18th International Workshop on Model Checking of Software (SPIN'11),
Snowbird UT, July 2011.

dBug: Systematic Testing of Unmodified Distributed and Multi-Threaded Systems*

Jiří Šimša, Randy Bryant, Garth Gibson

Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA, USA

1 Introduction

In order to improve quality of an implementation of a distributed and multi-threaded system, software engineers inspect code and run tests. However, the concurrent nature of such systems makes these tasks challenging. For testing, this problem is addressed by *stress testing*, which repeatedly executes a test hoping that eventually all possible outcomes of the test will be encountered.

In this paper we present the dBug tool, which implements an alternative method to stress testing called *systematic testing*. The systematic testing method implemented by dBug controls the order in which certain concurrent function calls occur. By doing so, the method can systematically enumerate possible interleavings of function calls in an execution of a concurrent system.

The dBug tool can be thought of as a light-weight model checker, which uses the implementation of a distributed and multi-threaded system and its test as an implicit description of the state space to be explored. In this state space, the dBug tool performs a reachability analysis checking for a number of safety properties including the absence of 1) deadlocks, 2) conflicting non-reentrant function calls, and 3) system aborts and runtime assertions inserted by the user.

Related Work The idea of systematic testing of concurrent systems has been first explored by VeriSoft [2]. In recent years, the research community has significantly advanced the state of the art of practical dynamic analysis and produced a number of powerful tools for finding faults in concurrent systems. The CHESS tool [5] implements systematic testing for multi-threaded Windows-based programs. The ISP tool [7] implements systematic testing for distributed MPI-based programs. The MaceMC tool [4] implements systematic testing for distributed and multi-threaded programs written in the Mace language. The MoDist tool [10] implements systematic testing using binary instrumentation for distributed and multi-threaded Windows-based programs. Compared to these tools, dBug is the only tool capable systematic testing of unmodified distributed and multi-threaded systems. As such, dBug can be used for analysis of the plethora of existing concurrent systems designed for POSIX-compliant operating systems.

* The work in this paper is based on research supported in part by the DoE, under award number DE-FC02-06ER25767 (PDSI), by the NSF under grant CCF-1019104, and by the MSR-CMU Center for Computational Thinking. We also thank the members and companies of the PDL Consortium (including APC, DataDomain, EMC, Facebook, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, NEC, NetApp, Oracle, Seagate, Sun, Symantec, and VMware) for their interest, insights, feedback, and support.

2 Tool

2.1 Architecture

Interposition Layer The interposition layer is responsible for monitoring and controlling the execution of the system under test. Notably, when a thread of the system under test is about to call a function which dBug seeks to control, the interposition layer transparently intercepts this call, suspends the execution of the calling thread, and informs the arbiter about this event. At some later point, the interposition layer receives a message from the arbiter which causes the calling thread to resume execution.

Arbiter The arbiter is responsible for selecting the order in which concurrently intercepted function calls execute. The arbiter is a centralized entity that maintains a global view of the system under test. In particular, the arbiter uses a client-server architecture (depicted in Figure 1) to collect information about the threads running in the system and intercepted calls via instances of the interposition layer. At some point, referred to as the *decision point*, the arbiter infers that the system under test can make no progress without the arbiter resuming execution of some of the suspended threads. For example, when the number of threads in the system under test matches the number of concurrently intercepted calls, the arbiter knows it has reached a decision point. At that point the arbiter decides which of the concurrently intercepted calls should execute first.

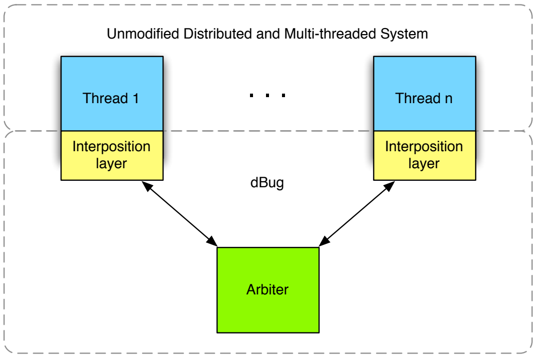

Fig. 1. dBug Architecture

Explorer The explorer is responsible for navigating the repeated execution of the test to yet unexplored parts of the state space. In particular, after each execution of a test, the explorer collects information about all the decision points encountered by the arbiter. This information is then used to gradually build a *decision tree* with nodes corresponding to decision points and edges corresponding to intercepted calls. The decision tree is in turn used to craft schedules for future instances of the arbiter to steer the execution down unexplored branches of the tree. Figure 2 illustrates how the explorer communicates with different instances of the system under test, while iteratively building the decision tree. To explore the decision tree, the explorer uses a stateless approach.

