Understanding Failures Through Facts

Jeremias Rößler
Saarland University, Saarbrücken, Germany
roessler@cs.uni-saarland.de

ABSTRACT

"Why does my program crash?"—This ever recurring question of software debugging drives the developer during the analysis of the failure. Complex defects are impossible to automatically identify; this can only be left to human judgment. But what we can do is empower the developer to make an informed decision, by helping her understand the failure. To fully comprehend a failure, one may need to consider many different aspects such as the range of the input parameters and the program's structure and runtime behavior. I propose an approach that gathers a variety of such *facts* from a given failing execution. To examine the correlation of those facts to the failure, it produces additional executions that differ in as few facts as possible. Then the approach creates generalizations and abstractions over the correlating facts. These explain different aspects of the failure and thus help the developer understand and eventually fix the underlying defect.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Diagnostics, Symbolic execution, Testing tools*

General Terms

Experimentation

Keywords

Automated debugging, test case generation, failure classification, statistical debugging

1. INTRODUCTION

What is special about Brazil as a time zone in regard to a particular date? And why does this make my program crash? These are the questions a developer might ask in order to understand the failure that is reproduced by the test case shown in Figure 1. This test case produces a certain date and transforms this date to a time interval that represents the 24 hours of that date in the Brazilian time zone. If a developer wants to write a fix for the underlying defect or even only assess it, she has to investigate the code and do some manual debugging to understand how the failure comes into existence. And since this three lines long test case invokes 367 methods and executes 1,465 lines of code (ignoring JAVA system libraries), she might have a hard time doing so.

```
public class JodaTest extends TestCase {
  public void testBrazil() {
    DateTimeZone dtz =
      DateTimeZone.forID("America/Sao_Paulo");
    LocalDate date =
      new LocalDate(2009, 10, 18);
    Interval interval = date.toInterval(dtz);
  }
}
```

Figure 1: JODA TIME bug 2487417: `toInterval()` **fails when processing October 18, 2009, Brazil time.**

The state of the art in automated debugging focuses on defect localization—the reporting of individual faulty lines of code. The evaluation of those approaches usually assumes an "ideal user" [28], who immediately recognizes the defect on sight. This convenient assumption allows for an easy and objective evaluation. But this assumption is also very questionable [25]: After some time has passed, developers usually do not understand their own code, let alone someone else's. If only the problematic lines of code are presented without explanation or context, *developers won't recognize the defect.* And even if the code is almost correct and the fix is simple, without understand the failure *developers cannot write the fix.* As for the example in Figure 1: in such a case defect localization does not help at all. The fix involves an API change with deprecation of several methods and classes, and the development of new classes and methods to replace them. For such cases most approaches to defect localization do not even fail well—they are not designed to detect when the fix involves more than a single line of code.

In contrast, little effort has been spent to help developers understand a failure. And for an obvious reason: the evaluation depends on human feedback and thus is much more laborious and much less objective. Yet this is a much better indicator of the overall goal of developer productivity. In the following, I will introduce an approach that is capable of extracting different *facts*: structural and behavioral runtime information that helps the developer understand different aspects of the failure.

2. EXPLANATORY FACTS

In previous work [29] the goal was to extract failure conditions in the form of constraints on the input. These constraints are helpful when assessing the frequency and severity of the failure. Yet they do not suffice to explain the failure in terms of the internals of the program. To fully understand a failure, one has to take into account various aspects, such as input, program structure and runtime behavior both in terms of control flow and data flow. Information about these aspects is represented by *facts* of different types that complement and confirm each other and are helpful in diverse scenarios. Such facts are:

Constraints on the input: The input-related path conditions, as they are encountered during execution, form constraints on that input. The proposed approach tries to extract 1-minimal [35] failure conditions that explain the failure in terms of the input (as shown by previous work [29]). For the initial example as given in Figure 1, these failure conditions are

$$\mathit{isDaylightCutDay}() \wedge (\mathit{cutoverTime} = \mathit{00{:}00{:}00})$$

That is, the failure occurs whenever the chosen date is a daylight cut day in the corresponding time zone and the cutover time is midnight.

Variable values: The relations and abstractions of the values of variables may also point to the underlying problem. For the initial example, the value of

```
dateTZ.nextTransition(date.getLocalMillis())
```

has always the same relation to the chosen date: It is the date of the next daylight savings time transition, whenever that transition is at 00:00:00. That is in line with the constraints on the input as mentioned above.

