VERDICTS : Visual Exploratory Requirements Discovery and Injection for Comprehension and Testing of Software

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

Abstract *- We introduce a methodology and its prototype implementation for visual exploratory software analysis. VERDICTS combines exploratory testing, tracing, visualization, dynamic discovery and injection of requirements specifications into a live quick-feedback cycle, without recompilation or restart of the system under test. This supports discovery and verification of software dynamic behavior, software comprehension, testing, and locating the defect origin. Our prototype implementation uses aspect-oriented method call interception, functional specification of requirements using dynamically injected Design by Contract, statistical analysis and various forms of visualization. We report on our personal experience.*

Keywords: software testing, program comprehension, software visualization, design by contract, requirements discovery.

1 Introduction

The internal state space of a typical program is significantly larger and more complex than the state space of its user interface. This often manifests itself in a delay between the event that causes internal state corruption and the consequent defective behavior becoming visible to the user. Due to its process (rather than implementation) orientation and relative simplicity, the user interface is a good place to start understanding the conceptual structure and dynamic behavior of a program.

The ideal tool for locating the origin of a defect would add no overhead in execution time or resources, allow normal interaction with the software system under test, and help the analyst discover the origin(s) of the defect starting with outwardly manifested defective behavior, testing various hypotheses, going backwards in time through causal links (by execution tree and shared resources), comparing what is observed against what is expected of the software.

Going backwards in time and having access to the full internal state of the program requires significant overhead in memory and disk space as every instruction and every event has to be traced and logged (see [Boothe 2000, Lewis 2003, Ko 2008], and section 4.2). On a single-CPU system, it is not possible to observe and track the internal state of a program without changing its timing, and this can significantly change the occurrence of defects that are due to race conditions.

There are a number of other innovative approaches to debugging (Algorithmic Debugging [Shapiro 1982], Delta Debugging [Zeller 2002], Query-Based Debugging [Lencevicius 1997, Lencevicius 1999]) that we borrow ideas and techniques from, and our approach is aligned with Exploratory Testing [Kaner 2004] and hypothesis testing that is central to all the software comprehension models surveyed in [Mayrhauser 1995].

In section 4 we review related work in some detail.

2 VERDICTS

VERDICTS (Visual Exploratory Requirements Discovery and Injection for Comprehension and Testing of Software) is a live, visual, and automated approach to software comprehension, testing and validation.

In a single run, new requirements can repeatedly be discovered, captured with an injected contract, cross-verified (checked against the behavior of the program), and modified without recompilation or restarting the software under test.

Visualization is used to aid comprehension by providing abundant information in a form that makes it relatively easy to detect common patterns. Executable requirement specifications (contract assertions) are used to automate erroneous behavior detection during normal operation.

We implemented our prototype VERDICTS system in Java using AspectJ method call interceptions and Beanshell Java interpretation to add dynamic Design by Contract (DBC) assertions as well as other variables ("observables") to the system under test. We use programmatic DBC to specify requirements using method precondition and postcondition assertions. Our high-level visualization is a simple chronological execution trace of methods of interest. This is enriched by highlighting failed contracts and a number of other types of visualizations (see section 2.2).

2.1 Contributions of Our Approach

Quick feedback: The target program can be retested with multiple contracts without the need for recompilation. As we focus on method calls, our approach is not as CPU-intensive as instruction-level tracing and logging approaches mentioned above; VERDICTS prototype is quite usable interactively (see section 4.2).

Visualization of aggregate behavior: A simple execution trace visualization can be used to detect patterns of method calls over time. We also have various visualizations for general functional/behavioral patterns of one method across multiple calls, through statistics and X-Y plots of parameters, user-defined variables and contract assertions.

