behave correctly. Note that the above model is fully compositional: it can (automatically) be composed from three instances of a model for a single inverter, and three equalities to make the connections of Fig. 1. This makes model-based diagnosis well suited for digital circuits and other devices whose components have clearly specified functionality.

2.2 Model Strength

Models that are used for diagnostic reasoning have different capabilities with respect to the kind of faults that they are able to identify. This is often intuitively referred to as a model's *strength*. In this sense, the above model for the three-inverter example is rather weak: it only describes the correct behavior of the system. A much stronger model is obtained if we also specify the different ways in which the components can fail. For example, for an inverter we may explicitly want to specify the behavior in the following cases.

- healthy behavior
- output stuck at one
- output stuck at zero
- bypass (input and output equal)

Because there are usually multiple ways in which a component can fail, a single binary health variable per component is not sufficient for most strong models. In addition, to facilitate the calculation of the probability of the different explanations, the failure modes of components can be assigned their own probabilities.

2.3 A Weak Model

While our example model is weak in the sense that it did not include the faulty behavior of the three component inverters, it still fully describes the behavior of the system in case all inverters are healthy. Removing also that information, while retaining the expected behavior $x = y_1 = y_2$ yields an even weaker model that tells us only that i_1 and i_2 determine output signal y_1, and that i_1 and i_3 determine output signal y_2. This weaker model can still be used for diagnosis as follows.

Consider the failure $x = 1$, $y_1 = 0$, and $y_2 = 1$. In this failure, two output signals y_1 and y_2 are derived from the same input signal x. Of these two output signals, y_2 is according to specification, but y_1 is wrong. Without knowing their intended functionality, component i_1 is involved in both output signals, i_2 is only involved in the incorrect output signal y_1, and i_3 is only involved in the correct output signal y_2. Looking only at these involvements, i_2 is the most likely cause of the failure, and i_3 is least likely to be involved.

This example illustrates the essence of *spectrum-based fault localization* (SFL): identify the components of a system that determine its various outputs. The components whose activities coincide with the occurrence of failures are also most likely causing these failures. The name of this diagnosis technique refers to its application to software, where the activity of components, or parts of a system is recorded in so-called *program spectra*.

Whereas MBD always yields diagnoses that are valid explanations for the observed behavior within the context of the model that is being used, SFL is inherently inaccurate in the sense that it may identify the wrong component as the most likely cause of the observed failures. The reason is that the activity of a component may coincide with the occurrence of a failure without causing it. This can easily lead to wrong diagnoses in the case of false negatives (no failure, or no error detected while the faulty component was active). A comparable problem with model-based diagnosis is that typically many different explanations for the observed behavior exist, and that the most likely explanation need not be the actual cause.

Despite these problems, MBD and SFL are both viable techniques. Whereas MBD has successfully been used in the diagnosis of circuits and complex mechanical systems, spectrum-based fault localization applies naturally to software: specifications of software typically cover the expected behavior without revealing the internal composition. Running faulty software on a test suite usually identifies a number of test cases for which the system behaves according to specification (passed runs), as well as a number of test cases for which this is not the case (failed runs). The software itself, as an executable model, will tell us which parts of the system were involved in the execution of these test cases. This information can be obtained via standard profiling techniques, and constitutes a so-called *program spectrum* per test case. In the remainder of this paper we will be exploring this technique further.

3 Spectrum-Based Fault Localization

3.1 Failure, Error, and Fault

In Sec. 2 we introduced the concepts of failure, error, and fault. Here we apply this terminology to simple computer programs that transform an input file to an output file in a single run. Specifically in this setting, faults are *bugs* in the program code, and failures occur when the output for a given input deviates from the specified output for that input.

As an illustration, consider the C function in Fig. 2. 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 of 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 `den[j]` $\neq$ `den[j+1]`. Such errors can be temporary, i.e., not automatically leading to errors: 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.

```
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 */
            temp = num[j];
            num[j] = num[j+1];
            num[j+1] = temp;
         }
      }
   }
}
```

Fig. 2. A faulty C function for sorting rational numbers

Error detection is a prerequisite for spectrum-based fault localization: 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 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.

3.2 Program Spectra

A program spectrum [18] 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 [11] for an overview. In this paper we mainly work with so-called block 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. 2.

In addition to block hit spectra, specifically in Sec. 4 we will be working with hit spectra at the level of individual lines of code.