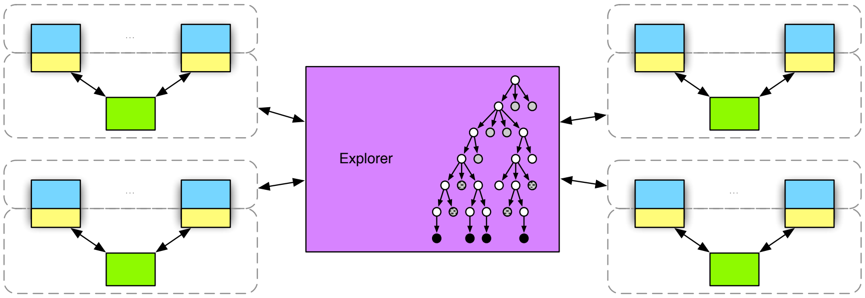

Fig. 2. dBug Exploration

2.2 Functionality

Using the architecture described above, the tool performs a reachability analysis over the decision tree, while checking for a number of safety properties including the absence of 1) deadlocks, 2) conflicting non-reentrant function calls, and 3) system aborts and runtime assertions inserted by the user.

To check for deadlocks, the interposition layer intercepts function calls that are potentially blocking while collecting information that could influence whether a call blocks. Using this information, the arbiter can infer for each intercepted call whether execution of the call from the current state would block. For example, the arbiter keeps track of all threads and processes running as part of the system under test in order to infer if an intercepted call to `waitpid()`, `sem_wait()` or `pthread_join()` would block. A deadlock is detected when the arbiter reaches a decision point knowing all intercepted calls would block.

To check for non-reentrant function call conflicts, the interposition layer intercepts calls to functions that are not guaranteed to be reentrant (e.g. by POSIX standard [3]). The dBug tool checks if a decision point can be reached such that two intercepted calls correspond to the same non-reentrant function being called by two threads of the same process.

To check for assertion violations as well as more general system aborts such as segmentation faults, the interposition layer installs its own signal handlers. These handlers inform the arbiter about a delivery of a signal before the system actually crashes and then forward the signal to the original handler.

The current version of the dBug tool supports interposition of the POSIX process management, POSIX threads, and POSIX semaphores interface. For a detailed list of events intercepted by the interposition layer, we refer the interested read to the dBug User Manual available from the project web page [8].

2.3 Limitations

First, the tool might fail to discover a concurrency error that requires an interleaving of events at a finer level of abstraction than that of a function call.

Second, given that the problem of precise detection of a decision point (i.e. no forward progress is possible without resuming execution of one of the suspended threads) is undecidable, the tool sometimes resorts to using a timeout as a mechanism for detecting a decision point. If the arbiter incorrectly infers it has reached a decision point, the method might fail to cover the decision tree.

Third, even if decision points are detected precisely, if the system under test contains a source of non-deterministic behavior which is not controlled by the arbiter, the method might still fail to cover the decision tree. For example, the overhead imposed by dBug can skew the timing of events in the system, which can be a problem if control flow is time-dependent.

Lastly, the method assumes that a test execution terminates. Although it is possible to execute the method using dove-tailing to avoid getting stuck in an infinite execution, the method does not check for infinite loops.

3 Evaluation

In our previous work [9], dBug has been applied on a parallel file system and distributed key-value storage. More recently, the dBug tool has been used for systematic testing of implementations of a proxy web server created by students as their final assignment for *15-213 Introduction to Computer Systems*, a class offered by the Computer Science Department at Carnegie Mellon University.

The assignment asked students to create a proxy server that receives an HTTP request from a client, inspects the request header to find out the target server, sends the request to the server, receives a reply from the server, and sends the reply to the client. Additionally, the proxy was expected to service multiple requests concurrently and to service duplicate requests from a local cache.