Table 1. Types of contracts for comprehension and testing

Contract Type	Predicates	Example for sortAscending(int[] ar)	Failure Marks
functional requirements	R = R1 ∧ R2 ∧ R3 ∧ . . . (note: there may be multiple decompositions of R, with different Ri)	R1: array ar is sorted afterwards : i in 1..n-1 : ar[i-1] ≤ ar[i], n = |ar| R2: ar contains the same elements (permuted; no element is lost)	faulty implementation or faulty requirement
actual implementation	P = P1 ∧ P2 ∧ P3 ∧ . . . (similarly, Pj may vary without changing P)	can be very specific (Pi: one test case), very general (P1: ar is sorted after call), or inbetween (P1: |ar| < 4 ⇒ ar gets sorted)	wrong comprehension (of implementation)
failure mechanism (pro)	Pj (where ∃i : Pj ⇒ ¬Ri)	ar[0] = 0 after call (bug; contradicts R1, R2)	wrong comprehension
failure mechanism (con); a partial requirement	Ri (where ∃j : Pj ⇒ ¬Ri)	ar[0] ≤ ar[1] (hypothesis: ar[0] is not sorted, so this contract will sometimes fail)	correct comprehension, and a case of failure
general pattern ; fuzzy	S (where S "often" holds)	ar[0] > 0 (observed usage pattern)	exception to the pattern

Hypothetical contracts: We can explore requirements and behavior through contract assertions that do not all come from requirements, as shown in table 1. In this table, predicates Ri come from requirements, and Pj come from the program (actual implementation behavior). R states what is required whereas P states how the program actualy runs. A defect causes a program behavior to contradict a requirement; ∃i,j : Pj ⇒ ¬Ri (equivalently, ∃i,j : Ri ⇒ ¬Pj).

In table 1, comprehension refers to how well we understand the current, possibly faulty, implementation. Understanding a faulty implementation also entails understanding its mechanism of failure.

2.2 Features of VERDICTS

Exploration: VERDICTS users need not consider requirements to be perfectly accurate, complete and fully satisfied by the software. Similar to Exploratory Testing [Kaner 2004], our approach attempts to challenge the tester to use critical thinking and experimentation with immediate feedback rather than follow a scripted set of steps.

Observables: Method call parameters, return value and this object (in non-static methods) are the primary observables (traced variables). Any other primitive value or object that is reachable during a method call, including private fields, can be declared as observables, with a user-defined variable and a corresponding Java expression. After a tracing session, we can examine the aggregate behavior of the observable (across multiple calls to the method) in isolation as well as in pairs and groups, using statistical analysis and X-Y plots.

Requirements Discovery and Injection: In exploratory testing, the tester's ongoing evolving hypotheses about origins and mechanisms of defects in software are not explicitly recorded. Our approach allows exploration and evolution of hypotheses while requiring explicit, recorded, executable specifications in the form of method contracts.

Software Comprehension and Testing: Both comprehension and testing require comparison (cross-verification) of developer's/tester's mental models of the program against the actual implementation of the program. When there is a mismatch, comprehension presumes mental models inaccurate while testing presumes implementation inaccurate. For most industrial-scale software, a developer may never have a complete mental model of the whole program. In debugging, there is often an exploration that combines comprehension and testing.

Hypotheses:: There are a number of competing models for how program comprehension works, but in most existing models (see section 4.4) a cycle of forming, testing, changing and discarding hypotheses is central to software comprehension (actually, to any learning task). Most hypotheses require some additional source code for testing, and traditionally, recompilation would be required, which would delay feedback.

Other than reducing overall efficiency, such delay is detrimental to the process of promptly discarding faulty hypotheses. Untested faulty hypotheses often are the causal precursors to more faulty hypotheses and even to some faulty convictions. Our approach allows hypotheses to be stated and cross-verified against the program efficiently and without delay, within a matter of minutes, and sometimes, seconds.

3 Sample Cases

In this section we analyze how VERDICTS has helped us discover failure mechanisms behind three actual defects in non-vital components of our VERDICTS prototype.

3.1 Advantage of Visualization: Correct Results, Inefficient Implementation

There are cases where a program runs correctly as per the functional specifications, but runs quite unexpectedly in terms of how it achieves the end result. One case was accidentally discovered in a search method that used simple sets of heuristics to discover the location of Java source code file (X.java) when the location of corresponding compiled bytecode file is known (X.class). The heuristics would allow the search method to start from the closest commonly associated directory locations, and try wider searches up to a selected limit until the source code is found.

In our tests, the method could easily and quickly find the source code. Nevertheless, visualization revealed that the program ran quite differently from what was intended. Instead

of trying the closest few directories first, due to inverted stack, it actually started with the many farthest directory locations, finding the close files only after unnecessary exhaustive search of many unrelated directories. Figure 1 shows the unexpectedly crowded trace visualization where we had expected to see only a few calls to the two getDirs methods.

