Zoltar: A Spectrum-based Fault Localization Tool

Tom Janssen Rui Abreu Arjan J.C. van Gemund

Embedded Software Lab
Delft University of Technology
The Netherlands
{t.p.m.janssen, r.f.abreu, a.j.c.vangemund}@tudelft.nl

ABSTRACT

Locating software components which are responsible for observed failures is the most expensive, error-prone phase in the software development life cycle. Automated diagnosis of software faults can improve the efficiency of the debugging process, and is therefore an important process for the development of dependable software. In this paper we present a toolset for automatic fault localization, dubbed Zoltar, which adopts a spectrum-based fault localization technique. The toolset provides the infrastructure to automatically instrument the source code of software programs to produce runtime data, which is subsequently analyzed to return a ranked list of likely faulty locations. Aimed at total automation (e.g., for runtime fault diagnosis), Zoltar has the capability of instrumenting the program under analysis with fault screeners, for automatic error detection. Using a small thread-based example program as well as a large realistic program, we show the applicability of the proposed toolset.

Categories and Subject Descriptors

D.2.5 [**Software engineering**]: testing and debugging—*debugging aids, diagnostics*.

General Terms

Reliability, Experimentation.

Keywords

Zoltar, spectrum-based fault localization, automatic error detection, runtime monitoring

1. INTRODUCTION

Debugging is an important, expensive phase of the software development cycle. Several debugging tools exist which are based on stepping through the execution of the program (e.g., [9, 4, 11]). These traditional, manual fault localization approaches have a number of important limitations. The placement of print statements as well as the inspection of their output are unstructured and ad-hoc, and are typically based on the developer's intuition. In addition, developers tend to use only test cases that reveal the failure, and therefore do not use valuable information from (the typically available) passing test cases.

Aimed at drastic cost reduction, much research has been performed in developing automatic fault localization techniques and tools. One of the predominant techniques are those based on a black box statistics-based method which takes a program and available test cases and returns the most probable location (component) that explains the observed failed test cases. In this paper a toolset is presented that implements a technique called spectrum-based fault localization (SFL [2, 6]). SFL is based on instrumenting a program to keep track of executed parts. This log, run-time data is then analyzed to yield a list of source code locations ordered by the likelihood of it containing the fault. Furthermore, the tool set enables a program to be trained with expected behavior and to automatically detect an error if unexpected behavior is observed. The fact that no knowledge is needed of the program to acquire possible fault locations makes this set of tools a useful extension to currently applied methods of testing and debugging. Moreover, this toolset could be very useful in the context of testing programs at runtime. As an example, our tool paves the way to triggering automatic recovery mechanisms.

The toolset, dubbed Zoltar, is the result of research done at the Delft University of Technology in the context of the TRADER project [10]. The tool is available for download from `http://www.fdir.org/zoltar`.

The remaining of this paper is organized as follows. In the next section we give background information on the spectrum-based fault localization technique which is adopted by the Zoltar toolset. In section 3 the toolset is described and an example program is used to demonstrate the tools. Section 4 shows usage of the toolset on a realistically sized program. Finally, section 5 concludes the paper.

2. SPECTRUM-BASED FAULT LOCALIZATION

An important part of diagnosis and repair consists in localizing faults, and several tools for automated debugging and systems diagnosis implement an approach to fault localization based on an analysis of the differences in program *spectra* 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 pro-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SINTER'09, August 25, 2009, Amsterdam, The Netherlands.
Copyright 2009 ACM 978-1-60558-681-6/09/08 ...$10.00.

gram spectrum is an execution profile that indicates which parts of a program are active during a run. Fault localization entails identifying the part of the program whose activity correlates most with the detection of errors.

Spectrum-based fault localization (SFL) 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.

2.1 Failures, Errors, and Faults

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. Since we focus on computer programs, 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.

```
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; } } } }
```

Figure 1: A faulty C function for sorting rational numbers.

