Debugging with Software Visualization and Contract Discovery

S. Kanat Bolazar, James W. Fawcett
Electrical Engineering and Computer Science
Syracuse University, Syracuse, NY 13244, USA
kanat2@yahoo.com, jfawcett@twcny.rr.com

Abstract

Despite advances, debugging remains a drudgery, especially for software without proper documentation. Both top-down visualization-based approaches and bottom-up programmatic automation can improve this process. For a given system under test (SUT), we propose a combined method of:

- Program trace visualization, with capability to zoom in to method call details (inputs and outputs)
- Interactive declaration of observables for the internal state of the system
- Statistical and visual analysis of the collected observable data
- Programmatic declaration of expected behavior of the system defined through invariant relationships and satisfaction of contracts (Design by Contract) on these observables
- The ability to modify and repeat these steps without restarting the SUT

We report that instrumentation necessary to collect trace data is feasible; large amounts of data can be gathered without significant performance penalty while the visualizer remains responsive to tester interaction. Our personal experience is that the system is very quick to set up, faults are discovered quickly, and inefficient algorithms (which may produce correct results) become obvious through the visualizer. We are setting up human-interaction experiments to support the claim that our approach improves the efficiency of discovery of fault origin for a given system failure, compared to using a state-of-the-art debugger for Java (Eclipse).

1. INTRODUCTION

Today's debuggers are becoming more powerful with many features that decorate the standard debugging paradigm. Popular IDEs such as Eclipse and IntelliJ IDEA improve efficiency by instant syntax checking, incremental compilation, and automated refactoring macros, but the debugging paradigm is not improved: At a given time, these IDEs still only allow the developer to see a single point in the execution time of the program, and a single point in the space of source code. This is in part a fundamental limitation due to dependence on symbolic representations whereas visualization could make larger patterns of program behavior visible.

Bidirectional debugging [1, 2] works by checkpointing to save the complete state of the SUT at certain intervals during its execution so that the debugger can step back, as well as forward, to any point in the execution of the SUT. This approach reduces the effort involved in tracing back from the manifestation of a failure to the original fault that caused the failure. Bidirectional debugging also removes the fear of stepping past an important point in the execution. For hard-to-replicate internal states of the SUT, this fear may cause the tester to take infinitesimally small steps forward thereby increasing the total effort expended.

But, for a large program, the whole internal state of the SUT is often too large to save in full. Also, the environmental side-effects such as deletion of a file are not always reversible.

Design by Contract (DBC) is a semi-formal approach for software requirements specification which is also used in creation of automated test oracles [3]. In two controlled experiments, Mueller et al. discovered that DBC improved cross-developer code reuse, reliability, and maintenance efficiency, but the initial development phase took longer [4]. Delaying the initial development has the disadvantage of delaying user feedback that can be utilized to improve the usability of the system. Instead of full-coverage starting with the initial development, we suggest an approach that allows selective just-in-time retrofitting of contracts onto a system developed (or otherwise acquired) to have no contracts. This approach can improve accuracy in a way similar to how profiling can improve efficiency. This way, important gains in comprehensibility and maintainability may also be achieved without significant start-up costs.

Execution trace visualizers such as Gammatella [5] use information murals [6] – two-dimensional canvases where the hue, saturation and brightness information encode specific features of the collected data. This allows very compact representation of vast amounts of data.

Our approach combines program understanding through the use of an execution trace visualizer with partial test automation through DBC-assertions for test oracle specification. A tester-selectable subset of the internal state information (we focus on data transfers during method calls) is stored and failures are highlighted. Our system does not require restarting the SUT, and target-language code is used to define observables and contracts during execution. The tester can use the tool to trace the execution backwards in time from failure to fault origin.

For the visualizer, we aim to minimize the user effort and maximize usable visual information, to make larger structures evident. Vision as an older evolutionary development is a more natural and intuitive way to absorb large amounts of data. Symbolic representation is a more recent construct. A higher level programming language is powerful, and a program demonstrates this power, when generalization and abstraction can be used in automation, as we attempt to do with our system.

During debugging, a software engineer constantly revises her or his incomplete and possibly wrong internal representation of the SUT and hypotheses about both the desired and the actual behavior of the SUT.