The operating system (Mac OS X, 10.5) efficiently executed the many calls to search directories and files, and the outputs were correct, even though implementation had a defect. This case demonstrates the power of visualization: It can make problems obvious even where none is being sought, because it can succinctly present much information.

The next two cases, in sections 3.2 and 3.3, are structured as seen in table 2 below.

Table 2. Organization of sections 3.2 and 3.3

Functionality	Functional requirements
Symptoms	Observed failure
Implementation	Overview of classes, methods, calls
Exploration: **Set-up** **Hypotheses** **Tests**	Our set-up, hypotheses and tests: User-defined observables, assertions Our guess for mechanisms of failure VERDICTS tests, what we learned
Conclusions	Discovered mechanism of failure
Solution	Defect origin, corrected implementation
Evaluation	Evaluation of VERDICTS

Note that VERDICTS focuses on method contracts, not individual instructions. After mechanism of failure is discovered, we still need to examine the source code to locate defect origin, the faulty instructions.

3.2 Memory Usage Predictor: Error in Library

Components in a library are rarely specified with formal contracts. Our memory usage predictor class depends on the standard Java library function Runtime.freeMemory(). For Java 1.5 on Linux Ubuntu 8.04, this library function behaved quite unexpectedly; standard Java documentation gave no hints that suggested what we discovered.

Functionality: MemoryUsagePredictor predicts memory usage in loops to allow loops to terminate without running out of memory.

Consider a loop which creates new objects of type C, where loop iteration i creates xi more objects and uses mi more bytes of memory. Our MemoryUsagePredictor uses a linear regression model between xi and mi, so that the loop can be safely terminated if the next iteration may run out of memory. MemoryUsagePredictor.check(xi) must be called and consulted after discovering xi at iteration i, and before creating xi objects of type C. Since this thread may be swapped out mid-iteration, there will be outliers in data (too much or too little compared to predicted range). Such outliers, often caused by other threads, have to be ignored.

Symptoms: There was a bug in MemoryUsagePredictor that caused it to not predict any values or predict unexpectedly wrong values of memory usage.

Implementation: We'll focus on two classes and four of their methods:

MemoryUsagePredictor:
getCurrentMem(): Returns used/max memory ratio
check(xi): Returns true iff creating xi more objects won't exhaust memory. Adds data points to linear regression model and uses the model for prediction.

SimpleLinearRegression:
addPoint(xi, mi): Called by check() to add a sample
getNumSamples(): Returns number of points added

Exploration Using VERDICTS: We started VERDICTS. In our first test, we just collected execution trace data without user-defined observables. Figure 3 shows an execution trace screenshot, with full method signatures. We added thick red lines to show links to individual method call details (these pop up when the linked box is clicked on) and to show the pattern of missing calls. We observed that **addPoint()** almost never gets called, even though **check()** is called many times. We had expected most calls to **check()** to call **addPoint()** as seen in figure 2.

In this case, we check the memory usage many times but only add a single point to our linear regression model. Our sample size remains one, so our MemoryUsagePredictor can't predict any memory usage.

Our hypothesis: *Many calls to check are falsely detected as outliers and ignored.*

We added some contract variables (user-defined "observables") to the **check()** method, to observe the change in memory usage over time. Our observables depend on the methods listed above and one field, **currentMem**, where the **check()** method caches the value returned from the

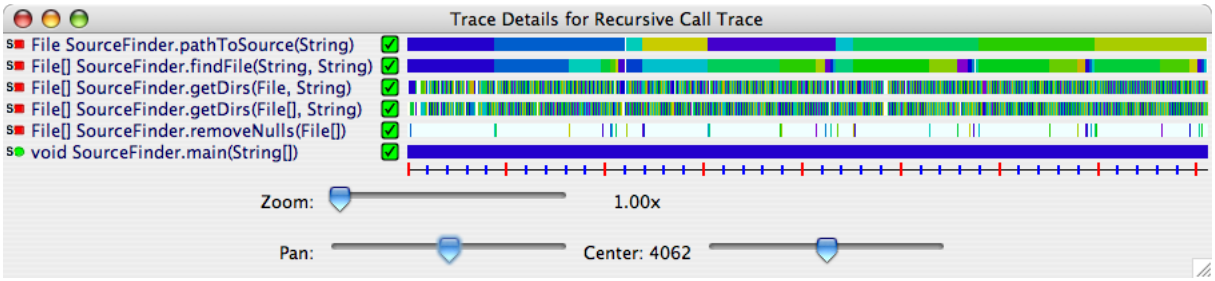

Figure 1. Correct results, incorrect behavior, in execution trace visualization. X axis is time; Y axis lists methods.

