ReCrashJ: a Tool for Capturing and Reproducing Program Crashes in Deployed Applications

Shay Artzi
IBM T.J. Watson
Research Center
artzi@us.ibm.com

Sunghun Kim
University of Hong Kong
Science and Technology
hunkim@cse.ust.hk

Michael D. Ernst
University of Washington
mernst@cs.washington.edu

ABSTRACT

Many programs have latent bugs that cause the program to fail. In order to fix a failing program, is it crucial to be able to reproduce the failure consistently. However, reproducing a failure can be difficult and time-consuming, especially when the failure is discovered by a user in a deployed application.

We present ReCrash, an approach to reproduce failures efficiently, both locally and in deployed applications, without any changes to the host's environment, and with low execution overhead.

During execution, ReCrash efficiently stores part of the state of method arguments. If the program fails, ReCrash uses the stored information to create unit tests that reproduce the failure. This is effective because programs written in object-oriented style rely mostly on near-by state.

This demo presents ReCrashJ, an implementation of ReCrash for Java. We show the ReCrashJ Eclipse plug-in (for developers) and the ReCrashJ command-line modules (for deployed software).

Categories and Subject Descriptors

D2.5 [**Testing and Debugging**]: Monitors and Tracing; F3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Languages, Theory

Keywords

crash, monitoring, tracing, unit test, reproducing

1. Introduction

It is difficult to find and eliminate a software failure, and especially to verify a solution, without the ability to consistently reproduce the failure. This demo presents ReCrash, a technique that simplifies the task of reproducing failures. ReCrash reproduces failures that were discovered in deployed applications.

ReCrash automatically converts a failing program execution into a set of deterministic, self-contained unit tests. Each of the unit tests reproduces the same failure by starting the execution from a different snapshot (checkpoint) of the system taken during the execution.

Reproducing a failure found in a deployed application is hard, even when program inputs, environmental variables, and GUI actions are available. Existing techniques either require changes to the host environment [10], [7], [11] or supply only one snapshot (checkpoint) of the system before the reproducible failure [3, 6, 9, 12]. Our technique requires no changes to the host environment and supplies the developer with multiple snapshots of the system before the failure. Execution from each snapshot leads to the original observed failure. Having multiple snapshots of the system can be useful in finding the underlying bug, as the developer might get a better view of the cause of failure. In addition, some snapshots allow the developer to debug the failure quickly using a shorter execution instead of a longer execution that exposes the same failure.

2. ReCrash

This section briefly describes the ReCrash technique, implementation, optimizations, limitations, and experimental results. More details appear in our ECOOP 2008 paper [1].

2.1 Technique

ReCrash is based on the following characteristics of object oriented programs: bugs are dependent on small parts of the heap, important state is encapsulated nearby, and global information is not used excessively. Thus, in order to reproduce a method's execution it might not be necessary to store the entire transitive state of the methods' arguments on entry.

ReCrash has two phases: monitoring and test case generation. In the monitoring phase, ReCrash maintains a shadow call-stack containing partial state of the arguments to the methods on the original call-stack. The part of the transitive state of the arguments that is not copied into the shadow stack refers to the original objects on the heap. When the program fails (i.e., crashes), ReCrash serializes the shadow stack contents, including all heap objects referred to from the shadow stack.

In the test generation phase, ReCrash generates candidate tests by calling methods from the de-serialized shadow call stack. Each test executes the original method using the de-serialized receiver and arguments (stored at the time of the crash). ReCrash filters all test candidate that do not reproduce the original failure.

ReCrash outputs multiple tests to create a better view of the failure for the developer. For example, tests that call methods from the top of the call stack may not provide enough context to find the error, while tests that call methods from the bottom of the call-stack provide more context, but are less likely to reproduce the original failure and may contain extraneous, distracting details.

2.2 Optimizations

ReCrash's time and space overheads are mostly determined by the cost of storing the state of arguments. We have considered two orthogonal ways to reduce overhead: reducing the depth of copied state for the method's arguments, and monitoring fewer methods. The full paper [1] evaluates the tradeoffs between performance and reproducibility for the different alternatives.

2.2.1 *Depth of Copying Arguments*

This section considers different strategies for copying arguments (including the receiver) at the method entry. Copy strategies differ in the amount of state copied into the shadow stack on method entry. The rest of the argument state refers to the original objects (that might get side-effected).