3.3 Fault Localization

The hit spectra of M runs constitute a binary matrix, whose columns correspond to N different parts (e.g., blocks of code) of a program (see Fig. 3). 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. Fault localization essentially consists in identifying the part whose column vector resembles the error vector most.

$$\begin{array}{c} \\ M \text{ spectra} \\ \\ \end{array} \begin{array}{c} \overset{N \text{ parts}}{\begin{bmatrix} x_{11} & x_{12} & \dots & x_{1N} \\ x_{21} & x_{22} & \dots & x_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M1} & x_{M2} & \dots & x_{MN} \end{bmatrix}} \\ \begin{array}{cccc} s_1 & s_2 & \dots & s_N \end{array} \end{array} \quad \overset{\text{errors}}{\begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{bmatrix}}$$

Fig. 3. The ingredients of spectrum-based fault localization

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., [13]). As an example, the Jaccard similarity coefficient (see also [13]) expresses the similarity s_j of column j and the error vector as the number of positions in which these vectors share an entry 1 (i.e., block was exercised and the run has failed), divided by this same number plus the number of positions in which the vectors have different entries:

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

where $a_{pq}(j) = |\{i \mid x_{ij} = p \wedge e_i = q\}|$, and $p, q \in \{0, 1\}$.

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, the block hit spectra for six input sequences $I_1, .., I_6$ to the `RationalSort` function of Fig. 2 are shown in Table 1 ('1' denotes

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

a hit). Block 5 corresponds to the body of the `RationalGT` function, which has not been shown in Fig. 2.

	block					
input	1	2	3	4	5	error
$I_1 = \langle\ \rangle$	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	$\frac{1}{6}$	$\frac{1}{5}$	$\frac{1}{4}$	$\frac{1}{3}$	$\frac{1}{4}$	

Table 1. Spectra and Jaccard coefficients for six inputs to `RationalSort`

Input sequences 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 Sec. 3.1: 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 Jaccard coefficients $s_1, \ldots, s_5$ shown at the bottom of the table (correctly) identify block 4 as the most likely location of the fault.

4 Getting Started

In this section we will show how to get started with applying spectrum-based fault localization to C programs. We will discuss the necessary tools, and will walk through an example involving the `RationalSort` function of Fig. 2. Finally, we discuss a set of benchmark faults that is widely used to quantify the effect of spectrum-based fault localization and other automated diagnosis / debugging tools.

4.1 Tools

Program spectra can be recorded using the GNU code coverage analysis tool `gcov`. This tool is a companion to the `gcc` compiler, which has options for instrumenting an executable to generate code coverage information at line granularity. Appendix A.1 contains a simple C program that converts the output of `gcov` to a row of binary numbers, indicating the activity of the various lines of code of a C program. The `gcov` tool generates information per source file, so here we assume single-file programs. This is applicable for the example experiment below, but also for the programs in the benchmark set discussed in Sec. 4.3.

After obtaining a set of program spectra for passed and failed runs using `gcc`, `gcov`, and `gcov2spectrum`, the program of Appendix A.2 can be used to perform the actual diagnosis. It reads the output of `gcov2spectrum` as a binary matrix, containing a program spectrum on each line. The last column of this matrix is assumed to contain the passed / failed information. The output is a list of column indices, sorted according to their Jaccard similarity to the last column, which is printed in parentheses, excluding the last column, and columns with zero similarity.

Assuming the programs of Appendices A.1 and A.2 are available as `gcov2spectrum.c` and `diagnosis.c` respectively, they can be compiled using the following commands:

```
$ gcc -o gcov2spectrum gcov2spectrum.c
$ gcc -o diagnosis diagnosis.c
```

`gcov2spectrum` is a very simple program: the macro `SPECTRUM_SIZE` must be adapted to the program size, and it can probably be made to fail for some constructions (see the comment in the program). For any serious applications, please use a proper parser.

4.2 An Example Experiment

To demonstrate spectrum-based fault localization, we will now apply all necessary steps on an example program. The program in Appendix A.3 expands the `RationalSort` function of Fig. 2 to a full program that reads the numerators and denominators from the command line. In order to use `gcov`, we must first compile it with two special flags. Assuming the program is in a file `rsort.c`, it can be compiled as follows.

```
$ gcc -fprofile-arcs -ftest-coverage rsort.c -o rsort
```

The following commands now run rsort on the inputs of Table 1.

```
$ ./rsort
$ ./rsort 1 4
$ ./rsort 2 1 1 1
$ ./rsort 4 1 2 2 0 1
$ ./rsort 3 1 2 2 4 3 1 4
$ ./rsort 1 4 1 3 1 2 1 1
```

After each run of `rsort`, the following commands must be applied to generate the spectra, and append them to the `spectra.txt` file, which serves as the input to `diagnosis`.

```
$ gcov rsort.c
$ rm rsort.gcda
$ ./gcov2spectrum < rsort.c.gcov >> spectra.txt
```

This is best done in a script, that first ensures `spectra.txt` is empty (e.g., by using `>` instead of `>>` in the first `gcov2spectrum` command). Running the resulting script results in a text file with six lines, one for each test case.