getCurrentMem() method (memory usage as a fraction of maximum). Before a call to **check()**, this field holds the old value from previous call to **check()**. Our observables below were defined to be evaluated before the **check()** method call, so they were precondition contract variables (see figure 4):

mem: Holds current memory usage

memDiff: Holds delta difference of memory usage.

lastX: Holds the last xi value (linear regression independent variable; number of elements allocated).

lastPredicted: Holds the last predicted memory usage corresponding to lastX. Prediction is by linear regression.

Using VERDICTS, we defined the contracts and reexamined the defective behavior without recompiling or restarting the program, to discover the picture seen in figure 5 (again, with thick red lines added to show associations with method call details). Three method call instances are displayed on top. Note that **memDiff** was 0 in the second and third calls, which was the reason why **addPoint()** was not called. The same held true for all other missing calls to **addPoint()**.

The error was not due to multithreaded evaluation and outliers. Memory usage must have changed with new objects created; this was not reported back to us (hence memDiff is 0).

Conclusions: The standard Java library call to find current memory usage returns a value that is not updated in real-time. There is a coarse-grain update rather than a fine-grain one. Reported free memory does not change with every memory allocation. This was not implied in any way by the library API documentation, and was completely unexpected.

Solution: We changed our linear regression model usage to allow for delayed feedback by aggregating memory usage and number of elements while memDiff remains 0.

Evaluation: Our approach allowed us to notice an unexpected pattern, zoom in to problematic methods, create general observables and discover problems in a single session where we could inject code, repeat the problematic interaction, and trace the program a second time without ever stopping the program. Note that in a debugger where method-call aggregate behavior and method-to-method call patterns are not obvious, it would be much harder to notice that many calls to **check()** do not cause a call to **addPoint()**, especially when the first few calls do. It took us only minutes to test our hypothesis, which we promptly discovered to be wrong.

3.3 Dual GUI Control of Single Model Variable: Potential For Infinite Loop

Functionality: We allow different visualizations related to thread state and executed method call. All method calls on one thread can be viewed together against time, in a "TimeLine" visualization. In a separate window, thread state can be viewed for a selected time (waiting, sleeping, active, blocked, ...). This time can be selected in one of two ways:

- by clicking on a method call on the TimeLine
- by using a slider on the thread state window

Symptoms: We should be able to select time by adjusting the slider, but a bug in the implementation caused the time to never change, regardless of the slider value.

Exploration Using VERDICTS: We added some observables to stateChanged method to compare "sliderValue"

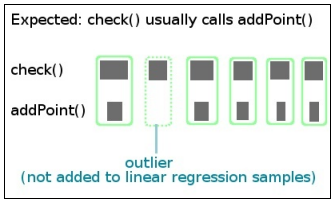

Figure 2. Expected pattern

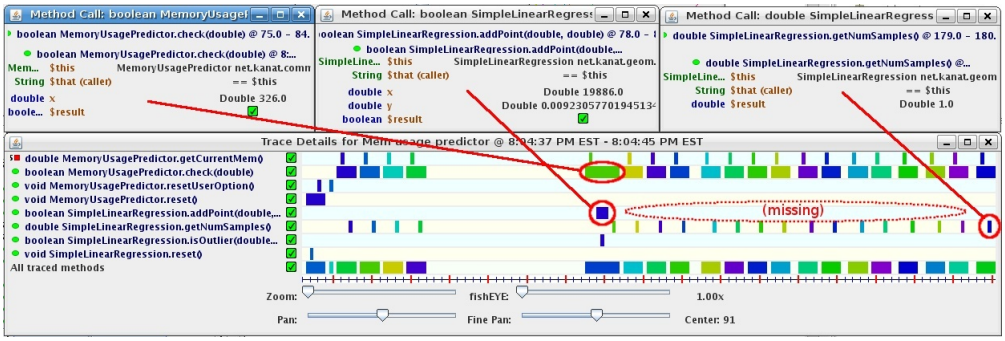

Figure 3. Traced method calls, their parameter values. X axis is time, Y axis is methods