To illustrate these concepts, consider the C function in Figure 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: 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, 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 SFL: 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. The Zoltar toolset supports instrumenting fault screeners, which are generic program invariants that are trained to be application specific. This will be discussed in more detail in Section 3.4.

2.2 Program Spectra

A program spectrum 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 [5], some of which are supported by the Zoltar toolset (such as spectra for basic block hits, function hits, def-use pairs). In this this example we 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. As an illustration, we have identified the blocks of code in Figure 1.

2.3 Fault Localization

The hit spectra of N runs constitute a binary $N \times M$ matrix A, whose columns correspond to M different parts (blocks in our case) of the program. The information in which runs an error was detected constitutes another column vector e, the error vector. This vector can be thought to represent a hypothetical part of the program that is responsible for all observed errors. The pair (A, e), which serves as input for SFL, is visualized in Figure 2. Fault localization essentially consists in identifying the part whose column vector resembles the error vector most.

$$
\begin{array}{c}
\\
N \text{ spectra}
\end{array}
\overset{M \text{ components}}{\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1M} \\ a_{21} & a_{22} & \dots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \dots & a_{NM} \end{bmatrix}}
\quad
\overset{\text{error detection}}{\begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_N \end{bmatrix}}
$$

Figure 2: Input to SFL

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. By default, Zoltar uses the Ochiai coefficient s_O, used in the molecular biology domain, since this coefficient performed best in experiments [2]:

$$s_O(j) = \frac{n_{11}(j)}{\sqrt{(n_{11}(j)+n_{01}(j)) * (n_{11}(j)+n_{10}(j))}}$$

where $n_{pq}(j)$ indicates the number of runs in which block j has been touched during the execution, denoted by $p \in \{0,1\}$, and where the runs are failed or passed, indicated by $q \in \{0,1\}$:

n	touched	run
n_{00}	no	passed
n_{10}	yes	passed
n_{01}	no	failed
n_{11}	yes	failed

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 $I_1, \ldots, I_6$ (see below). The block hit spectra for these runs are as follows ('1' denotes a hit), where block 5 corresponds to the body of the `RationalGT` function, which has not been shown in Figure 1.

	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_O	.40	.44	.50	.58	.50	

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 coefficient s_O (correctly) identifies block 4 as the most likely location of the fault. In general, however, the faulty component(s) may be outranked by other components, entailing non-zero search effort by the users. Research has shown that for small programs ($O(100)$ lines) $5 - 20\%$ of the code remains to be inspected by the user [12]. However, for large programs this fraction drops to less than a percent [3], making SFL an interesting debugging aid.

3. TOOL ARCHITECTURE

The Zoltar toolset provides a blackbox method involving three basic steps:

- instrumentation;
- data gathering;
- data analysis.

In the following, we discuss these three steps in more detail.

3.1 Instrumentation

A program has to be instrumented in order to create execution log data during the tests. When running the instrumented program, a counter corresponding to an instrumented piece of code is incremented each time that piece of code is executed. This results in a spectrum of the execution of the program under test.

The `instrument` tool of the Zoltar toolset is based on the LLVM framework[7]. Instrumentation of a program is implemented as different, customizable, passes of the LLVM optimizer. This enables multiple instrumentation passes to be applied on a single program. A custom instrumentation can be added with minimal effort. The instrumentation is performed on the LLVM intermediate byte code. Using LLVM tools the resulting instrumented byte code can be compiled to a native executable having the same functionality as the original plus functionality to generate spectrum information of runtime behavior.

Different types of program points can be instrumented for generating program spectra using the `instrument` tool. It supports, for example, *function* level instrumentation and instrumentation at the *basic block* level, but also supports custom program point instrumentation. Multiple types of program points can be instrumented within the same program, resulting in different program spectra for each type. This enables different types of analysis on the same set of executions.

Next to the generation of program spectra, programs can be instrumented with generic program invariants, which can be trained to become application-specific. Program invariants are predicates that retain the same value throughout the execution of the program. For example, each loop in the program can hold an invariant for the maximum number of iterations.