From the output of the `rsort` program it can be seen that the fifth run fails: the output is not sorted. As in Sec. 3.3, the other runs complete without a failure. This information must be added to the `spectra.txt` for the `diagnosis` program: to the fifth line we append an entry 1 to indicate that the run has failed. To all other lines we append an entry 0 to indicate that the run has passed. Since `gcov2spectrum` generates spectra of 100 flags, this results in a 6×101 matrix, whose last column is the error vector.

After thus having included the error vector, the `diagnosis` program is applied to this data as follows.

```
$ diagnosis 6 101 spectra.txt
```

The output of `diagnosis` ranks the lines of code of `rsort.c` according to decreasing Jaccard similarity with the error vector:

```
22 (0.33) 23 (0.33) 24 (0.33) 11 (0.25) 6 (0.25) 20 (0.25) 7 (0.25)
54 (0.20) 45 (0.20) 19 (0.20) 53 (0.17) 56 (0.17) 52 (0.17) 51 (0.17)
33 (0.17) 32 (0.17) 16 (0.17) 35 (0.17) 18 (0.17) 39 (0.17) 44 (0.17)
15 (0.17) 57 (0.17)
```

This clearly identifies lines 22–24 as the most likely location of the fault. These are the three lines that correspond to block 4 of Fig. 2.

4.3 Benchmark Set

Experiments with spectrum-based fault localization are often performed on a benchmark set of software faults known as the *Siemens set* [12], which consists of seven small C programs. Every single program has a correct version and a set of faulty versions of the same program. Each faulty version contains exactly one fault. However, the fault may span through multiple statements and/or functions. Each program also has a set of inputs that ensures full code coverage. Table 2 provides more information about the programs in the package. See [12] for more information.

The availability of the correct version of every program provides an easy means of error detection: using simple scripts we can run all input files, and compare the output of the faulty version of a program with the output generated by the correct version.

5 Similarity Coefficient and Diagnostic Accuracy

At the end of Sec. 3.3 we described spectrum-based fault localization as 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 as an example of the kind of experiments that can be performed using the Siemens set, in this section we investigate the influence of the similarity coefficient on the quality, or accuracy of the diagnosis delivered by spectrum-based fault localization.

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

Table 2. Set of programs used in the experiments

5.1 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 there are two or more blocks ranking with the same coefficient, we use the average ranking position for all the blocks.

Let $d \in \{1, \ldots, N\}$ be the index of the block that we know to contain the fault. For all $j \in \{1, \ldots, N\}$, let s_j denote the similarity coefficient calculated for block j. Then the ranking position is given by

$$\tau = \frac{|\{j|s_j > s_d\}| + |\{j|s_j \geq s_d\}| - 1}{2} \tag{2}$$

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

$$q_d = (1 - \frac{\tau}{N-1}) \cdot 100\% \tag{3}$$

5.2 Experiment

To investigate the influence of the similarity coefficient on the diagnostic accuracy we evaluate q_d on the same set of faults using several different similarity coefficients. For the benchmark set we selected 120 faults from the Siemens set that do not span multiple locations. We evaluated the Jaccard coefficient of Eq. (1), which is used by the Pinpoint tool ([6], see Sec. 7), the coefficient used in the Tarantula fault localization tool ([14], see also Sec. 7), and the Ochiai coefficient. We experimentally identified the latter as giving the best results among all eight coefficients used in a data clustering study in molecular biology [7], which also included the Jaccard coefficient. Both the Tarantula coefficient and the Ochiai coefficient can be expressed using the the notation introduced in Sec. 3.3:

– Tarantula:

$$s_j = \frac{\frac{a_{11}(j)}{a_{11}(j)+a_{01}(j)}}{\frac{a_{11}(j)}{a_{11}(j)+a_{01}(j)} + \frac{a_{10}(j)}{a_{10}(j)+a_{00}(j)}} \tag{4}$$

– Ochiai:

$$s_j = \frac{a_{11}(j)}{\sqrt{(a_{11}(j)+a_{01}(j)) * (a_{11}(j)+a_{10}(j))}} \tag{5}$$

In addition to Eq. (4), the Tarantula tool uses a second coefficient, which amounts to the maximum of the two fractions in the denominator of Eq. (4). This second coefficient is interpreted as a *brightness* value for visualization purposes, but the experiments in [14] indicate that the above coefficient can be studied in isolation. For this reason, we have not taken the brightness coefficient into account.