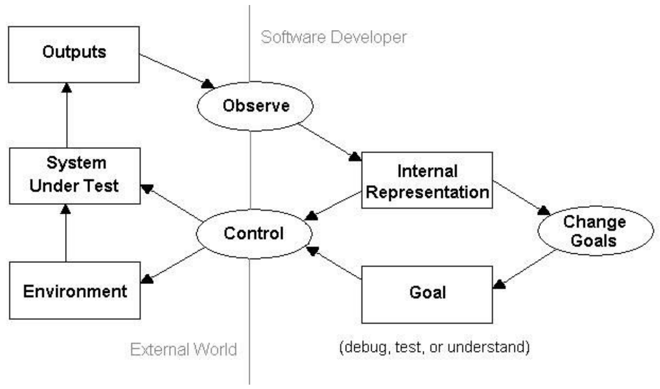


Figure 1. Interactions between a software engineer and a SUT, during debugging. Rectangles represent the tangibles and ellipses represent human actions.

With a debugger, much of the internal state of the program that is not ordinarily observable or modifiable becomes both observable and modifiable.

2. OUR APPROACH

Our goal is explicit representation of hypotheses and use of visualization and symbolic automation for comprehension and verification. We allow tracing the SUT while it is running, and the process can be repeatedly performed without restarting the SUT. We describe our proposed approach in detail in the following sections.

2.1. Tracing Sessions, Intercepting Method Calls

For each tracing session, the tester selects the methods to trace, starts tracing, interacts with the SUT to recreate the erroneous behavior, and stops tracing. This process can be repeated without restarting the SUT. Each session is recorded in a single execution trace record.

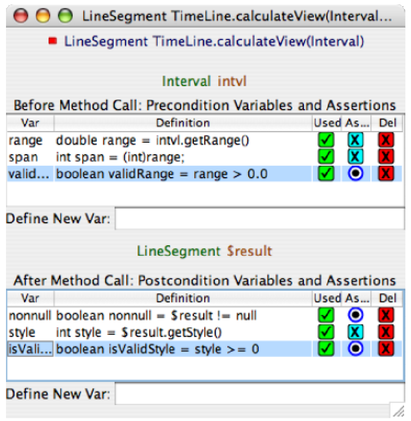
As a method is usually the smallest unit of interest for testing, our current implementation intercepts method calls. This means that our tracer can insert code before, after, or instead of selected method calls, but not at any other point in the program execution. Note that this is not merely blackbox testing. Within a method, any call to any traced method will also be caught and can be separately instrumented.

Our implementation uses AspectJ to intercept method calls. AspectJ [7] is an Aspect-Oriented Programming (AOP) framework for Java which can perform load-time weaving of bytecode. This means source code is not needed; in a library which is shipped without source code, we can instrument and trace the methods of the classes in the library, including intra-library method calls if desired.

2.2. Tester-Defined Observables and Method Contracts

The standard observables we save for a method call are its parameters, "this" and "that" references, and the return value. The “that” reference is the “this” reference of the caller. Beyond these standard observables, the tester can define *tester-defined observables* (fig. 2) using arbitrary Java code that will be evaluated before (precondition observables) and after (postcondition observables) a method call. Beyond variables whose values will be stored, print statements, loops to display or aggregate values of an array, code to pop open a new GUI frame, or dialog can also be used. We currently also allow calling private methods and accessing private fields of any object. We use Beanshell to parse and interpret these expressions for each method call, without need for compilation.

Without expectations, one cannot separate normal from unusual behavior. In our system, for each traced method, the tester can specify the expected behavior of the method in terms of the observables, using "method contracts" that consist of DBC pre- and postconditions. We use programmatic syntax for these contracts (fig. 2); the tester defines one Boolean variable for each expected behavior assertion, and declares that it should always hold true. For some common tasks, we use macros that expand to code.