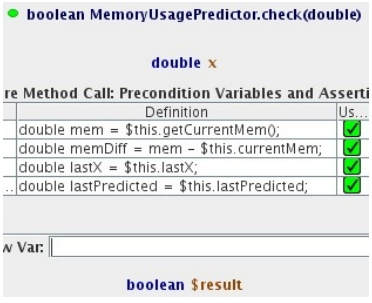

Figure 4. User-defined observables

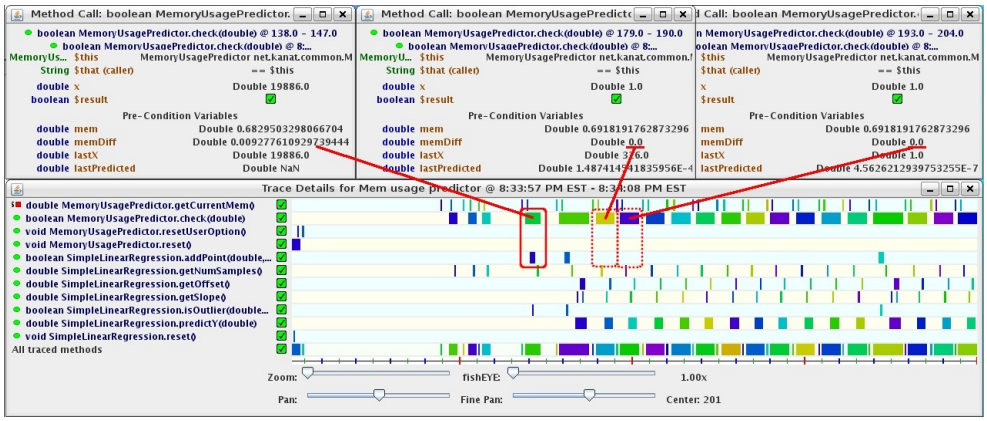

Figure 5. Traced method calls, observable values, unexpected memDiff == 0.0 cases

that represents the view state against the "timeValue" which represents the model state:

sliderValue: Expected current slider value

timeValue: Expected current time value

oldSliderValue: Recorded (actual) previous slider value

oldTime: Recorded (actual) previous time value

sliderValueChanged: Assertion: Slider value has changed.

Note that these four "precondition" variables and one assertion are evaluated before the method call, so the recorded slider and time values hold the previous values.

The user interface does not react to slider adjustment, and time appears to never change. Our contract for expected behavior states the opposite; sliderValueChanged asserts that slider value changes with each call to "stateChanged" method.

Our Hypothesis: *Slider GUI ought to change the sliderValue. As a general (but not universal) rule, sliderValueChanged ought to pass most of the time during correct operation. But we suspect that the slider GUI is disconnected and the back-end does not receive its value. So we expect sliderValueChanged assertion to always fail.*

After running the program with these observables and contractual assertion, we viewed statistics (figure 6) and plots of these observables. Here, we highlighted two unexpected patterns using a green box (expected relationship), a red box and underline (unexpected relationship and statistics). We would expect a similar correlation between the expected value pairs (sliderValue, timeValue) and the actual recorded value pairs (oldSliderValue, oldTime), but instead we see oldTime to be constant (std. dev. = 0; min = max = 130.6) and completely uncorrelated with oldSliderValue (or any other variable).

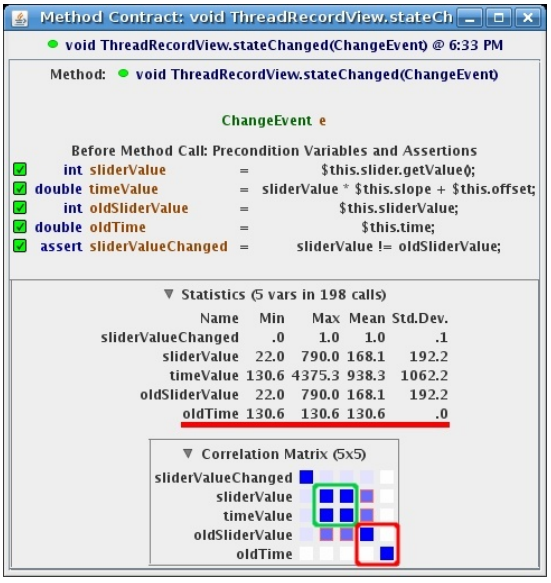

Figure 6: Statistics for our observables (contract variables) across many calls to the method. Unexpected pattern is marked in red, expected pattern, in green.