Figure 4 shows the results of this experiment. It plots q_d, as defined by Eq. (3), for the three similarity coefficients mentioned above, averaged per program of the Siemens set. Instead of per line of code, as suggested in Sec. 4, for these experiments we obtained the hit spectra per block of code. However, apart from statements that alter the control flow within a block (`goto`, `break`, `continue`, `return`), this leads to the same results. See [2] for more details on these experiments.

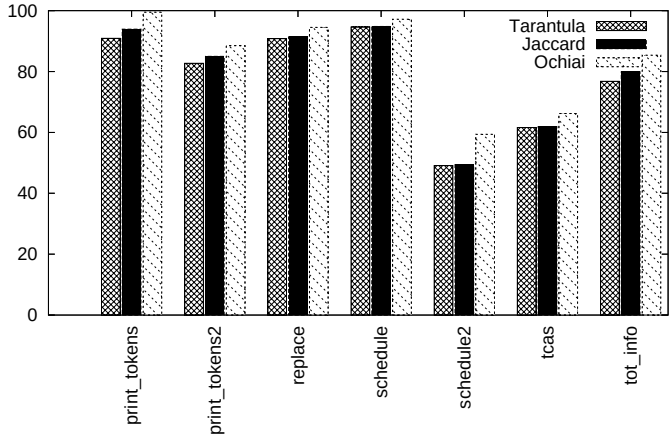

Fig. 4. Diagnostic accuracy q_d

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 5% over the second-best case, and improvements of up to 30% for individual faults. Factors that likely contribute to this effect are the following. First, for $a_{11}(j) > 0$ (the only relevant case: $a_{11}(j) = 0$ implies

$s_j = 0$) the Tarantula coefficient can be written as $1/(1 + c\frac{a_{10}(j)}{a_{11}(j)})$, with c the constant $\frac{a_{11}(j)+a_{01}(j)}{a_{00}(j)+a_{10}(j)}$. This depends only on presence of a block in passed and failed runs, while the Ochiai coefficient also takes the absence in failed runs into account. Second, compared to Jaccard (Eq. 1), for the purpose of determining the ranking the denominator of the Ochiai coefficient contains an extra term $\frac{a_{01}(j)\cdot a_{10}(j)}{a_{11}(j)}$, which amplifies the differences in the column vectors of Fig. 3. This can be seen by squaring Eq. 5, and dividing the numerator and denominator by $a_{11}(j)$, which does not change the ranking.

Other parameters that may influence the diagnostic accuracy of spectrum-based fault localization are the quality of the error detection information, and the number of passed and failed runs taken into account. These parameters, and their influence on the superior performance of the Ochiai coefficient are investigated in [1].

6 Case Study: Embedded Software

As a case study, to give an indication of the extent to which spectrum-based fault localization can practically be applied to commercial software products, we performed an experiment involving embedded software in the consumer electronics area. The actual experiment is described in Sec. 6.2 and 6.3, and discussed in Sec. 6.4. First, in Sec. 6.1, we discuss the relevance of SFL for this specific application domain.

6.1 Relevance to Embedded Software

Especially because of constraints imposed by the market, the conditions under which embedded software in consumer electronics products is developed, are somewhat different from those for other software products. Typical characteristics of the computing environment found in these products are non-commodity hardware with limited CPU and memory resources, and the absence of standard tools for getting insight in the dynamic behavior. In addition, the systems are highly concurrent, and the software operates at a low level of abstraction from the underlying hardware. Therefore, the design and implementation of these systems are complicated by factors that can largely be abstracted away from in other software systems.

Furthermore, on top of challenges that the entire software industry has to deal with, such as geographically distributed development organizations, the strong competition between manufacturers of consumer electronics makes it absolutely vital that release deadlines are met. Finally, although important safety mechanisms, such as short-circuit detection, are sometimes implemented in software, for a large part of the functionality there are no personal risks involved in transient failures.

Under these circumstances, it is not uncommon that consumer electronics products are shipped with several known software faults outstanding. To a certain extent, this also holds for other software products, but the combination

of the complexity of the systems, the tight constraints imposed by the market, and the relatively low impact of the majority of possible system failures creates a unique situation. Instead of aiming for correctness, the goal is to create a product that is of value to customers, despite its imperfections, and to bring the reliability to a commercially acceptable level (also compared to the competition) before a product must be released.

Spectrum-based fault localization can help to reach this goal faster, and may thus reduce the time-to-market, and lead to more reliable products. Specific benefits are the following.