LineSegment TimeLine.calculateView(Interval)

Interval intvl

Before Method Call: Precondition Variables and Assertions

Var	Definition	Used	As...	Del
range	double range = intvl.getRange()	✓	X	X
span	int span = (int)range;	✓	X	X
valid...	boolean validRange = range > 0.0	✓	●	X

Define New Var:

LineSegment $result

After Method Call: Postcondition Variables and Assertions

Var	Definition	Used	As...	Del
nonnull	boolean nonnull = $result != null	✓	●	X
style	int style = $result.getStyle()	✓	X	X
isVali...	boolean isValidStyle = style >= 0	✓	●	X

Define New Var:

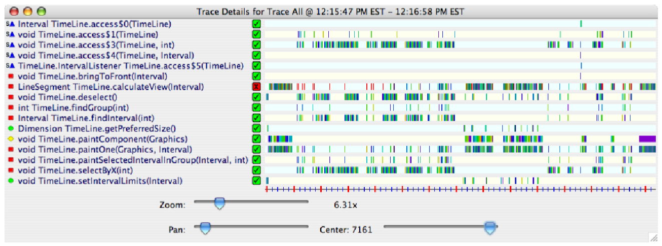
Figure 2. Method contracts use tester-defined Boolean variables to specify the preconditions and the postconditions asserted (middle column).

2.3. Visualizer

While the SUT is still being traced, summary statistics are displayed and updated in real time. Any failure is highlighted in red. This allows the tester to see instantly that some methods have failed, possibly due to the last tester interaction with the SUT.

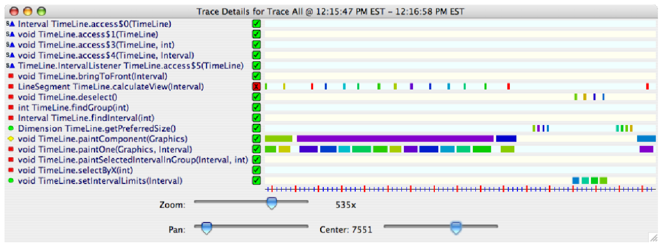
The execution trace record visualizer gathers and displays individual method calls over time as an execution mural. Our visualizer also allows zooming in to a selected method in order to inspect the details of that method call.

In the visualizer, we use discriminable colors from a relatively cool color spectrum (blue-cyan-green-yellow segment of the spectrum) to depict any method call which conforms to expectations. Those method calls which have any failed assertions are shown in red. If any call to a method has a failed assertion, we also highlight that method by displaying a red cross mark next to the method signature (figs 3, 4). Execution time axis, displayed as tick marks, uses tool tips to interactively display the time.



Trace Details for Trace All @ 12:15:47 PM EST – 12:16:58 PM EST
Interval TimeLine.access$0(TimeLine)
void TimeLine.access$1(TimeLine)
void TimeLine.access$3(TimeLine, int)
void TimeLine.access$4(TimeLine, Interval)
TimeLine.IntervalListener TimeLine.access$5(TimeLine)
void TimeLine.bringToFront(Interval)
LineSegment TimeLine.calculateView(Interval)
void TimeLine.deselect()
int TimeLine.findGroup(int)
Interval TimeLine.findInterval(int)
Dimension TimeLine.getPreferredSize()
void TimeLine.paintComponent(Graphics)
void TimeLine.paintOne(Graphics, Interval)
void TimeLine.paintSelectedIntervalInGroup(Interval, int)
void TimeLine.selectByX(int)
void TimeLine.setIntervalLimits(Interval)
Zoom:
6.31x
Pan:
Center: 7161

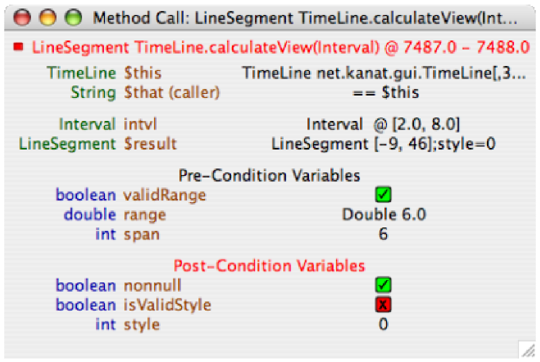
Figure 3. Visualizer displays all method calls traced in one session, highlighting failures with red blocks and cross marks.



Trace Details for Trace All @ 12:15:47 PM EST – 12:16:58 PM EST
Interval TimeLine.access$0(TimeLine)
void TimeLine.access$1(TimeLine)
void TimeLine.access$3(TimeLine, int)
void TimeLine.access$4(TimeLine, Interval)
TimeLine.IntervalListener TimeLine.access$5(TimeLine)
void TimeLine.bringToFront(Interval)
LineSegment TimeLine.calculateView(Interval)
void TimeLine.deselect()
int TimeLine.findGroup(int)
Interval TimeLine.findInterval(int)
Dimension TimeLine.getPreferredSize()
void TimeLine.paintComponent(Graphics)
void TimeLine.paintOne(Graphics, Interval)
void TimeLine.paintSelectedIntervalInGroup(Interval, int)
void TimeLine.selectByX(int)
void TimeLine.setIntervalLimits(Interval)
Zoom:
535x
Pan:
Center: 7551

Figure 4. Individual passed and failed calls. This is a single thread (Java Swing event thread) and the method calls (from paintComponent to paintOne to calculateView) are obvious in this view.