The `instrument` tool is able to instrument different types of invariants. It can keep track of values that are *stored* to memory, values that are *loaded* from memory, the number of *iterations* of a loop and the amount of *time* that the control flow remains in a function. Also, custom invariant instrumentations can be added.

To make sure that the instrumented program does not accidentally invalidate the data that is gathered during runtime *memory protection* can optionally be instrumented. This will instrument every store operation with a check if the part of memory which contains the instrumentation data is not overwritten.

Finally, the `instrument` tool will bypass the main function of the program to be able to initialize the instrumentation data and to handle exceptions while not losing recorded data.

3.2 Data Gathering

Running the instrumented program on test inputs results in the gathering of runtime data, which consists of the program spectra of different runs. A run can either be the complete execution of the program, or the result of a timed transaction. The latter option creates multiple runs of a predefined lenght of time, which can be useful for continuous programs. In a future version of the toolset, there will be support for transactions that will be triggered by the execution of instrumented program points. This data is extended with the pass/fail status of each run, which can be either set manually, or generated automatically using automatic error detection. The gathered data is essentially the pair (A, e) to be used as input for the SFL technique.

In most software projects tests are created along with the software itself. During the implementation and testing phases these tests are used to validate the functionality of the software. Failing tests indicate which features of the software are incomplete or faulty. Existing test suites can be

used with the instrumented program for the data gathering phase to generate program spectra.

Some types of bugs are difficult to locate. For example, multithreaded programs with thread related faults are hard to debug, since a piece of code can contain a fault which did cause an error in small tests but will lead to a failure when integrated into a larger code base. The Zoltar toolset can aid in locating various types of bugs using the same basic steps.

To illustrate the data gathering process, consider the example `textVal` program of which the pseudocode is given in Figure 3. It calculates a value based on the number of occurrences of different types of characters within a text. For demonstration purposes this program is deliberately created using three separate threads, each scanning for a different type of character and incrementing a shared value.

```
/* shared data */
int val;

/* function for updating val */
void updateVal(int d) {
  int tmp = val;
  tmp += d;
  val = tmp;
}

/* thread for reading letters */
void *readLetters(void *buffer) {
  // while not at end of buffer
    // if current character is letter
      // lock mutex
      updateVal(1);
      // unlock mutex
}

/* thread for reading digits */
void *readDigits(void *buffer) {
  // while not at end of buffer
    // if current character is digit
      updateVal(2);
}

/* thread for reading other characters */
void *readOther(void *buffer) {
  // while not at end of buffer
    // if current character is non alphanumeric
      // lock mutex
      updateVal(10);
}
```

Figure 3: Pseudo code for the textVal program.

Two bugs are introduced, which relate to mutual exclusion. The critical section of this code is at the `updateVal` function, which adds the value of its argument to the shared value containing the result of the program. The thread that is responsible for reading letters of the alphabet, running the `readLetters` function, correctly locks and unlocks the mutex. However, it is assumed that other threads honor the same rules, which is not always the case and which in practice results in hard to find bugs. In this case, the thread that scans for digits (`readDigits`) does not lock and unlock the mutex, which results in a critical section that is not exclusively executed by one thread at a time. If the first threads reads the shared value and the second thread reads the same value before the first thread has written a value back, then information is lost and the resulting value would be lower than expected.

The second introduced bug in this code is located in the third thread, which scans for special characters. This thread does lock the mutex, but does not release the exclusive right to the critical section, resulting in a situation in which the first thread waits forever for the mutex to become unlocked.

Data is gathered by running the instrumented program on available test inputs. For the example program a number of tests were created. These include testing only letters for input, only digits or special characters, or combinations of character types. These are standard tests to verify the functionality of the program. A problem in this case is that some small tests will work as expected, but will fail if they are scaled up. The mutex problem is the cause of this irregular behavior, since task switching between the threads will unlikely occur during the scanning of small input.

test file	input	expected output	textVal output
test1.in	`"abcdefghij"`	10	10
test2.in	`"0123456789"`	20	20
test3.in	`"A0B1C2D3E4"`	15	15
test4.in	large text only	1040	1040
test5.in	large text and digits	920	784
test6.in	`".=+-#"`	50	hangs

Table 1: Test suite for the textVal program with information on the number of letters, digits and other characters.

