Zoltar: A toolset for automatic fault localization[*]

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 hosts a range of spectrum-based fault localization techniques featuring BARINEL, our latest algorithm. 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 diagnosis candidates. Aimed at total automation (e.g., for runtime fault diagnosis), Zoltar has the capability of instrumenting the program under analysis with fault screeners as a run-time replacement for design-time test oracles.

Categories and Subject Descriptors

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

General Terms

Reliability, Experimentation.

1. FAULT LOCALIZATION

An important part of diagnosis and repair consists in localizing faults. Several tools for automated debugging/diagnosis implement an approach to fault localization based on an analysis of the differences in abstraction of program traces (aka program *spectra* [4]) for *passed* and *failed* runs. Passed runs are executions of a program that behaved as expected, whereas failed runs are executions in which an error was observed.

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 [4], some of which are supported by the Zoltar toolset [5] (such as spectra for basic block hits, function hits, def-use pairs).

Error detection is a prerequisite for fault localization. We must know that something is wrong before trying to locate the responsible fault. Failures constitute a rudimentary form of error detection, but many errors remain latent and never lead to a failure. Test oracles can provide error detection at the development stage. An example of an operational stage technique that increases the number of errors that can be detected is program instrumentation with invariants such as checks on null pointers and array bounds checking. The Zoltar toolset supports instrumenting fault screeners [1], which are generic program invariants that are trained to be application specific.

The hit spectra of N runs constitute a binary $N \times M$ activity matrix A, whose columns correspond to M different parts of the program. The information in which runs an error was detected constitutes another column vector e, the error vector. The pair (A, e) serves as input for spectrum-based fault localization approaches.

Two approaches to fault localization that take (A, e) as their *only* input are supported by the Zoltar toolset. The first is based on statistics. Under the assumption that a high similarity to the error vector e 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. The best performing coefficient is the Ochiai coefficient [2] and is used by Zoltar by default. However, the tool set also provides several other coefficients [2], such as the Tarantula coefficient [6].

The second, distinguishing, approach implemented in the Zoltar toolset is a reasoning method, dubbed BARINEL [3]. BARINEL is a spectrum-based, logic reasoning approach to fault localization, and is able to deduce multiple-fault candidates. It returns a ranked list of diagnosis candidates based on a posterior probability. The ranked list of multiple-fault diagnosis candidates is subsequently translated to the ordered listing of program locations, similar to the statistical techniques. The BARINEL approach has been shown to

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

outperform the coefficient-based methods, even with single-fault cases [3].

Spectrum-based fault localization techniques, such as the ones provided by the Zoltar toolset, do not rely on a model of the system under investigation, and can easily be integrated with existing testing procedures. Due to the relatively small overhead with respect to CPU time and memory requirements, they lend itself well for application within resource-constrained environments. Research has shown that for small programs ($O(100)$ lines) $5-20\%$ of the code remains to be inspected [2]. However, for large programs this fraction drops to less than a percent [3], making these techniques an interesting debugging aid.

2. ZOLTAR TOOLSET

The Zoltar toolset provides a blackbox method involving three phases: (1) instrumentation, (2) data gathering, and (3) data analysis.

With respect to the instrumentation phase, a program under analysis is instrumented using the LLVM [7] framework. The instrumentation phase also allows partial instrumentation of the program, in order to reduce run-time overhead (see Figure 1 for an overview on the instrumentation phase). The resulting instrumented executable has the same functionality as the original plus functionality to generate spectrum information of runtime behavior. Additionally, variables in the program can be instrumented to train program invariants, enabling automatic error detection.

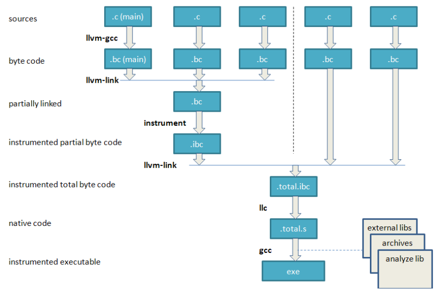


Figure 1: Instrumentation Phase

Running the instrumented program on test inputs results in the gathering of runtime data, which consists of the program spectra of different runs. 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. We have measured the run-time overhead to be 5% on average on our experiments with the Siemens benchmark set of programs [2].