- As a black-box diagnosis technique, it can be applied without any additional modeling effort.
- It improves insight in the run-time behavior. Because of the concurrency, but also because of the decentralized development this is often lacking.
- It can easily be integrated with existing testing procedures, such as overnight playback of recorded usage scenarios. In addition to the information that errors have occurred in some scenarios, this gives a first indication of the parts of the software that are likely to be involved in these errors. In the large, geographically distributed development organizations that are involved, this may help to identify which teams of developers to contact.
- SFL is a light-weight technique, and does not require an extensive tooling infrastructure. This is relevant because of the limited computing resources and the non-commodity hardware.

While none of these benefits are unique, their combination makes program spectrum analysis an attractive technique for diagnosing embedded software in consumer electronics.

6.2 Experiments

The subject of our experiments is the control software in a particular product line of analog television sets from a well known international manufacturer of consumer electronics products. All audio and video processing in these sets is implemented in hardware, but the software is responsible for tasks such as decoding remote control input, displaying the on-screen menu, and coordinating the hardware (e.g., optimizing parameters for audio and video processing based on an analysis of the signals). Most teletext[2] functionality is also implemented in software.

The software itself consists of approximately 450K lines of C code, which is configured from a much larger (several MLOC) code base of software components.

The control processor is a MIPS running a small multi-tasking operating system. Essentially, the run-time environment consists of several threads with increasing priorities, and for synchronization purposes, the work on these threads

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

is organized in 315 logical threads inside the various components. Threads are preempted when work arrives for a higher-priority thread.

The total available RAM memory in consumer sets is two megabyte, but in the special developer version that we used for our experiments, another two megabyte was available. In addition, the developer sets have a serial connection, and a debugger interface for manual debugging on a PC.

We diagnosed two faults in the control software, one existing, and one that was seeded to reproduce an error from a different product line:

Load Problem A known problem with the specific version of the control software that we had access to, is that after teletext viewing, the CPU load when watching television (TV mode) is approximately 10% higher than before teletext viewing. This is illustrated in Fig. 5, which shows the CPU load for the following scenario: one minute TV mode, 30 s teletext viewing, and one minute of TV mode. The CPU load clearly increases around the 60th sample, when the teletext viewing starts, but never returns to its initial level after sample 90, when we switch back to TV mode.

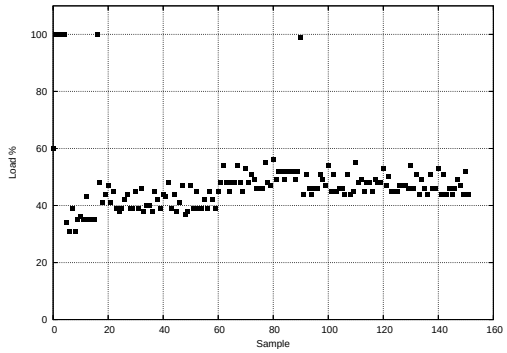

Fig. 5. CPU load measured per second

Teletext Lock-up Problem. Another product line of television sets provides a function for searching in teletext pages. An existing fault in this functionality entails that searching in a page without visible content locks up the teletext system. A likely cause for the lock-up is an inconsistency in the values of two state variables in different components, for which only specific combinations are allowed. We hard-coded a remote control key-sequence that injects this error on our test platform.

For diagnosing these two faults, we wrote a small software component for recording and storing program spectra, and for transmitting them off the television set via the serial connection. The transmission is done on a low-priority thread while the CPU is otherwise idle, in order to minimize the impact on the

timing behavior. Pending their transmission via the serial connection, our component caches program spectra in the extra memory available in our developer version of the hardware.

For diagnosing the load problem we obtained hit spectra for the logical threads mentioned above, resulting in spectra of 315 binary flags. We approached the lock-up problem at a much finer granularity, and obtained block hit spectra for practically all blocks of code in the control software, resulting in spectra of over 60,000 flags.

The hit spectra for the logical threads are obtained by manually instrumenting a centralized scheduling mechanism. For the block hit spectra we automatically instrumented the entire source code using the Front [4] parser generator.

In Sec. 3.3 we use program spectra for different runs of the software, but for embedded software in consumer electronics, and indeed for most interactive systems, the concept of a run is not very useful. Therefore we record the spectra per *transaction*, instead of per run, and we use two different notions of a transaction for the two different faults that we diagnosed:

- for the load problem, we use a periodic notion of a transaction, and record the spectra per second.
- for the lock-up problem, we define a transaction as the computation in between two key-presses on the remote control.