To check student proxies, we would start up a local web server and the proxy and issue requests to the proxy using the `curl` utility. After eliminating 35 proxies that failed simple sequential tests, we were left with 80 proxies. To test the concurrent nature of these proxies, we would concurrently issue two identical requests and used dBug to systematically enumerate possible ways in which the concurrent events triggered by the test could occur. The dBug tool was configured to timeout after 60 minutes. The detailed experiment data for the results below are provided at the dBug project web page [8].

The dBug tool found concurrency faults in 25 of the 80 tested proxies, including a fault in the reference proxy written by one of the course teaching assistants. We confirmed the existence of all of these faults by mapping the traces generated by dBug into the source code of the proxies.

For 13 of the 25 faulty proxies, more than 60% of the executions witnessed by dBug were erroneous. We suspect that the reason why these faults went undetected by the programmers is because dBug, similar to Eraser [6], marks executions that could have gone wrong, but did not, as faults. For example, dBug flags an execution when two threads of the same process make concurrent calls to a non-reentrant function or when a free mutex is unlocked, irrespective of whether such behavior maps to an observable fault.

Lastly, we compared the findings of dBug to the code inspection carried out during grading of the assignment by experienced course instructors and teaching assistants. Surprisingly, the code inspection identified only 20% of the faults identified by dBug. On the other hand, unlike dBug, the code inspection was able to identify multiple data races that happened because of insufficient synchronization of concurrent memory accesses.

4 Conclusion

In this paper we have presented a novel tool, dBug, which addresses the challenge of finding concurrency faults in unmodified distributed and multi-threaded systems. The tool is an open-source project and is available for download from the dBug project web page [8]. As an on-going work we are working on extending functionality and improving efficiency of the tool. In particular, we are adding support for 1) interposition of events of the Linux sockets interface, 2) virtualization and control of time, and 3) state space reduction akin to [1].

Finally, while the experiment has been carried out independently of the 15-213 class, we are working towards making dBug an integral tool available to the education of programmers of concurrent systems.

References

1. Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.*, 40(1):110–121, 2005.
2. Patrice Godefroid. VeriSoft: A Tool for the Automatic Analysis of Concurrent Reactive Software. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 476–479, London, UK, 1997. Springer-Verlag.
3. Austin Joint Working Group. IEEE Standard for Information Technology- Portable Operating System Interface (POSIX) Base Specifications, Issue 7. *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)*, pages 1–3826, December 2008.
4. Charles Edwin Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI '07: Proceedings of the 5th Conference on USENIX Symposium on Networked Systems Design and Implementation*, 2007.
5. Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI '08: Proceedings of the 8th Conference on USENIX Symposium on Operating Systems Design and Implementation*, pages 267–280, 2008.
6. Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.
7. Sarvani S. Vakkalanka, Subodh Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. ISP: a tool for model checking MPI programs. In *Proceedings of PPoPP '08*, pages 285–286, New York, NY, USA, 2008. ACM.
8. Jiří Šimša. dBug Project. `http://www.cs.cmu.edu/~jsimsa/dbug`.
9. Jiří Šimša, Garth Gibson, and Randy Bryant. dBug: Systematic Evaluation of Distributed Systems. In *SSV '10: Proceedings of 5th International Workshop on System Software Verification*, 2010.
10. Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MoDist: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI '09: Proceedings of the Sixth Symposium on Networked Systems Design and Implementation*, pages 213–228, April 2009.

Part 2 - Tool Presentation

The oral presentation of the tool will start with a short tutorial on how to use the tool. In this demonstration it will be shown how dBug can step through an execution of an unmodified distributed and multi-threaded system. In particular, the tool can be run in an interactive mode, which allows the user to input the choices for the arbiter. The batched mode of the tool will be then demonstrated in order to give the audience an idea about the rate at which dBug explores the state space. Finally, if provided with ample time and interest from the audience, we could demonstrate the steps required to extended dBug with support for a new library interface. Notably, adding initial support for reordering calls to a new function requires minutes of effort, which makes dBug easily extensible and we would love others to use the tool for their own experiments and analysis.