Clicking on the boxes in the correlation matrix, we can see X-Y plots of these relationships. Figure 7 shows two of these plots, for expected and actual value pairs mentioned above. In this figure, the large-font legends and “expected”/“actual” boxes are added to these plots to make the window title and observed pattern more visible and obvious.

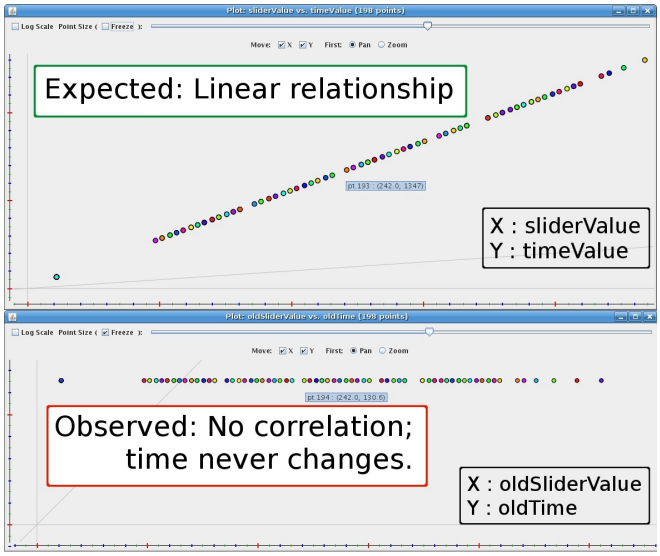

Figure 7: Slider value vs. time value plots showing expected and actual recorded relationship.

Next, let's take a look at the method calls over time. In figure 8, the red boxes represent failed contract. Surprisingly, we see that the slider value almost always changes, in accordance with our assertion, and only fails to change in the first and the last calls to the stateChanged() method.

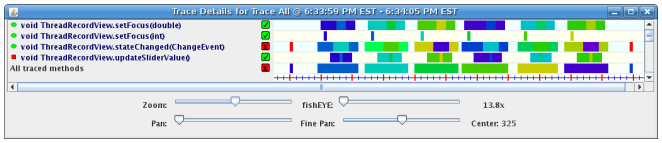

Figure 8: Traced method calls, showing recursion and surprisingly few failed calls as per our contract – our hypothesis about mechanism of failure was wrong.

More significantly, we see patterns of recursion. If method calls X and Y are in the same thread and X is subsumed by Y, then X is directly or indirectly called by Y. We see that three methods, “setFocus(double), stateChanged(...) and updateSliderValue() are mutually recursive. Specifically, going from largest to smallest method call, we have:

stateChanged(...) → setFocus(double) → updateSliderValue()
→ stateChanged(...) → setFocus(double) → updateSliderValue()

We were lucky to not have an infinite recursion; it appears that we came quite close to it. Both stateChanged and setFocus went beyond depth 2 in mutual recursion, but updateSliderValue, for some yet unknown reason, decided to terminate mutual recursion at depth 2.

In figure 9, either the old value or the new value is always 22; (x, 22) and (22, y) are the only choices. Together with depth 2 mutual recursion and observed behavior of time never appearing to change, it now appears that we repeatedly set the slider value to what the user desires (the (22, y) case), and

then, in a recursive call, reset it back to its old value (the (x, 22) case), always coming back to the initial and minimum value of the slider, 22.

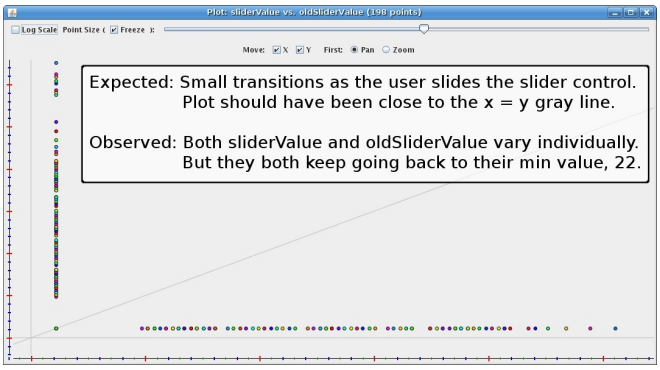

Figure 9. Changes in slider value show a strange pattern.