6.3 Diagnosis

For the load problem we used the scenario of Fig. 5. We marked the last 60 spectra, for the second period of TV mode as 'failed,' and those of earlier transactions as 'passed.' In the ranking that follows from the analysis of Sec. 3.3, the logical thread that had been identified by the developers as the actual cause of the load problem was in the second position out of 315. In the first position was a logical thread related to teletext, whose activation is part of the problem, so in this case we can conclude that although the diagnosis is not perfect, the implied suggestion for investigating the problem is quite useful.

For the lock-up problem, we used a proper error detection mechanism. On each key-press, when caching the current spectrum, a separate routine verifies the values of the two state variables, and marks the current spectrum as failed if they assume an invalid combination. Although this is a special-purpose mechanism, including and regularly checking high-level assert-like statements about correct behavior is a valid means to increase the error-awareness of systems.

Using a very simple scenario of 23 key-presses that essentially (1) verifies that the TV and teletext subsystems function correctly, (2) triggers the error injection, and (3) checks that the teletext subsystem is no longer responding, we immediately got a good diagnosis of the detected error: the first two positions in the total ranking of over 60,000 blocks pointed directly to our error injection code. Adding another three key-presses to exonerate an uncovered branch in this code made the diagnosis perfect: the exact statement that introduced the state inconsistency was located out of approximately 450K lines of source code.

6.4 Discussion

Especially the results for the lock-up problem have convinced us that program spectra, and their application to fault diagnosis are a viable technique and useful tool in the area of embedded software in consumer electronics. However, there are a number of issues with our implementation.

First, we cannot claim that we have not altered the timing behavior of the system. Because of its rigorous design, the TV is still functioning properly, but everything runs much slower with the block-level instrumentation (e.g., changing channels now takes seconds). One reason is that currently, we collect block *count* spectra at byte resolution, and convert to block *hit* spectra off-line. Updating the counters in a multi-threaded environment requires a critical section for every executed block, which is hugely expensive. Fortunately, this information is not used, and a binary flag update can be implemented without a critical section.

Second, we cache the spectra of passed transactions, and transmit them off the system during CPU idle time. Because of the low throughput of the serial connection, this may become a bottleneck for large spectra and larger scenarios. In our case we could store 25 spectra of 65,536 counters, which was already slowing down the scenarios with more than that number of transactions, but even with a more memory-efficient implementation, this inevitably becomes a problem with, for example, overnight testing.

For many purposes, however, we will not have to store the actual spectra. In particular for fault diagnosis, ultimately we are only interested in the calculated similarity coefficients, and all similarity coefficients that we are aware of are expressed in terms of the four counters a_{00}, a_{01}, a_{10}, and a_{11} introduced in Sec. 3.3. If an error detection mechanism is available, like in our experiments with the lock-up problem, then these four counters can be calculated on the fly, and the memory requirements become linear in the number columns in the matrix of Fig. 3.

7 Related Work

Pinpoint [6] is a framework for root cause analysis on the J2EE platform. It is developed in the context of the Recovery Oriented Computing project [16], and is targeted at large, dynamic Internet services, such as web-mail services and search engines. It combines spectrum-based fault localization with a specific form of error detection, based on information coming from the J2EE framework, such as caught exceptions, and errors visible to users, such as HTTP errors. This makes the approach self-contained in the sense that no external characterization of traces is needed.

The Tarantula system [14, 15] has been developed for the C language, and applies spectrum-based fault localization to statement hit spectra. The resulting analysis is therefore quite close to that of Sec. 4. Tarantula comes with a graphical user interface, that interprets the calculated value for the similarity coefficient as a color index, used to visualize the suspiciousness of program statements. Tarantula relies on external error detection for the classification of runs

as passed or failed: whereas Pinpoint uses information from the J2EE framework for this classification, this information is input data for Tarantula. In other words, Tarantula implements only the diagnosis, and has to be complemented by adding a method of error detection.

AMPLE (Analyzing Method Patterns to Locate Errors) [8] is a system for identifying faulty classes in object-oriented software. It collects hit spectra of *method call sequences*, which are subsequences of a given length that occur in a full trace of incoming or outgoing method calls, received or issued by individual objects of a class. Each call sequence is assigned a weight, which captures the extent to which its occurrence or absence correlates with the detection of an error, i.e., it is a combined measure of similarity and dissimilarity. These weights are averaged over all call sequences of a class, leading to a class weight. Classes with a high weight are most likely to contain the fault that causes the detected error. Although the calculation of the sequence weights in AMPLE can be explained as an application of the technique of Sec. 3.3, the diagnosis is at class level, and the calculated coefficients are used only 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. As we already mentioned in Sec. 6.1, spectrum-based fault localization can be seen as a black-box technique. An example of a white box technique is model-based diagnosis (see, e.g., [9]), which we already encountered in Sec. 2. Here a diagnosis is obtained by logical inference from a formal model of the system, combined with a set of run-time observations. Model-based approaches to software diagnosis exist (see, e.g., [19]), but software modeling is extremely complex, so most software diagnosis techniques are black box.