The visualizer allows the tester to zoom in to these problem methods and calls to determine the reason for the failure of these assertions. The tester can examine the stored values of all the observables for the individual passed and failed method calls, inspect the details of any observable object (fig. 5), including private fields.



```
Method Call: LineSegment TimeLine.calculateView(Int...
LineSegment TimeLine.calculateView(Interval) @ 7487.0 – 7488.0
      TimeLine $this          TimeLine net.kanat.gui.TimeLine[,3...
        String $that (caller)              == $this

      Interval intvl                 Interval  @ [2.0, 8.0]
   LineSegment $result           LineSegment [-9, 46];style=0

                     Pre-Condition Variables
       boolean validRange                     ✓
        double range                      Double 6.0
           int span                           6

                     Post-Condition Variables
       boolean nonnull                        ✓
       boolean isValidStyle                   ✗
           int style                          0
```

Figure 5. Details of a failed method call, with failures highlighted in red. Each object shown ("$this", "$that", "intvl" and the return value in "$result") can be further analyzed in a separate window by clicking on the object's string representation.

2.4. Statistical Analysis and Plots

The aggregate data gathered for all calls of one method are also statistically analyzed: A graphical two-dimensional table showing statistical covariances between all observables highlights strong relationships.

The tester may click on one cell in the covariance matrix to pop up a plot of one observable against another. The patterns of correlation can then be further inspected. For unexpected data points, the method call that supplied the data point can be inspected in detail. Any hypotheses about how the program currently behaves or should behave can be converted to contracts that define the expected relationship between the observables plotted.

2.5. Goals: Debugging, Understanding, and Testing

Starting from the goal of finding the cause for a failure, the tester can convert the negation of the observed failure behavior to an assertion. The visualizer will then highlight the method calls that fail when the failure is manifested, and by going backwards in time (and possibly by adding more assertions for the earlier methods and re-tracing) the cause of the failure can be discovered.

If the SUT is too large or unfamiliar, the tester may temporarily switch to the goal of understanding the system and declare more observables in order to discern the patterns in the big picture formed by the visualizer, correlation matrix, and plots.

As DBC can be used as an automated testing oracle, contracts can also be utilized to test parts of a system within the framework of the integrated system. This would exercise the input state space of each method according to the common use of that method. For more thorough testing of a method in isolation, the interactively discovered contracts of that method can be exported.

3. EXPERIMENTS

3.1. Performance Degradation

For collecting and visualizing moderate to large amounts of data, our implementation does not cause a significant degradation in the SUT performance. Using a Mac Mini with G4 1.25 GHz PPC CPU and 512 MB memory, for a system trace of 11728 method calls collected in 20.8 seconds, we observed a 19.2% performance degradation. At five times this rate, 55385 method calls could be recorded in 19.9 seconds with 35.5% performance degradation. This is not a particularly fast system, and yet the SUT remained quite usable. For comparison, moving the mouse to traverse the diagonal span of the screen in two seconds causes about 30% performance degradation. The resulting trace mural was quite responsive in the first case of about 12000 method calls, but a little lagging in the second case with about 55000 calls. We use AspectJ for interception and reflection-based interpretation of Beanshell for contract evaluation. In a commercial system, to decrease the overhead, JVMPI interface hooks and incremental compilation would be preferable.

3.2. Future Experiments

We are soliciting human subjects (software engineers and students) for randomized controlled experiments to test the utility of our approach. Subjects will be given a short hands-on introduction to using our visualizer. Each subject will be asked to discover and fix two bugs in one program using Eclipse, and two other bugs in another program using our visualizer. To eliminate any dependency on individual variations of ability, the total time taken for all four bugs will be used to normalize the time each subject took for each bug. Also, each subject will be asked to complete a very short questionnaire about comparative ratings of these two approaches, and which features of the visualizer were most useful / hard to use.