Table 1 shows the test suite that is available for this program. The input, or a description of the input, is given together with the expected output and the actual output of the program.

First, test 1 to 5 are run, since test 6 clearly shows different behavior than giving incorrect output. Most likely, one will see the correct output for the first four tests, but an output that differs from the expected output at test 5. In our case the output turned out to be 784 instead of 920. Note that, in practice, we have no real clue what the cause would be. We see that a text with mixed letters and digits works as expected, given that the size of the text is limited, but the same test fails if the text size is large. In conventional debugging one could be mislead by these results and would start looking at the text buffer or the reading of the input. In this case we use a black box method. A program is instrumented and run in the same way the original program would have been run.

A script running these tests can compare the output of the instrumented program with the expected output and save this information of passed and failed runs. The result of this data gathering process is a program spectrum and a vector of pass/fail information of each run. To convert the program spectrum, which consists of counters for each program point, to a hit spectrum, as discussed in Section 2.2, every entry is substituted with 1 if the count is greater than zero. For the example program, instrumented to generate a program spectrum on the basic block level, this results in the pair (A, e) of which a simplified[1] version is shown in Figure 4. Block 1 corresponds to the basic block within the `updateVal` function, Block 2 to 4 correspond to the basic blocks within the `if` statements of the three thread functions.

[1]other basic blocks are either never executed or always executed or are not represented in the pseudo code given in Figure 3.

test file	block 1	2	3	4	error
test1.in	1	1	0	0	0
test2.in	1	0	1	0	0
test3.in	1	1	1	0	0
test4.in	1	1	0	0	0
test5.in	1	1	1	0	1

Figure 4: Simplified (A, e) for the textVal program.

3.3 Data Analysis

The resulting run-time data of the test runs is stored in a separate file. This file is created after the first run and is incrementally updated at every next run. This is illustrated in Figure 5. The data file contains program spectrum information, among other things.

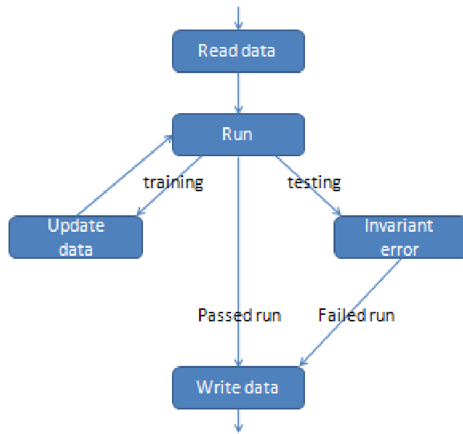

Figure 5: Zoltar data gathering workflow.

Two tools are provided for the data analysis. The `zoltar` tool is created to work in a console environment. This enables it to be used in many environments, including a remote shell. The `zoltar` tool provides an interface for the data file, can give a summary of the instrumentation of the program, is able to show spectrum data for each instrumented spectrum and can alter the pass/fail status of each run, among other things. By default the `zoltar` tool is started with a menu based interface.

By selecting an instrumented spectrum the SFL technique using the Ochiai coefficient (see Section 2.3) can be applied to return a ranked list of program locations ordered by the likelihood of containing the fault. These locations are mapped from points within the executable to lines in the source files.

Running the `zoltar` tool on the data acquired of the example program `textVal` results in the ranked list shown in Figure 6. It shows that line 43 in the textVal.c source file has the highest SFL score. In other words, since instrumentation is done on the basic block level, the execution of the basic block starting at line 43 correlates most with the failing run. This ranking gives a good starting point for debugging, since the most likely candidates for containing the bug are given.