Examples of other black box techniques are Nearest Neighbor [17], dynamic program slicing [3], and Delta Debugging [20]. 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 searching space to the set of statements which have influence on the value of a faulty point (e.g., output variable). 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 [10] Delta Debugging is combined with dynamic slicing in 4 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.

8 Conclusion

In this paper we have introduced spectrum-based fault localization, an automated diagnosis technique that is well suited for diagnosing software faults. We positioned the technique in the context of the general, model-based, diagnosis problem, and related it to other approaches to software fault diagnosis.

In addition to a sample of current research on spectrum-based fault localization, we discussed a case study involving industrial (embedded) software to show the practical relevance and applicability of the technique. Finally, we provided instructions for quickly getting started with applying spectrum-based fault localization on C software, using standard tools for code coverage analysis.

References

1. R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of TAIC PART 2007*. To appear.
2. R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, pages 39 – 46. IEEE Computer Society, 2006.
3. H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software - Practice and Experience*, 23(6):589–616, 1993.
4. L. Augusteijn. Front: a front-end generator for Lex, Yacc and C, release 1.0. `http://front.sourceforge.net/`, 2002.
5. A. Avižienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.
6. M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society.
7. A. da Silva Meyer, A. A. Franco Farcia, and A. Pereira de Souza. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (*Zea mays* L). *Genetics and Molecular Biology*, 27(1):83–91, 2004.
8. V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for Java. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, volume 3568 of *LNCS*, pages 528–550, Glasgow, UK, 2005. Springer-Verlag.
9. J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artif. Intell.*, 32(1):97–130, 1987.
10. N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 263–272, Long Beach, CA, USA, 2005. ACM Press.
11. M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. In *Proceedings of the SIGPLAN/SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, Montreal, Canada, June 16, 1998*, pages 83–90, 1998.

12. M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering*, pages 191–200, Sorrento, Italy, 1994. IEEE Computer Society.
13. A. K. Jain and R. C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.
14. J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, New York, NY, USA, 2005. ACM Press.
15. J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering, Orlando, Florida, USA, May 2002*, pages 467–477. ACM Press.
16. D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery Oriented Computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB/CSD-02-1175, U.C. Berkeley, March 2002.
17. M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, Montreal, Canada, October 2003. IEEE Computer Society.
18. T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the Sixth European Software Engineering Conference*, volume 1301 of *LNCS*, pages 432–449. Springer–Verlag, 1997.
19. F. Wotawa, M. Strumptner, and W. Mayer. Model-based debugging or how to diagnose programs automatically. In T. Hendtlass and M. Ali, editors, *IAE/AIE 2002*, volume 2358 of *LNCS*, pages 746–757. Springer-Verlag, 2002.
20. A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th International Symposium on the Foundations of Software Engineering, Charleston, South Carolina, November 2002*. ACM Press.

A Appendix

A.1 The gcov2spectrum Program

```
#include <stdio.h>
#include <assert.h>

#define SPECTRUM_SIZE 100 /* enough for rsort.c */

int spectrum[SPECTRUM_SIZE];

int main()
{
   int res,i;

   for ( i=0; i<SPECTRUM_SIZE; i++ ) {
      spectrum[i] = 0;
   }
   do {
      int counter, index;

      /* WARNING: unexecuted lines with conditional expressions or
       * matching comments may lead to invalid spectra!
       */
      res = scanf( "%d:%d", &counter, &index );
      assert( res == EOF || res == 0 || res == 2 );
      if ( res == 2 ) {
         if ( index >= SPECTRUM_SIZE ) {
            fprintf( stderr, "SPECTRUM_SIZE too small\n" );
            exit( 1 );
         }
         else if ( counter != 0 ) {
            assert( spectrum[index] == 0 );
            spectrum[index] = 1;
         }
      }
      if ( res != EOF ) {
         /* eat the rest of the line.
          */
         while ( getchar() != '\n' );
      }
   } while ( res != EOF );
   for ( i=0; i<SPECTRUM_SIZE; i++ ) {
      printf( " %4d", spectrum[i] );
   }
   printf( "\n" );
   return 0;
}
```