3.3. A Personal Observation

The first author has been using this system to discover and patch (temporarily fix) bugs in his own system for a while. We report on one interesting session here.

In one portion of the visualizer, a method traverses directory structures and tries to find Java source code automatically, using a simple heuristic. Using the class path as the starting point, this algorithm first searches for an x.java file in the directory where the corresponding x.class file resides. If this fails, parent and sibling directories, and farther directories are traversed, with each next traversal covering an exponentially larger area.

There was a bug in this method which was not obvious upon inspection of the source code.

Using AOP, it was very easy to intercept all the related methods in the enclosing class. Setup was a snap, requiring only two lines of AOP code to be modified, to change the definition of a pointcut, and to call the faulty method from the tracer. The visualization and ability to zoom in to method call details helped discover the source of the bug in about half an hour. The bug was then corrected, and the program ran correctly.

But there was something seriously out of place. The visualizer showed that a very large number of method calls were taking place (fig. 6). The stack of directories to traverse was being scanned in reverse order (due to stack/queue structure usage): The farthest relative (third degree parent and all its descendents) was searched first. The method still found the source file, but after much unnecessary wasted effort. Fixing this bug was also simple once the problem was discovered.

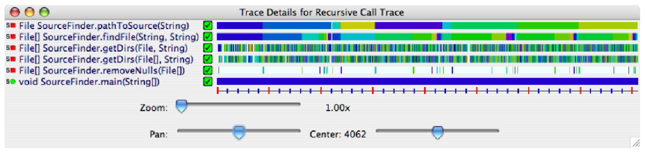

Trace Details for Recursive Call Trace
File SourceFinder.pathToSource(String)
File[] SourceFinder.findFile(String, String)
File[] SourceFinder.getDirs(File, String)
File[] SourceFinder.getDirs(File[], String)
File[] SourceFinder.removeNulls(File[])
void SourceFinder.main(String[])
Zoom:
1.00x
Pan:
Center: 4062

Figure 6. File search algorithm gave correct result but there was much unnecessary disk access and computation.

The visualizer highlighted a problem of inefficiency, even after the program was correct (had no bug), and despite the fact that no such problem was sought after. Debuggers allow precise, narrow, deep analysis and full control, but they do not collect information from multiple calls of the same method. Because of this, no debugger today can make such large scale patterns of behavior obvious.

4. CONCLUSION

We have proposed a new methodology that integrates visual and symbolic approaches to program understanding, testing, and debugging. Our experience shows that an execution trace visualizer can make high-level patterns of faulty behavior obvious. Using our approach, we can run tests without restarting the SUT, access a wealth of information about its internal state, and use code to define observables and DBC contracts to specify the expected behavior of the SUT. We believe this approach makes the debugging process more efficient as compared to a state-of-the-art debugger (Eclipse). We are setting up human-subject experiments to test the validity of this claim.

5. ACKNOWLEDGEMENTS

I would like to thank Marju Purin for improving my grammar as well as my critical thinking through proper word usage.

6. REFERENCES

[1]. B. Lewis, and M. Ducasse, "Using events to debug Java programs backwards in time," Companion of the 18th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) '03, pp. 96-97, 2003.

[2]. B. Boothe, "Efficient Algorithms for Bidirectional Debugging," Proc. of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation, vol. 35, no. 5, pp. 299 - 310, 2000.

[3]. D. Coppit, J. M. Haddox-Schatz, "On the Use of Specification-Based Assertions as Test Oracles," 29th Annual IEEE/NASA Software Engineering Workshop, pp. 305-314, 2005.

[4]. M. Müller, R. Typke, and O. Hagner, "Two controlled experiments concerning the usefulness of assertions as a means for programming," Proc. of the Intl Conf on Software Maintenance, pp 84 - 92, 2002.

[5]. J. A. Jones, A. Orso, and M. J. Harrold, "GAMMATELLA: visualizing program-execution data for deployed software," Information Visualization, vol. 3, no. 3, pp 173-188, 2004.

[6]. D. Jerding, J. Stasko, "The Information Mural: A Technique for Displaying and Navigating Large Information Spaces," Proc. IEEE Symposium on Information Visualization, pp. 43 - 50, 1995.

[7]. Aspectj Project, http://www.eclipse.org/aspectj/