A more intuitive visualization is achieved with the `xzoltar` tool, which shows a graphical representation of the same ranked list, where each line is color coded. A red line specifies a large SFL score and thus will be more likely to be the location of the fault. A green line will have little correlation with the failure. Next to the tab for the ranked list there is a separate tab for each instrumented source file. In these files the lines which are present in the ranking have the same coloring. This simplifies the process of locating areas of interest in each file and to view the surrounding code, which gives some context of why the specified location would possibly be at fault.

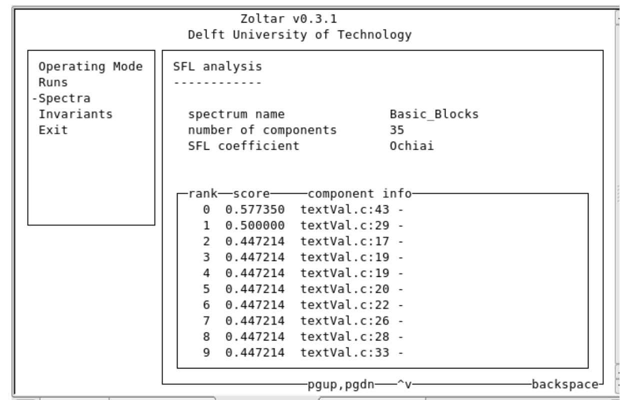

Figure 6: SFL results of the textVal tests.

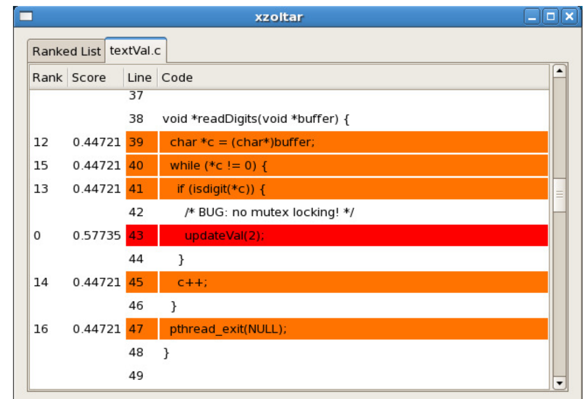

Figure 7: Visualization of SFL in the textVal source code.

In Figure 7 the `xzoltar` tool shows the highest ranking location in the code. It turns out that the call to `updateVal` without locking the mutex correlates most with the failing test runs. Even without having been part of the implementation process, a person debugging this program could use this tool, get these results, and notice that another thread function does lock the mutex when calling the same function.

It is important to realize that this tool will not explain why an error occurred. It statistically determines the most probable location a fault could reside. However, it gives a very useful starting point for inspecting the code. A debug strategy can be created using this ranking to reduce the wasted effort of inspecting code.

3.4 Automatic Error Detection

During the design phase of a program test oracles are usually available. However, this is not the case during the operational stage of a program. To facilitate debugging of a program in this stage we would like to have a test oracle as in the design state, however this would require automatic generation of invariants based on the program specifications, which is hard and currently not done in practice. Also, the detailed program specifications needed to achieve this are usually unavailable.

Secondly, some program faults cause a crash of the program, or cause the program to hang at a certain point. Next to that, determining correct execution is often more difficult than simply comparing outputs. Examples of these programs are programs with buggy user interfaces, continuous programs with sparse buffer overflow events, etc. Implementing application-specific invariants to detect these bugs is an expensive task and very error-prone and often incomplete, as it is done manually. Rather, we would like simple, generic program invariants, often dubbed fault screeners [1], which can be automatically trained to become application specific. In combination with the fault localization techniques, this automatic error detection could ultimately evolve into a fully automated fault localization, given that a program is trained for normal behavior using the test oracle during the design phase.

To monitor programs for errors the Zoltar tool set is able to instrument programs to automatically detect errors using fault screeners and stop the running program, while recording a failed run. In the `textVal` example (Section 3.2), the first five test inputs result in an output which can easily be verified. However, the sixth test input causes the program to hang. No output is returned, so it can not be verified, although the behavior is such that we know that this run fails expected execution. The erratic behavior of the sixth test input can be caught by instrumenting the program to generate an error if the program remains within a function for too long (i.e., a function timer invariant). A timer with a 1 ms interval signals a counter increment of each function that is *active*, i.e., which is still in the process of being executed. At the start of a function execution the associated counter is set to zero.