Invariants: Program invariants [13] that hold for failing runs, but never for passing runs or vice versa often reveal some interesting properties. For the initial example, invariants of passing runs that are violated by failing runs are:

$\mathit{millisLocal} = \mathit{millisUTC} + \mathit{getOffset}(\mathit{millisUTC})$
$\mathit{millisUTC} = \mathit{millisLocal} - \mathit{getOffsetFromLocal}(\mathit{millisLocal})$

This means that the local time is always the universal time plus offset, where the universal time is always the local time minus local offset. For the faulty runs, this mapping is inconsistent, since the created local time maps to some universal time, but this universal time in turn maps back to a different local time.

Definition-usage pairs: The definition and later usage of a variable in the code gives a good understanding of how the data flows through an execution of the program and what data dependencies exist. If the problematic definition and later usage of a value are far away from each other, which is especially common in object oriented programs, that connection can be important to show. For the initial example, such definition-usage pairs could be used to show the origin of the invariant violation as given above:

```
getOffsetFromLocal(millisLocal) !=
            getOffset(millisLocal -
               getOffsetFromLocal(millisLocal))
```

This is the guarding condition of an internal consistency check which realizes that the start of the day (midnight) does not exist in the chosen time zone and raises an exception.

Executed branches: The correlation between executed branches (branch coverage) and defects is an often chosen indicator to single out problematic code. Yet for the given example, this shows a problem that arises when only considering a single aspect of the execution: the code has several places where daylight savings time transitions are treated differently, and due to the nature of the problem, all those branches are highly correlated to the failure. Only when also considering other aspects, it becomes clear which of these branches are really relevant for the failure and which are just coincidentally correlated.

Number of loop-iterations: When it comes to loops, path conditions may not be enough information; the developer might also want to know how often a loop was executed. For example, in JODA TIME, there is a piece of code where two dates from different APIs are aligned:

```
while (date.getDate() == dom) {
  date.setTime(date.getTime() - 1000);
}
```

In that situation, the developer wants to know whether the number of loop iterations is correlated to the failure. For instance, it would be interesting if the loop is always executed 3.600 times (representing one hour).

The facts are gathered during execution. Every execution produces many facts of each type. As most of them are irrelevant for the failure, the next step is to identify facts that are related to the failure and discard facts that are not. To achieve this, I propose an approach of systematic experimentation that works for all types of facts: for each fact it tries to come up with an execution that differs in only the selected fact. If the failure still occurs, the fact is not related; otherwise it is. Of course, for many cases such an execution does not exist. The solution is to generate several executions where each differs in as few facts as possible, but those facts are as divers among the executions as possible. These executions can be used to calculate the correlation of the facts to the failure [2].

This approach is an extension of previous work [29]. The idea to examine the constraints individually by generating additional executions nicely generalizes to all types of facts. Formerly this was done by changing the inputs to the program under test using a constraint solver. Now the constraint solver is embedded into a genetic algorithm [24] to also change the test. This approach partly explores the execution space. The alternative is to explore the complete execution space by applying symbolic execution [20]. Symbolic execution suffers from the problem of state explosion [31]. For most real-life programs the execution space is infinite and its complete exploration infeasible. Because of this, concrete executions were chosen over symbolic execution. But since the execution space is only partly explored, the proposed approach is not complete. For instance it generates a set of constraints on the input that could possibly be further reduced; thus it does not produce the weakest precondition [12]. In return the approach scales much better and is applicable to programs of any size. It starts with a concrete failure and iteratively expands to additional executions, adding more information and increasing the precision in the process. This allows to abort at any stage and still get some results.

After identifying correlating facts, the approach generalizes over them and tries to find abstractions. This is done differently for every type of fact. In many cases, one can leverage the names of the methods the facts are generated in, as well as names of variables and constants. The idea is to reuse as much domain specific semantic information from the code as possible.

The less non-related facts there are to start with, the faster and more precise are the results. Thus techniques that reduce the size or the trace of the execution could be applied to improve the approach. One way to do this is to create the dynamic backward slice [33] of the failing statement. Another approach as proposed by Burger and Zeller [7] minimizes object interactions. After the general applicability of the initial approach has been shown, such techniques could be used to increase its efficiency and the precision of the results.

3. RESEARCH HYPOTHESES

The overall goal is to extract different *facts* and abstractions thereof, that help the developer understand the failure. Of course, helpfulness is not a measurable quantity so it is not directly and objectively verifiable whether the approach meets its goal. Perhaps this is one of the reasons, why this goal has received so little attention in the past. For the proposed approach, two things are done to remedy the problem: A quantitative evaluation is performed on defects for which the fix is available. And a qualitative evaluation is performed manually on defects that are resolved and the reason for the failure is filed.