A.2 The diagnosis Program

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int **matrix;
int nrow, ncol;

double jaccard( int j )
{
   /* calculate the jaccard similarity of columns j and ncol-1 */
   int i, a01=0, a10=0, a11=0;

   for ( i=0; i<nrow; i++ ) { /* these are mutually exclusive: */
      if ( matrix[i][j] == 0 && matrix[i][ncol-1] != 0 ) a01++;
      if ( matrix[i][j] != 0 && matrix[i][ncol-1] == 0 ) a10++;
      if ( matrix[i][j] != 0 && matrix[i][ncol-1] != 0 ) a11++;
   }
   return ((double) a11) / ((double) (a01+a10+a11));
}

int cmp_columns( const void *ip, const void *jp )
{
   int i = *((int*) ip);
   int j = *((int*) jp);

   if ( jaccard( i ) == jaccard( j ) ) {
      return 0;
   }
   else if ( jaccard( i ) > jaccard( j ) ) {
      return -1;
   }
   else {
      return 1;
   }
}

int main(int argc, char *argv[] )
{
   FILE *matrix_file;
   int *indices;
   int i, j;

   if ( argc !=4 ||
        sscanf( argv[1], "%d", &nrow ) != 1 ||
        sscanf( argv[2], "%d", &ncol ) != 1) {
      printf( "Usage: diagnosis <nrows> <ncol> <file>\n" );
      exit( 0 );
   }
```

```
   if ( (matrix_file = fopen( argv[3], "r" )) == NULL ) {
      fprintf( stderr, "cannot open file %s\n", argv[3] );
      exit( 1 );
   }
   if ( (matrix = malloc( nrow * sizeof( int* ) )) == NULL ||
        (indices = malloc( ncol * sizeof( int ) )) == NULL ) {
      fprintf( stderr, "malloc error\n" );
      exit( 1 );
   }
   for ( i=0; i<nrow; i++ ) {
      if ( (matrix[i] = malloc( ncol * sizeof( int ) )) == NULL ) {
         fprintf( stderr, "malloc error\n" );
         exit( 1 );
      }
      for ( j=0; j<ncol; j++ ) {
         if ( fscanf( matrix_file, "%d", &( matrix[i][j] ) ) != 1 ) {
            fprintf( stderr, "input error\n" );
            exit( 1 );
         }
      }
   }
   if ( fscanf ( matrix_file, "%d", &i ) == 1 ) {
      fprintf( stderr, "warning: file %s contains more data\n",
               argv[3] );
   }
   fclose( matrix_file );
   for ( j=0; j<ncol; j++ ) {
      indices[j] = j;
   }
   /* sort the column indices on their calculated jaccard similarity to
    * the last column, and print them in this order.
    */
   qsort( (void*) indices, (size_t) ncol-1, sizeof( int ), cmp_columns );
   for ( j=0; j<ncol-1 && jaccard( indices[j] ) > 0.0 ; j++ ) {
      printf( "%d (%.2f) ", indices[j], jaccard( indices[j] ) );
   }
   return 0;
}
```

A.3 The `rsort` Program

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int RationalGT( int a, int b, int c, int d )
{
   if ( b*d == 0 ) {
      fprintf( stderr, "zero denominator\n" );
      exit( 1 );
```

```
   }
   return ( b*d>0 && a*d > b*c ) || ( b*d<0 && a*d < b*c );
}

void RationalSort(int n, int *num, int *den)
{
   int i,j,temp;

   for ( i=n-1; i>=0; i-- ) {
      for ( j=0; j<i; j++ ) {
         if (RationalGT(num[j], den[j],
                        num[j+1], den[j+1])) {
            temp = num[j];
            num[j] = num[j+1];
            num[j+1] = temp;
            /* fault: forgot to swap the denominators */
         }
      }
   }
}

int main( int argc, char *argv[] )
{
   int *num, *den, i;

   if ( (argc-1) %2 != 0 ) {
      fprintf( stderr, "odd input\n" );
      exit(1);
   }
   if ( (num = malloc( (argc/2) * sizeof( int ) )) == NULL ||
        (den = malloc( (argc/2) * sizeof( int ) )) == NULL ) {
      fprintf( stderr, "malloc error\n" );
      exit(1);
   }
   for ( i=0; i<argc/2; i++ ) {
      if ( sscanf( argv[2*i+1], "%d", num+i ) != 1 ||
           sscanf( argv[2*i+2], "%d", den+i ) != 1 ) {
         fprintf( stderr, "input error\n" );
         exit( 1 );
      }
   }
   RationalSort( (argc-1)/2, num, den );
   printf( "\nsorted:\n" );
   for ( i=0; i<(argc-1)/2; i++ ) {
      printf( "%d/%d ", num[i], den[i] );
   }
   printf( "\n" );
   return 0;
}
```