The gathered data is essentially the pair (A, e) to be used as input for the fault localization techniques. This results in a ranking of fault locations, where the top rankings consists of locations most likely to contain the fault. The techniques provided by the Zoltar toolset are of low complexity (e.g., the BARINEL technique, the most expensive technique in the toolset, takes 41s to yield the diagnostic ranking for a 10KLOC-program). Refer to [3] for a detailed comparison of the techniques provided by Zoltar. Moreover, Zoltar provides tools to instrument a program, examine the program spectra, edit the program invariants of the instrumented program, and to visualize diagnostic results within the original source code (see [5] for detailed info).

3. RELATED WORK

Automatic software fault localization has been an active area of research for the past years, and several tools are publicly available. Similar to the Zoltar toolset, the following techniques do not use any prior knowledge of the system to produce the set of diagnosis candidates. Sober [9] and the Cooperative bug isolation [8] are statistical debugging tool which analysis traces fingerprints and produces a ranking of predicates by contrasting the evaluation bias of each predicate in failing cases against those in passing cases. Delta Debugging [11] compares the program states of a failing and a passing run, and actively searches for failure-inducing circumstances in the differences between these states. The Tarantula tool [6] is a statistics-based technique that takes as input abstraction of program traces and computes a list of diagnosis candidates. The Tarantula technique is also provided by the Zoltar toolset.

Although some other approaches to fault localization exist, such as model-based approaches [10], we refrain from discussing them here because (1) space constraints, and (2) as opposed to our diagnostic tool and the other approaches described in the previous paragraph, these approaches take not only observations, but also use previous knowledge of the program (e.g., a model) to reason over observed failures.

4. CONCLUSIONS

This paper describes Zoltar, a toolset for automatic software fault localization. The toolset as well as a comprehensive tutorial can be obtained from `www.fdir.org/zoltar`. Furthermore, refer to [3] for detailed information on the featuring technique provided by our toolset.

Adding a new techniques to the toolset is straightforward due to the design of the toolset [5], and that would provide researchers a common ground to compare their approaches. We plan to add other techniques to the toolset.

5. 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 *Proc. of ENASE'08*.

[2] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proc. of TAIC PART'07*.

[3] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. Spectrum-based multiple fault localization. In *Submitted to ASE'09*.

[4] 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.

[5] T. Janssen, R. Abreu, and A. van Gemund. The Zoltar toolset v1.0. http://www.fdir.org/zoltar, 2009.

[6] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proc. ASE'05*.

[7] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. CGO'04*.

[8] B. Liblit. Cooperative debugging with five hundred million test cases. In *Proc. ISSTA'08*.

[9] C. Liu, L. Fei, X. Yan, J. Han, and S. Midkiff. Statistical debugging: A hypothesis testing-based approach. *TSE*.

[10] W. Mayer and M. Stumptner. Evaluating models for model-based debugging. In *Proc. ASE'08*.

[11] A. Zeller. Isolating cause-effect chains from computer programs. In *Proc. FSE'02*.

APPENDIX

A. TOOL PRESENTATION

To illustrate the fault localization process using the Zoltar toolset, and more specifically the BARINEL approach, we consider three different programs, and we investigate various types of bugs. First, we consider a small thread-based program with two mutex related bugs. This illustrates the basic concepts and show the value of the toolset for this rather difficult to debug but common kind of fault. Second, we investigate two problems in `mplayer`, a program of realistic size. In particular, we investigate a GUI related bug and a problem which is caused due to corrupted input data. Finally, we investigate a fault multiplicity of five in the well-known `space` program.

A.1 textVal

Consider the example `textVal` program of which the pseudocode is given in Figure 2. 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 2: Pseudo code for the textVal program.

Two bugs related to mutual exclusion are introduced. 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 do the same, which is not always the case and which in practice results in difficult 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 will 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.

Instrumentation is performed on LLVM bytecode, which is obtained using the LLVM `gcc` frontend. An overview of instrumentation passes is given in Table 1. Some instrument the program to generate different kinds of program spectra, others instrument variables within the program for training program invariants to enable automatic error detection. Additionally, the Zoltar toolset provides instrumentation for protecting the part of memory in which the program spectra and invariant data are stored.

pass name	description
`-spfunction`	function level spectrum generation
`-spbasicblock`	basic block level spectrum generation
`-spdefuse`	spectrum generation of def-use pairs
`-invstore`	memory store value invariants
`-invload`	memory load value invariants
`-invfunctiontimer`	function execution time invariant
`-invloopcount`	loop iteration count invariant
`-memprotection`	protection of instrumentation data
`-mainbypass`	bypass of main and exit functions

Table 1: Overview of instrumentation passes.