The ground truth in automated debugging is always the fix. All the facts that the approach reports are somehow related to the failure, meaning that they are part of the chain of causation. The fix breaks this chain such that the failure does not occur anymore. This leads to the first falsifiable research hypothesis. **RH1: Once the fix is applied, the reported facts are not observable anymore**. So the strategy to quantitatively evaluate the proposed approach is to apply it to some defects for which a fix is available. Such defects are contained in the iBugs project [11], which provides a benchmark of real-life subjects with real bugs. When the approach is applied to the defects of that project, the results and the generated executions are recorded. Then the fixes are applied to the defects and the generated executions are performed again. Now the previously reported facts should not be observable anymore. However, this evaluation strategy has a problem: the reported facts might stem from earlier in the chain of causation than where the fix is later applied. This means that the chain is only broken after the facts are observed. So even though the failure is not observable anymore, the facts still are. It is yet unclear how prevalent this problem is. Therefore it will be dealt with when it occurs during evaluation.

Since the generalizations and abstractions produced in the last step involve domain specific semantic information, they cannot be verified automatically. Therefore, I perform a manual and subjective investigation to confirm the last research hypothesis. **RH2: The generalizations and abstractions over the facts are correct and helpful to understand the failure.** For a body of known faults, I manually inspect the results of the approach and, only using that information, try to understand the failure. Then I compare this understanding to the description of the defect as I find it in the bug database and how it was fixed in the code. This non-repeatable evaluation qualitatively indicates the success of the approach.

4. RELATED WORK

Automated debugging is a very dense field and depending on which aspects are emphasized, the proposed approach is related or comparable to many others.

The intuition of informative *facts* about the runs that are presented to the developer is comparable to the facts LaToza and colleagues [22] observed when monitoring developers as they tried to understand foreign code. This notion is mirrored in other contexts as “spectra”, “features” or “execution profiles”. Harrold and colleagues [16] give a very good overview and classification of the different possible types of program spectra. Santelices and colleagues [30] find that for a set of lightweight fault-localization techniques, a combination of different types of program spectra (such as we use) outperforms individual approaches.

Much work in the context of debugging is targeted at defect localization. Since most of these approaches are evaluated on the Siemens Suite [17] their performance and thus the rising precision of the state of the art is easy to follow. Agrawal and colleagues [3] created a tool termed χslice that produces dices (set difference of dynamic slices) to localize defects. Renieris and Reiss [28] try to localize the fault by comparing the coverage spectra of a failing run to its “nearest” passing run (according to their distance metric). Pytlik and colleagues [26] do preliminary work with two types of simple variable invariants. Given a counterexample trace of a model checker, Groce’s Explain tool [14] creates a passing run with minimal distance and extracts a specialized dynamic slice. Liblit and colleagues [23] do statistical debugging with remotely recorded predicate evaluations. Chilimbi and colleagues [8] do the same, but use path profiles instead. Dallmeier and colleagues [10] use method call sequences to identify defective classes. Zeller [34] and Cleve and Zeller [9] transfer data from a healthy to an infected program state to extract cause-effect chains and cause transitions. Jones and colleagues [18] changed their Tarantula tool to localize the defect according to the number of passing and failing test cases a certain block of code participates in. Abreu and colleagues [2] perfect this approach by creating a detailed mathematical probabilistic model of a component chain. Qi and colleagues [27] use the path conditions of a correct and a faulty version of a program to generate an additional execution path and present differences as possible locations of the defect. The Deputo tool of Abreu and colleagues [1] combines a spectrum-based approach with model-based diagnosis. Baah and colleagues [6] use a probabilistic program dependence graph whose probabilities are based on observed executions and causal effect estimation to rank executed statements. The approaches to defect localization show an impressive advancement in the field and many of the ideas can be reused in the context of defect explanation.

Current research takes this idea even one step further: Instead of only locating the failure, some approaches try to automatically come up with fixes for it. For instance Weimer and colleagues [32] use genetic programing to generate fixes and Arcuri and Yao [4] try to co-evolve the program and its test suite. Naturally these approaches suffer from the same shortcomings as defect localization approaches. For a developer to accept a proposed fix, he first has to understand the underlying problem—which is not facilitated.