Conclusions: Upon focused source code examination, the origin of the problem first appears to be an inverted condition for returning false from a helper method, setTimeIfNeeded(), which should be called from setFocus(int).

But this is not the actual bug. As can be seen (by its absence) in the trace visualization in figure 7, setTimeIfNeeded() itself is never called. The only condition that could cause this was "entryComp" field being null. In fact, entryComp remained uninitialized, null, because in the method that should have initialized entryComp, the field was shadowed by a same-named local variable entryComp. This defect was caused by faulty manual refactoring.

Solution: After fixing these two relatively simple bugs, the program ran correctly.

Evaluation: In this case, we also added an assertion, a contract for expected behavior. This was a contract for a requirement that ought to be satisfied "most of the time" rather than at all times. Specifically, it stated that the internal state should change with each call to one method. Our hypothesis about mechanism of failure suggested that this assertion will actually fail at all times. To our surprise, with few exceptions, the assertion was satisfied. Our hypothesis about mechanism of failure was wrong. Such quick negative feedback is extremely useful as it allows us to stop expanding on deductions based on faulty assumptions about mechanisms of failure.

4 Related Work

4.1 Software Testing

Testing is often conceived as starting with a known set of requirements, going to a known set of tests to be conducted, which are then applied. This does not provide for the opportunity to learn from results of tests to modify, evolve, and discard a test or prioritize the set of tests. One exception is exploratory testing [Kaner 2004]. Our approach works well with the exploratory (context-driven) testing paradigm, and allows it to be dynamic and automated (through injected specifications), with the additional aid of visualizations.

4.2 Instruction Tracing: Reversible / Omniscient Debugging, and Whyline for Java

During debugging, developer is often lost in space (source code) and time (execution), and is taking tiny steps for fear of losing program internal state. With reversible debugging, the developer can debug the program in reverse. Two main approaches are:

Post-mortem analysis: Much event data is collected and can be efficiently searched and queried, jumping to any point in execution forward or backward. Variable values, state and program behavior cannot be changed. Examples: Omniscient debugging [Lewis 2003], Whyline for Java [Ko 2008]. Allows bidirectional debugging after execution completes.

Live bidirectional execution: Execution can move forward or backward, variable values can be changed, and forward execution paths will change to reflect this change. [Boothe 2000].

These approaches are costly. Full state information must be saved. Usually, this is done at regular intervals and is called checkpointing. Checkpointing takes a very significant amount of disk space, and takes time, and may not always be feasible, due to external side effects of the program. For example, a program can delete a file with a simple OS call, but there may be no easy way to undelete such a file to recover its contents. In a distributed program that uses the network, the whole distributed state, possibly spanning many machines and OSes, would have to be checkpointed and reverted to.

Also, the tester remains steeped in the details of execution in live bidirectional debugging, and can not change program state at all in post-mortem analysis.

Whyline for Java [Ko 2008] is a recent post-mortem causal-link analysis tool. Whyline follows causal links from user interface to implementation by tracing events such as field value change. At execution time t, the question "Why is field xyz 0?" can be answered with the source code that caused the last change to field xyz before time t.

These approaches are powerful, but may not be practical due to their overhead. Unlike instruction-level approaches, the overhead in VERDICTS does not depend on total program size; it depends on number of calls to a preselected set of methods, number of observables and contracts [Bolazar 2006].

Whyline for Java [Ko 2008] reports overhead factors of 4.1 – 14.3 compared to normal program execution. In early tests of our own prototype, we observed an overhead factor of 0.19 - 0.35 (19% - 35%) [Bolazar 2006], which is about 20 to 40 times less than Whyline (this is not a general claim; we had not used the same target programs for our tests). Many interactive programs with real-time behavior (such as a GUI component with task timeout for user interactivity) cannot be traced accurately if the software slows down by a factor of 10.

4.3 Other Innovative Approaches to Debugging

Algorithmic debugging [Shapiro 1982] repeatedly asks the developer questions about whether inputs and outputs of each function appears correct. Starting from top-level function and going down in the function call tree, one branch is implicated

as faulty at each step, until the function with faulty implementation is discovered. Making inputs and outputs of method calls visible helps not only in debugging, but also in comprehension of software dynamic behavior.