Using Zoltar's `instrument` tool the `textVal` program is instrumented to generate a program spectrum on the basic block level and to be able to train function timer invariants using the following commands:

```
 # llvm-gcc -emit-llvm -g -c textVal.c -o textVal.bc
# instrument -f -invfunctiontimer -spbasicblock \
> -memprotection -bypassmain textVal.bc -o textVal.ibc
# llc -f textVal.ibc -o textVal.s
# gcc textVal.s -o textVal -lpthread -linstrument
```

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	< 920
test6.in	`".=+-#"`	50	hangs

Table 2: Test suite for the textVal program with information on the number of letters, digits and other characters.

Data is gathered by running the instrumented program on available test inputs. Table 2 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. By just running the instrumented program on the first four test inputs, (which result in normal perceived behavior of the program) the function timer invariants are trained. These trained ranges are then stretched to allow inputs of larger scale (the function execution time is input related). The instrumented program is then set from training mode to test mode, which enables the automatic error detection. The fifth test input returns a value lower than the expected value. The sixth test input causes the function timer invariant to trigger an error, because one of the functions keeps waiting for the mutex to become unlocked. The data gathering process results in the pair (A, e) of which a simplified[1] version is shown in Figure 3. 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.

	block				
test file	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
test6.in	1	1	0	1	1

Figure 3: Simplified (A, e) for the textVal program.

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 4. The data file contains program spectrum information, invariant data and the operating mode of the instrumented program, among other things.

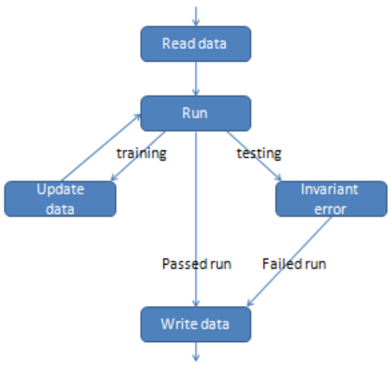
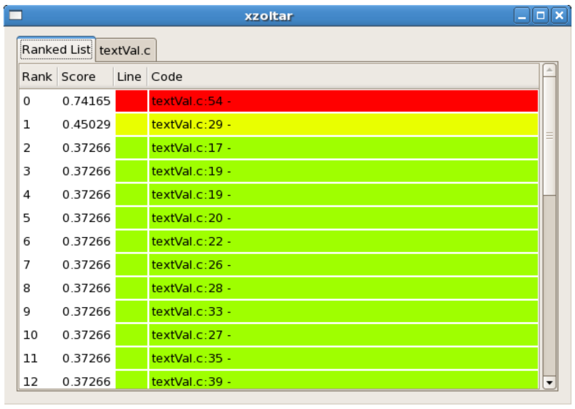


Figure 4: Zoltar data gathering workflow.

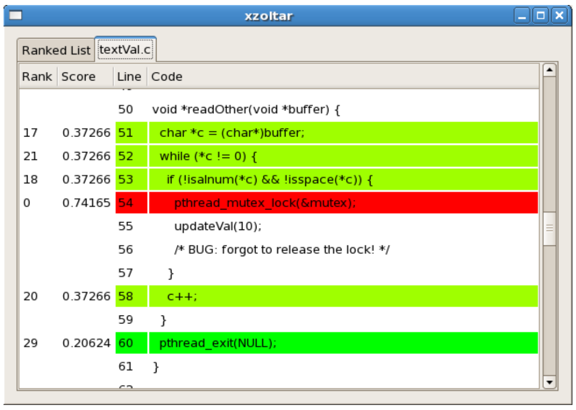
Two tools are provided for the data analysis. The `zoltar` tool is created to work in a console environment. This enables it to be used in many environments, including a remote shell. The `zoltar` tool provides an interface for the data file, can give a summary of the instrumentation of the program,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

An intuitive visualization is achieved with the `xzoltar` tool, which shows a graphical representation of the ranked list returned by the underlying diagnostic approach, where each line is color coded. A red line indicates that it is likely to be the location of the fault. A green line indicates that it has 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. A screenshot of the resulting ranking of basic blocks of the `textVal` case, using the Barinel approach, is given in Figure 5.