In contrast to the maturity of the defect localization discipline, only few approaches try to help the developer understand a failure. Originally, "Tarantula" should visualize test information and highlight suspicious code [19], but without further explanation. The cause-effect chains and cause transitions of Cleve and Zeller [9] explain a failure in terms of how the data flows through a program execution but ignore other important aspects. The probabilistic program dependence graphs from Baah and colleagues [5] can also be used to comprehend a fault but need further improvement in that regard. Other techniques aim at the same goal, but take a dialog-oriented approach, rendering them completely different. Whyline is a tool from Ko and colleagues [21] that allows developers to ask questions about the visual output of the program based on a recorded execution. Hao and colleagues [15] implemented a tool that suggests breakpoints to the developer for an interactive localization of the defect.

5. CONCLUSIONS

In this paper I argued for the need to help the developer understand a failure, enabling her to make informed decisions. The contribution of this paper is the proposal of a new approach that, given a failing execution, gathers *facts* that capture important aspects of the input, the structure and the runtime behavior of the program. Additional executions are generated to examine the correlation of those facts to the failure. The approach reuses semantic information found in the source code of the program to abstract and generalize over the correlating facts. Those abstractions and generalizations help the developer understand the failure. The overall goal is an explanation rather than a localization of the underlying defect.

6. ACKNOWLEDGEMENTS

I want to thank my advisors Andreas Zeller and Alessandro Orso for their continuing support and patience. I had valuable discussions with Kevin Streit, Clemens Hammacher and Kim Herzig. David Schuler and Yana Mileva provided helpful comments on earlier revisions of this paper. The anonymous reviewers pointed out some additional related work.

7. REFERENCES

[1] R. Abreu, W. Mayer, M. Stumptner, and A. J. C. van Gemund. Refining spectrum-based fault localization rankings. In *SAC*, pages 409–414, 2009.

[2] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. An observation-based model for fault localization. In *WODA*, pages 64–70, 2008.

[3] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. In *ISSRE*, pages 143–151, 1995.

[4] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *IEEE Congress on Evolutionary Computation*, pages 162–168, 2008.

[5] G. K. Baah, A. Podgurski, and M. J. Harrold. The probabilistic program dependence graph and its application to fault diagnosis. In *ISSTA*, pages 189–200, 2008.

[6] G. K. Baah, A. Podgurski, and M. J. Harrold. Causal inference for statistical fault localization. In *ISSTA*, pages 73–84, 2010.

[7] M. Burger and A. Zeller. Minimizing reproduction of software failures. In *ISSTA*, 2011. to appear.

[8] T. M. Chilimbi, B. Liblit, K. K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *ICSE*, pages 34–44, 2009.

[9] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, pages 342–351, 2005.

[10] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In *ECOOP*, pages 528–550, 2005.

[11] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *ASE*, pages 433–436, 2007.

[12] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Inc., 1976.

[13] M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington, 2000.

[14] A. Groce. Error explanation with distance metrics. In *TACAS*, pages 108–122, 2004.

[15] D. Hao, L. Zhang, T. Xie, H. Mei, and J. Sun. Interactive fault localization using test information. *J. Comput. Sci. Technol.*, 24(5):962–974, 2009.

[16] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. In *PASTE*, pages 83–90, 1998.

[17] M. Hutchins, H. Foster, T. Goradia, and T. J. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE*, pages 191–200, 1994.

[18] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, pages 273–282, 2005.

[19] J. A. Jones, M. J. Harrold, and J. T. Stasko. Visualization of test information to assist fault localization. In *ICSE*, pages 467–477, 2002.

[20] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[21] A. J. Ko and B. A. Myers. Finding causes of program output with the java whyline. In *CHI*, pages 1569–1578, 2009.

[22] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers. Program comprehension as fact finding. In *ESEC/SIGSOFT FSE*, pages 361–370, 2007.

[23] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, pages 15–26, 2005.

[24] J. Malburg and G. Fraser. Combining search-based and constraint-based testing. In *ISSTA*, 2011. to appear.

[25] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, 2011. to appear.

[26] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. In *AADEBUG*, pages 273–276, 2003.

[27] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: an approach for debugging evolving programs. In *ESEC/SIGSOFT FSE*, pages 33–42, 2009.

[28] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 30–39, 2003.

[29] J. Rößler, A. Orso, and A. Zeller. When does my program fail? In *CSTVA*, 2011. to appear.

[30] R. A. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *ICSE*, pages 56–66, 2009.

[31] A. Valmari. The state explosion problem. *Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models*, 1491:429–528, 1998.

[32] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE*, pages 364–374, 2009.

[33] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10:352 –357, July 1984.

[34] A. Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT FSE*, pages 1–10, 2002.

[35] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28:183–200, February 2002.