LCT-D: Proof Guided Tests for C Programs on LLVM

Olli Saarikivi and Keijo Heljanko

Department of Computer Science and Engineering,
Aalto University, School of Science,
PO Box 15400, FI-00076 Aalto, Finland
{olli.saarikivi, keijo.heljanko}@aalto.fi

Software defects can be very expensive, especially when encountered in economically critical or safety critical systems. Many of these defects can be avoided if it can be ensured that a program meets its specification. When the specification is given formally, for example with assertions embedded in the source code, automated software verification methods can be applied to determine whether a program complies to its specification.

Recently there has been much interest in combining underapproximation and overapproximation based approaches to software verification. Such a technique is employed in the DASH algorithm by Beckman et al. [1], which combines dynamic symbolic execution (DSE) [3] with counterexample guided abstraction refinement (CEGAR) [2]. DASH attempts to generate tests based on counterexamples found in the abstraction. When test generation fails the abstraction is refined to remove the counterexample. The tests can be seen as an underapproximation of the reachable states of the program under test, which DASH tries to expand to include an error. The abstraction on the other hand is an overapproximation which, if error free, also proves the program under test to be so.

The flowchart in Figure 1 provides a high-level overview of the DASH algorithm. DASH implements a modified CEGAR loop, where instead of directly checking whether a counterexample is spurious, DSE is used to generate a test that follows the path to the error in the abstraction at least one step more than in previously executed tests. When test generation fails abstraction refinement is performed to eliminate the path from the abstraction.

To explain the algorithm better we will apply DASH to the program in Figure 2. The example program takes an input, which is marked by the call to `input()`. We wish to verify that no matter what this input value is the program can not execute the error statement on line 4.

At startup DASH creates the initial abstraction from the program's control flow graph (CFG): each program location corresponds to one node in the graph, called a region, and there is a directed edge between two regions if control could flow from the first region to the second. Now each region represents all states of the program at that program location. DASH also runs one initial test on the program with a random input. Let us say this input is 41, in which case

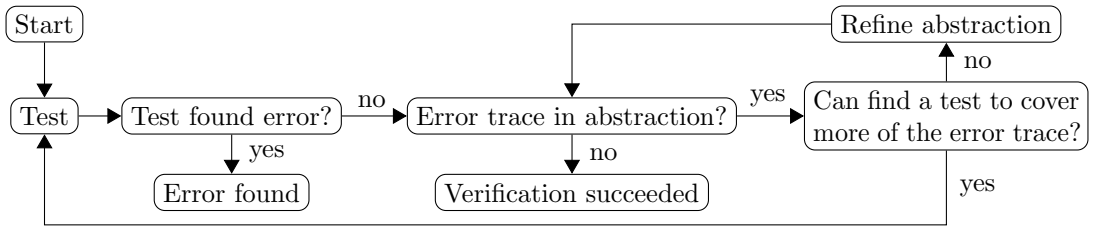
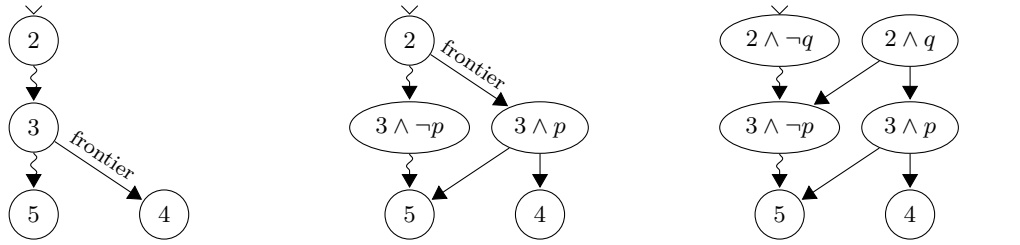


Figure 1: Flowchart for the DASH algorithm

```
int main() {                                    1
    int x = input() * 2;                        2
    if (x % 2 != 0)                             3
        error();                                4
    return 0;                                   5
}                                               6
```

Figure 2: Example program to verify with DASH



(a) After initial test run (b) After first split (c) After splitting the final frontier

Figure 3: Stages of the abstraction for the example program in Figure 2

the program does not enter the body of the `if` statement. The initial abstraction can be seen in Figure 3(a), where the path of the initial test is marked by the wavy lines.