Rank	Score	Line	Code
0	0.74165		textVal.c:54 -
1	0.45029		textVal.c:29 -
2	0.37266		textVal.c:17 -
3	0.37266		textVal.c:19 -
4	0.37266		textVal.c:19 -
5	0.37266		textVal.c:20 -
6	0.37266		textVal.c:22 -
7	0.37266		textVal.c:26 -
8	0.37266		textVal.c:28 -
9	0.37266		textVal.c:33 -
10	0.37266		textVal.c:27 -
11	0.37266		textVal.c:35 -
12	0.37266		textVal.c:39 -

Figure 5: Visualization of Barinel ranking of the textVal case.

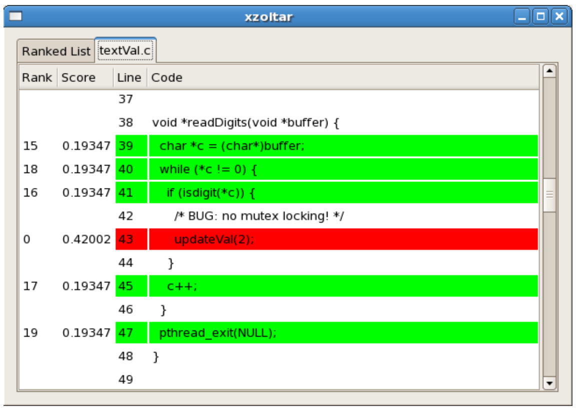


Rank	Score	Line	Code
		50	void *readOther(void *buffer) {
17	0.37266	51	char *c = (char*)buffer;
21	0.37266	52	while (*c != 0) {
18	0.37266	53	if (!isalnum(*c) && !isspace(*c)) {
0	0.74165	54	pthread_mutex_lock(&mutex);
		55	updateVal(10);
		56	/* BUG: forgot to release the lock! */
		57	}
20	0.37266	58	c++;
		59	}
29	0.20624	60	pthread_exit(NULL);
		61	}

Figure 6: Source code location of the first textVal bug.

Figure 6 shows the source code context of the highest ranking location. The indicated location is indeed the location of one of the bugs (the mutex is locked but not unlocked). After this bug is fixed, the tests can be repeated, resulting in another (A, e) which results in another Barinel ranking. The part of the source code with the highest rank-

[1]Other basic blocks are either never executed or always executed or are not represented in the pseudo code given in Figure 2.



Rank	Score	Line	Code
		37	
		38	void *readDigits(void *buffer) {
15	0.19347	39	char *c = (char*)buffer;
18	0.19347	40	while (*c != 0) {
16	0.19347	41	if (isdigit(*c)) {
		42	/* BUG: no mutex locking! */
0	0.42002	43	updateVal(2);
		44	}
17	0.19347	45	c++;
		46	}
19	0.19347	47	pthread_exit(NULL);
		48	}
		49	

Figure 7: Source code location of the second textVal bug after fixing the first bug.

ing location, after the first fault was fixed, is shown in Figure 7. This is the basic block in which the locking and unlocking of the mutex is neglected.

A.2 mplayer

To test the toolset on a more realistically sized program, we have instrumented `mplayer`[2], which totals around 650,000 LOC. We investigate two cases of unexpected behavior using `mplayer` version 1.0rc2:

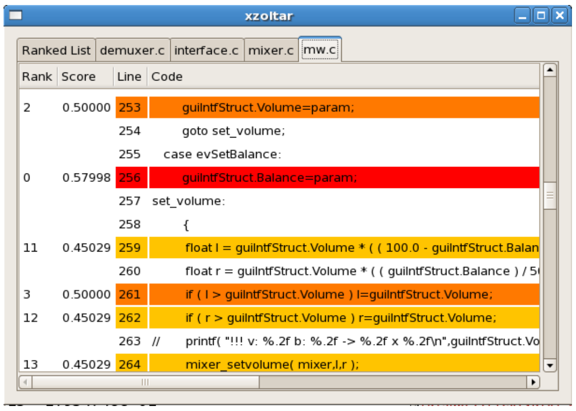
1. When using a GUI and changing the position of the volume slider while the balance slider is not at center, the position of the balance slider changes as well.
2. Using a particular .avi video file as input, changing the position in the file beyond some point causes the video to end immediately.