Delta Debugging [Zeller 2002] requires software input and control automation to find minimal change in input that causes failure to appear. Our interception mechanisms would work well with delta debugging by making any subcomponent of the program automatically testable.

Software visualization, especially when combined with testing and verification methods can aid in comprehension and fault origin discovery. One example, Tarantula [Jones 2001], uses unit test results overlayed on a view of the source code, and shows one way to relate static structure to dynamic behavior.

Query-based debugging [Lencevicius 1997, Lencevicius 1999] appears very similar to our approach, but requires that the developer already understand the internal state variables responsible for failure. The recommended approach of checking queries whenever any field value changes would not work with most contracts that depend on multiple field values: At some point during the execution, one field value would have been updated without the other one conforming to their contract yet. We instead focus on inter-method communication and state.

4.4 Software Comprehension Models

There are many competing models of software comprehension. Hypothesis/Conjecture-testing is central to all software comprehension models surveyed in [Mayrhauser 1995]. The developer who is trying to understand a program generates hypotheses, tests/checks them out, and learns from the results of these experiments and static code analysis to improve the set of hypotheses by generating new ones, modifying and at times discarding old ones. In the face of contradictory evidence, expert developers discard their own faulty hypothesis more quickly than novice developers. VERDICTS combines contracts with quick visual feedback for efficient testing of hypotheses.

5 Conclusions

5.1 Observations

We collect, statistically analyze, and visually present, aggregate method call information, which aids in program comprehension, together with our high-level execution trace visualization. In our experience, this is significantly better than single-point-in-time single-place-in-code debugging.

Ability to modify method contracts and test them on-the-fly, without having to stop and restart the system under test (SUT) greatly reduces the delay in feedback and helps the developer quickly eliminate wrong paths. This quick feedback cycle also allows the developer to test some ideas without having to commit to these method contracts. As mentioned before, expert programmers are flexible, and are quick to discard faulty hypotheses. We believe our tool is well suited for flexible exploration of the specification and behavior space of a program.

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 arbitrary Java code to define observables and DBC contracts to specify the expected behavior of the SUT.

5.2 Ongoing and Future Work

We are currently testing various other forms of visualizations that we have implemented and integrated in the VERDICTS prototype, and exploring fuzzy (need not always pass) and temporal contracts, and their visualizations. We are also working on an approach to quantify the quality of contracts, so that we may objectively compare VERDICTS to other approaches.

References

Bolazar, S. K., Fawcett, J. W. 2006. Debugging with software visualization and contract discovery. In *Proc. Software Engineering and Data Engineering (SEDE '06)*, 47-50.

Boothe, B. 2000. Efficient algorithms for bidirectional debugging. In *Proc. ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*, 299-310.

Jones, J. A., Harrold, M. J., Stasko, J. 2001. Visualization for fault localization. In *Proc. Workshop on Software Visualization, 23rd International Conference on Software Engineering (ICSE '01)*, 467-477.

Kaner, C., Bach, J., Pettichord, B. 2004. Lessons learned in software testing: A context-driven approach. John Wiley & Sons, New York, NY.

Ko, A. J., Myers, B. A. 2008. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proc. International Conference on Software Engineering (ICSE 2008)*, 301-310.

Lencevicius, R., Hölzle, U., Singh, A. K. 1997. Query-based debugging of object-oriented programs. In *Proc. 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '97)*, 304-317.

Lencevicius, R., Hölzle, U., Singh, A. K. 1999. Dynamic query-based debugging. In *13th European Conference on Object-Oriented Programming (ECOOP'99)*, published as *Lecture Notes in Computer Science 1628*, 135-160.

Lewis, B. 2003. Debugging backwards in time. In *Proc. 5th International Workshop on Automated Debugging (AADEBUG 2003)*, 226-236.

Mayrhauser, A. V., Vans A. M. 1995. Program comprehension during software maintenance and evolution. In *IEEE Computer August 1995*, 44-55.

Shapiro, E. Y. 1982. Algorithmic program debugging. MIT Press, Cambridge, MA.

Zeller, A., Hildebrandt, R. 2002. Simplifying and isolating failure-inducing input. In *IEEE Transactions on Software Engineering* 28, 2, 183-200.