The first iteration of the algorithm starts by finding an abstract error trace $(2, 3, 4)$. A central concept in DASH is the *frontier*, which is the edge along an abstract error trace where the first region has been visited by a test and the second one has not. Here the frontier is $(3, 4)$. DASH will now attempt to extend the frontier by using the constraints gathered from the initial test (as is done in DSE) to generate a test that would execute up to the frontier and across it. In this example the constraint would be $(x = (input * 2)) \wedge ((x \bmod 2) \neq 0)$, which is unsatisfiable. Because a test can not be generated the abstraction is refined to remove the abstract error trace. The refinement is done by splitting region 3 with a predicate p, which is such that when p is false then the execution would never proceed to region 4 from any state in region 3. In DASH such a predicate is produced by a technique similar to how weakest preconditions can be constructed. For more information see the paper by Beckman et al. [1]. In this example $((x \bmod 2) \neq 0)$ is a suitable predicate. Once the predicate has been constructed the first region at the frontier is split into versions where p is true and where p is false, which can be seen in Figure 3(b). The edge from region $(3 \wedge \neg p)$ to region 4 is eliminated, which removes this abstract error trace.

On the second iteration a new abstract error trace $(2, 3 \wedge p, 4)$ is found. Again DASH attempts to generate a test to cross the frontier (marked in Figure 3(b)), but this time the predicate p in the region $(3 \wedge p)$ is added to the constraint for test generation. However, the constraint is again unsatisfiable and the abstraction will be refined. The resulting graph from splitting the frontier with a new predicate q can be seen in Figure 3(c). Now the abstraction no longer contains a path from the initial region to the error and therefore the verification task is done.

We have implemented the DASH algorithm as a modification to the Lime Concolic Tester (LCT) [4], which is an open source dynamic symbolic execution (DSE) tool for C and Java programs. Our tool LCT-D extends the LLVM based C support in LCT.

LCT uses a client-server model to distribute test execution and constraint solving work. A testing server keeps track of the execution tree and selects which paths are to be explored next. When a client, which is an instrumented version of the program under test, connects to the

server a new path to explore is selected and the constraint corresponding to that path is sent to the client. The client calls an SMT solver with this constraint and if it is satisfiable the client executes the program with the obtained inputs. During execution the client sends details of each instruction it executes to allow the server to record constraints for generating further tests. The clients lose all state after each execution and all persistent state is stored on the testing server.

Our implementation of DASH in LCT-D follows the same general model. However, in DASH the clients (which have access to the executable program) are used in two modes: (1) to execute tests with previously solved inputs and (2) to solve constraints for generating new tests. In our tool these two modes can not be combined like they can be in normal DSE, because when solving constraints we execute the program up to the frontier to recover the concrete state of the program. This could be avoided by storing complete program states for test runs on the server, but we chose not to due to memory usage concerns. When a new set of inputs is solved on a client it is sent to the server to be used in a subsequent test execution. Currently LCT-D does not support multiple concurrent clients.

The DASH algorithm requires some way to map states visited in concrete test executions back to regions in the abstraction. The YOGI tool [5], in which the DASH algorithm was originally implemented, does this by evaluating the predicates in regions with the complete concrete states along a test execution. One of our contributions to DASH is how LCT-D infers the correct regions from control flow information combined with only the concrete values of pointers used by the program.

For more information on LCT-D and our improvements to DASH see the Master's Thesis of Olli Saarikivi [6]. The version of LCT-D used for the evaluation in the Master's Thesis is available at `http://users.ics.aalto.fi/osaariki/lctd-msc/`.

References

[1] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. Proofs from tests. In Barbara G. Ryder and Andreas Zeller, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 3–14. ACM, 2008.

[2] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[3] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM, 2005.

[4] Kari Kähkönen, Tuomas Launiainen, Olli Saarikivi, Janne Kauttio, Keijo Heljanko, and Ilkka Niemelä. LCT: An open source concolic testing tool for Java programs. In *Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, pages 75–80, 2011.

[5] Aditya V. Nori, Sriram K. Rajamani, SaiDeep Tetali, and Aditya V. Thakur. The YOGI project: Software property checking via static analysis and testing. In Stefan Kowalewski and Anna Philippou, editors, *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5505 of *Lecture Notes in Computer Science*, pages 178–181. Springer, 2009.

[6] Olli Saarikivi. Test-guided proofs for C programs on LLVM. Master's thesis, Aalto University, School of Science, Department of Information and Computer Science, 2013. `http://users.ics.aalto.fi/osaariki/msc-thesis-osaariki.pdf`.