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. In total, approximately 100 kLOC of the 650 kLOC was instrumented. 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.

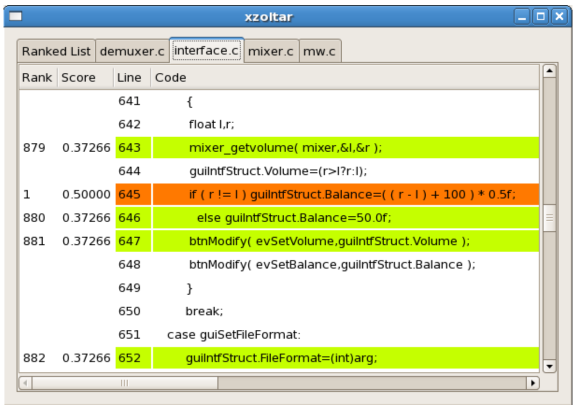
The code responsible for the first behavior was located by testing the instrumented executable with six different user inputs. Two test inputs caused the peculiar behavior and thus resulted in a failing run. The remaining four tests involved changing one or both of the sliders without the behavior appearing to the user, or not changing the sliders at all. Running the `xzoltar` tool afterwards resulted in a ranking of basic blocks, the relevant of which are distributed over just 4 of the instrumented 132 files. One basic block at the top of the BARINEL ranking is involved in handling 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. This is shown in Figure 8. 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. This part of the code is shown in Figure 9. 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.

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



Rank	Score	Line	Code
2	0.50000	253	guiIntfStruct.Volume=param;
		254	goto set_volume;
		255	case evSetBalance:
0	0.57998	256	guiIntfStruct.Balance=param;
		257	set_volume:
		258	{
11	0.45029	259	float l = guiIntfStruct.Volume * ((100.0 - guiIntfStruct.Balan
		260	float r = guiIntfStruct.Volume * ((guiIntfStruct.Balance) / 5
3	0.50000	261	if (l > guiIntfStruct.Volume) l=guiIntfStruct.Volume;
12	0.45029	262	if (r > guiIntfStruct.Volume) r=guiIntfStruct.Volume;
		263	// printf("!!! v: %.2f b: %.2f -> %.2f x %.2f\n",guiIntfStruct.Vo
13	0.45029	264	mixer_setvolume(mixer,l,r);

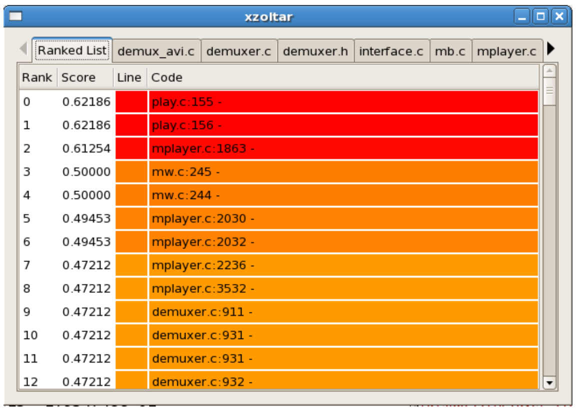
Figure 8: Relevant part of the source code of the mplayer GUI behavior.



Rank	Score	Line	Code
		641	{
		642	float l,r;
879	0.37266	643	mixer_getvolume(mixer,&l,&r);
		644	guiIntfStruct.Volume=(r>l?r:l);
1	0.50000	645	if (r != l) guiIntfStruct.Balance=((r - l) + 100) * 0.5f;
880	0.37266	646	else guiIntfStruct.Balance=50.0f;
881	0.37266	647	btnModify(evSetVolume,guiIntfStruct.Volume);
		648	btnModify(evSetBalance,guiIntfStruct.Balance);
		649	}
		650	break;
		651	case guiSetFileFormat:
882	0.37266	652	guiIntfStruct.FileFormat=(int)arg;

Figure 9: Related source code ranked next explaining the GUI behavior further.

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 total there were ten passed runs during training and four failed runs in the operational stage. After the data gathering process BARINEL resulted in a ranked list shown in Figure 10. Five files of the 132 instrumented files are in the top five ranking (different basic blocks can get the same score, when they are executed consecutively in all tests). All locations are related to the seeking of a video file. The top three locations are small pieces of code which call a seek function or are the result of seeking until the end of the file. These locations can be quickly inspected.