After having trained the generic function timer invariant using the first five test inputs (in general, a large test suite is required for training to represent *normal* program behavior), the program can be configured to generate an error if an instrumented invariant is violated during execution. The trained values are usually very strict. In the case of the example, the time spent in a function is related to the size of the input, which may vary. To be able to support a wider range of inputs, these trained values can be stretched using the `zoltar` tool. Only if an unexpected large amount of time is spent within a function an error will be generated.

When running the trained example program on the sixth test input, an error is generated while threads are blocked forever waiting for the mutex to become unlocked. The program is shut down while recording a failed run for this last test. Running the `xzoltar` tool at this point shows that the execution of the basic block within the if-statement of the third thread (reading the other character types) corresponds most with the failing run and is the candidate most likely to be inspected first. This is indeed the block of code in which unlocking the mutex is neglected.

4. CASE STUDY

To test the toolset on a more realistically sized program, we have instrumented `mplayer`[2] [8]. We investigated two cases of unexpected behavior using this version (1.0rc2) of `mplayer`:

1. When using a GUI and changing the position of the volume slider while the balance slider is not at center, the position of the balance slider changes as well.
2. Using a particular .avi video file as input, changing the position in the file beyond some point causes the video to end immediately.

Instrumenting a complete program is not always practical or can become inefficient. Instrumented code could slow down execution, depending on the program and the kind of instrumentation. For example, a program that writes to memory intensively could suffer slowdown when instrumented with store invariants and memory protection. To overcome this problem and to be able to investigate certain parts of the program (as a result of previous tests), the Zoltar toolset enables partial instrumentation of a program.

In the case of `mplayer` the core of the program together with the `gui` and `demux` libraries were instrumented on the basic block level together with memory protection and store invariants. This resulted in an instrumented movie player executable of which a slowdown of performance was barely noticeable, i.e., playing videos on the instrumented version showed little difference compared to the original version.

The code responsible for the first behavior was located by testing the instrumented executable with six different user inputs. Two test inputs caused the peculiar behavior and thus resulted in a failing run. The remaining four tests involved changing one or both of the sliders without the behavior appearing to the user, or not changing the sliders at all. Running the `xzoltar` tool afterwards resulted in one basic block at the top of the ranking. This part of the code handles the event of a changing value for the balance. In the following statements, also executed for a volume change event, the values for volume of the left and right channel are calculated from the values of the volume and balance. The next ranked basic block, located in another file, involves the reverse calculation, where the value for the balance is calculated from the volume values of the left and right channel. This value is then represented by the balance slider in the user interface. A fault in this part of the code results in the strange behavior of the balance slider. The top ranking locations are all related to changing the balance. The Zoltar toolset allows users without detailed information of the source code of `mplayer` to localize the cause of certain behavior. This can be very useful in the testing process of large software projects.

The second issue of `mplayer` is caused by a particular input. During the tests only one of the available test input videos resulted in the inability to jump to a location further in the file. By training the instrumented program with normal behavior (jumping to various locations in other input files that cause no abrupt ending) we were able to create program-specific invariants. During the operational phase we used the problematic file as input, which caused an invariant violation and resulted in a recorded failed run. In

[2] `mplayer` is a free and open source movie player able to run from command line but also supporting optional GUIs

total there were six passed runs during training and two failed runs in the operational stage. This resulted in a limited number of source files to investigate, since 7 files of the 132 instrumented source files, were represented in the basic blocks with a top 3 score in the ranking. Different basic blocks can get the same score in the SFL calculation, for example, when they are executed consecutively in all tests. This can be improved upon if more tests are run, collecting more data for the SFL technique. Extending the test procedure with seven more inputs, four of which resulting in a passed run and three resulting in a failed run, resulted in only 3 files that were represented by the basic blocks with a top 3 score. The basic blocks are distributed in three separate functions all related to seeking within a video file. One of these functions skips frames until a video keyframe is found. This is the part causing the perceived behavior, since it turns out that the input file partially lacks video keyframe information causing `mplayer` to keep skipping frames until the end of the file.