The different copying strategies, in order of increasing overhead, are: **reference (depth-0)** copy only the reference; **shallow (depth-1)** copy the direct fields of the arguments; **depth-*i*** copy an argument to a specified depth of dereferences; **deep** copy the entire transitive state.

ReCrash has an additional copying option, **used-fields**, applicable to all copying strategies except reference. In the **used-fields** mode ReCrash performs the selected copy strategy on the arguments, and also copies all the used fields of the arguments. A field is used if it is read or written in the method; ReCrash uses a simple static pointer analysis to determine the applicable set of fields. Since the method is likely to depend on that set of fields, copying them increases the chance of reproducing the failure.

ReCrash always uses the reference strategy for immutable parameters (calculated using the technique presented in [2]).

2.2.2 *Monitoring Fewer Methods*

ReCrash need not monitor methods that are unlikely to expose or reveal problems, or that cannot be used in the generated tests. Those include empty methods, non-public methods, and simple methods such as getters and setters.

It would be most efficient to only monitor methods that will fail. However, it is impossible to compute this set of methods in advance. One way of approximating this set is by storing the set of methods that already failed, updating the set each time a new method fails.

This is the underlying idea behind *second chance*, a mode of operating ReCrash that only monitors methods that have contributed to a failure at least once. In second chance mode, ReCrash initially monitors no method calls. If a failure occurs, then on future executions ReCrash enables method argument monitoring for all methods found on the (real) stack back-trace at the time of the failure.

2.3 Usage Modes

ReCrashJ is our implementation of ReCrash for Java. ReCrashJ has two modes of operation: Eclipse plug-in and stand-alone instrumentation. The plug-in allows the developer to execute a ReCrash-enabled program (configurable with all the optimizations), and it automatically generates the set of unit tests if the program fails.

In stand-alone mode the software vendor distributes a ReCrash-enabled program. When the program fails, the instrumented program automatically (or manually) sends the shadow stack to the vendor's ReCrashJ server. The server runs the test generation phase and sends the tests that reproduces the original failure to the developers.

2.4 Experimental Study

Evaluation on real applications indicates that ReCrashJ is effective, scalable and useful. Below is a summary of the experimental results [1].

- ReCrashJ reproduced 11 real failures from the Eclipse Java compiler[1], SVNKit[2], BST [4, 5][3], and JSR308 [8]. In our experiments, ReCrashJ was able to reproduce every failure we tried.
- ReCrashJ's run-time overhead with the **shadow + used-fields** copying strategy was 13%–64%. In second-chance mode the overhead dropped to 0%–1.7%.
- In a small survey, developers confirmed the usefulness of the tests generated by ReCrashJ in debugging failures.

3. Conclusions

In this demo, we show the ReCrash technique, and our implementation for Java, ReCrashJ. We believe that ReCrash can improve programmer productivity by reducing the time it takes to reproduce failures.

ReCrashJ is publicly available at http://pag.csail.mit.edu/reCrash/.

4. References

[1] S. Artzi, S. Kim, and M. D. Ernst. Recrash: Making software failures reproducible by preserving object states. In *ECOOP*, pages 542–565, July 2008.

[2] S. Artzi, J. Quinonez, A. Kieżun, and M. D. Ernst. A formal definition and evaluation of parameter immutability. *ASE*, 16(1):145–192, 2009.

[3] J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *ICSE*, pages 261–270, May 2007.

[4] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, Sep. 2004.

[5] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *ICSE*, pages 422–431, May 2005.

[6] D. A. S. de Oliveira, J. R. Crandall, G. Wassermann, S. F. Wu, Z. Su, and F. T. Chong. ExecRecorder: VM-based full-system replay for attack analysis and system recovery. In *ASID*, pages 66–71, Oct. 2006.

[7] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, pages 211–224, Dec. 2002.

[8] M. D. Ernst. Type Annotations specification (JSR 308). `http://pag.csail.mit.edu/jsr308/`, Sep. 12, 2008.

[9] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *USENIX*, pages 289–300, June 2006.

[10] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *ISCA*, pages 284–295, June 2005.

[11] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX*, pages 29–44, June/July 2004.

[12] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA*, pages 122–135, June 2003.

[1] `http://eclipse.org/jdt`
[2] `http://svnkit.com/`
[3] `http://www.cc.gatech.edu/cnc/index.html`