Rank	Score	Line	Code
0	0.62186		play.c:155 -
1	0.62186		play.c:156 -
2	0.61254		mplayer.c:1863 -
3	0.50000		mw.c:245 -
4	0.50000		mw.c:244 -
5	0.49453		mplayer.c:2030 -
6	0.49453		mplayer.c:2032 -
7	0.47212		mplayer.c:2236 -
8	0.47212		mplayer.c:3532 -
9	0.47212		demuxer.c:911 -
10	0.47212		demuxer.c:931 -
11	0.47212		demuxer.c:931 -
12	0.47212		demuxer.c:932 -

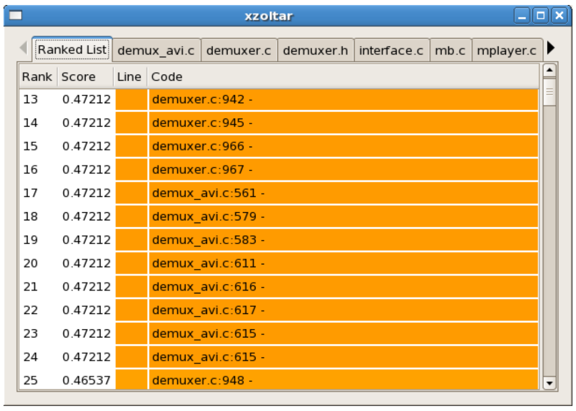
Figure 10: Ranked list of the basic blocks related to the second mplayer behavior.

More interesting are the locations at rank 5 (shown in Figure 11), belonging to a function in `demuxer.c` and a function in `demux_avi.c`. After investigating the latter, it becomes clear that an input video is used, which contains only the first few video key frame indices, but lacks the rest. In this part of the code, the program continues to skip frames until a video key frame is found, or the end of the file is reached. In the case of the input file in the test this results in abrupt ending of the video. Without detailed knowledge of the sources, we were able to find the location responsible for handling the data error. Search effort was limited to only a limited amount of source code locations.

The described `mplayer` cases show that the Zoltar toolset is also suitable for analyzing behavior of large programs. 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.

A.3 space

The `space` program is an interpreter for an array definition language (ADL), which was developed for the European Space Agency. This C program totals just over 6,000 LOC and there are 38 known faults. The `space` benchmark package provides 1,000 test suites that consist of a random selection of (on average) 150 test cases out of 13,585 and

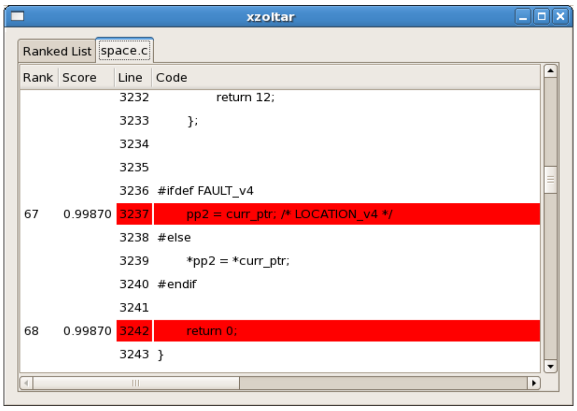


Rank	Score	Line	Code
13	0.47212		demuxer.c:942 -
14	0.47212		demuxer.c:945 -
15	0.47212		demuxer.c:966 -
16	0.47212		demuxer.c:967 -
17	0.47212		demux_avi.c:561 -
18	0.47212		demux_avi.c:579 -
19	0.47212		demux_avi.c:583 -
20	0.47212		demux_avi.c:611 -
21	0.47212		demux_avi.c:616 -
22	0.47212		demux_avi.c:617 -
23	0.47212		demux_avi.c:615 -
24	0.47212		demux_avi.c:615 -
25	0.46537		demuxer.c:948 -

Figure 11: Relevant basic blocks ranked somewhat lower.

guarantees that each branch of the program is executed by at least 30 test cases.