The described `mplayer` cases show that the Zoltar toolset is also suitable for analyzing behavior of large programs (`mplayer` totals around 650,000 LOC of which approximately 100,000 were instrumented). Furthermore, it shows that the toolset could be useful for localizing the type of bugs that show up as user interface inconsistencies as well as localizing pieces of code contributing to certain behavior when the bug is actually in the input data itself.

The examples also show that the lines of code most relevant to the user are not at the absolute top in the ranking. This is caused partially by the (limited) tests that are performed, but also by the nature of the fault localization technique. It will not return the exact location of the fault, but rather help in reducing the locations to investigate.

5. CONCLUSION

In this paper the Zoltar toolset was discussed, which adopts a technique to localize software faults based on statistical information retrieved from an instrumented version of the program under analysis. To demonstrate that the Zoltar toolset can aid developers in identifying software faults we have used two example. First, the example program used to present the tool contains mutex locking bugs, which is a type of bug that can result in a large debugging effort. To demonstrate the applicability of the toolset, a realistic program, `mplayer`, is also used as an example subject program. Two rather difficult to analyze, but very realistic, program behaviors are analyzed using the tools.

A great advantage of the Zoltar toolset is the variety of types of bugs that can be located using the underlying technique. As shown in this paper, thread related bugs, unexpected user interface behavior and input data faults, are among the types the Zoltar toolset is able to aid in locating.

Given the large applicability, the transparent way of using it, and the fact that it involves a blackbox method, the Zoltar toolset is of real added value to current testing practices. Supporting automatic error detection provides the ability to test programs at runtime without the need for manual invariant programming.

Finally, the Zoltar toolset can also be used to trigger automatic recovery mechanisms, paving the way for fully automatic, runtime system fault diagnosis and isolation.

6. ACKNOWLEDGMENTS

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

7. REFERENCES

[1] R. Abreu, A. González, P. Zoeteweij, and A. J. C. van Gemund. On the performance of fault screeners in software development and deployment. In *Proceedings of the 3rd International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'08)*, pages 123–130. INSTICC Press, May 2008.

[2] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART'07)*, pages 89–98, Windsor, United Kingdom, September 2007. IEEE Computer Society.

[3] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. A new Bayesian approach to multiple intermittent fault diagnosis. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'09)*, Pasadena, California, USA, 11 – 17 July 2009. AAAI Press.

[4] DBX. Debugging tools – DBX, SunOS 4.1.1 ed., March 1990. SUN MICROSYSTEMS, INC.

[5] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verifation and Reliability*, 10(3):171–194, 2000.

[6] 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 (ASE'05)*, pages 273–282, Long Beach, California, USA, 7 – 11 November 2005. IEEE Computer Society.

[7] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO'04),*, pages 75–88, San Jose, California, USA, 20 – 24 March 2004. IEEE Computer Society.

[8] MPlayer. Mplayer project website. http://www.mplayerhq.hu/.

[9] R. Stallman. Debugging with GDB – The GNU source level debugger, January 1994. Free Software Foundation.

[10] Trader. Trader project website, Embedded Systems Institute. http://www.esi.nl/trader/, 2005 – 2009.

[11] A. Zeller and D. Lütkehaus. DDD – A free graphical front-end for UNIX debuggers. *ACM SIGPLAN Notices*, 31(1):22–27, 1996.

[12] P. Zoeteweij, J. Pietersma, R. Abreu, A. Feldman, and A. J. van Gemund. Automated fault diagnosis in embedded software. In *Proceedings of the the ESI / Bits & Chips Embedded Systems Conference*, October 17 – 18 2007. Eindhoven, the Netherlands.