To show the performance of BARINEL on multiple faults we combined five known faults into one source file and we run a randomly chosen test suite (of the 1,000) to obtain a ranking of fault locations. These fault locations are at the statement level, which is more fine-grained that the basic block level discussed previously. Figure 12 shows the location of the fault (line 3,237) that scored highest in the ranking. It is shown in position 67 in the list (of the 6,218), however the actual score returned from BARINEL is second in rank, and the first locations that are ranked first are distributed in only four functions and form larger blocks of statements. This leads to less investigation effort than the 67th place in the ranking suggests.



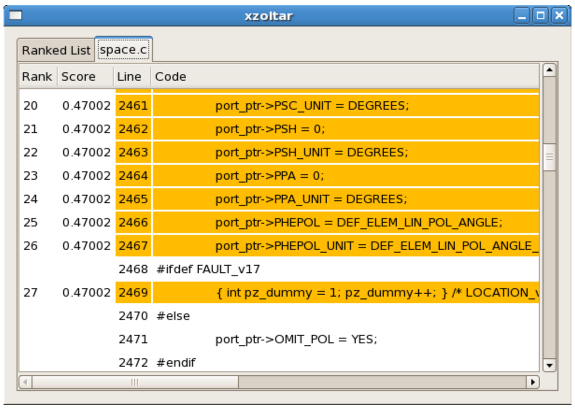
Rank	Score	Line	Code
		3232	return 12;
		3233	};
		3234	
		3235	
		3236	#ifdef FAULT_v4
67	0.99870	3237	pp2 = curr_ptr; /* LOCATION_v4 */
		3238	#else
		3239	*pp2 = *curr_ptr;
		3240	#endif
		3241	
68	0.99870	3242	return 0;
		3243	}

Figure 12: Location of the first of five bugs present in space.

When the first bug was fixed, the tests were run again and this still resulted in some failures, yielding another ranked list of locations. The actual fault scoring highest in this case is at position 73 in the ranked list, but, similar as in the first case, only four larger blocks of code need investigation before getting to the fault location.

Fixing this bug and running the tests for the third time resulted in two of the faults sharing the highest ranking score in the list. Line 5,201 and line 5,516 were located within the five larger blocks of code in the highest ranking, which sum up to 113 of the 6,218 executable lines of code.

Finally, after fixing the previous two bugs, the bug in line 2,469 appeared in the third ranking block of code. The location in the code is shown in Figure 13. It would in fact be the twenty-eighth line to investigate, if all lines were investigated in order of appearance in the ranked list. The lines are less red in this result, because the line that ranked first scored rather high in comparison (0.74 vs. 0.47).



Rank	Score	Line	Code
20	0.47002	2461	port_ptr->PSC_UNIT = DEGREES;
21	0.47002	2462	port_ptr->PSH = 0;
22	0.47002	2463	port_ptr->PSH_UNIT = DEGREES;
23	0.47002	2464	port_ptr->PPA = 0;
24	0.47002	2465	port_ptr->PPA_UNIT = DEGREES;
25	0.47002	2466	port_ptr->PHEPOL = DEF_ELEM_LIN_POL_ANGLE;
26	0.47002	2467	port_ptr->PHEPOL_UNIT = DEF_ELEM_LIN_POL_ANGLE_
		2468	#ifdef FAULT_v17
27	0.47002	2469	{ int pz_dummy = 1; pz_dummy++; } /* LOCATION_\
		2470	#else
		2471	port_ptr->OMIT_POL = YES;
		2472	#endif

Figure 13: Location of the last bug in space.

Zoltar was shown to be able to handle multiple-fault scenarios, aiding in the debugging of each fault separately. After a bug has been fixed, the number of failing runs will typically be reduced, which results in another input matrix and a new ranking to help locate another fault. Little effort was needed to investigate the limited amount of code in the top ranking, since the fault resided in the top 5 blocks of code in each of the discussed rounds of debugging.

A.4 Tool maturity and availability

Currently, the Zoltar toolset is at a beta stage. The developers package can be downloaded from

`http://www.fdir.org/zoltar`

It is currently only tested on a Linux environment. Zoltar depends heavily on the LLVM framework and as a result languages that are supported by Zoltar are the languages supported by the LLVM frontend, although only C sources are currently tested. A tutorial is made available to get acquainted with the toolset, which can be downloaded from the same site. The Zoltar toolset can easily be extended by adding new program instrumentations or other spectrum-based calculations for returning a ranked list of fault locations.