Research Exam

Eric Seidel

Abstract

Program testing is a crucial yet tedious part of software development. The standard *unit-testing* practice dictates manual specification of input-output pairs for each expected behavior of a program, which involves large amounts of programmer effort and does not scale well with the size of a program. As a result, many groups have investigated techniques for automatically generating test-cases from a specification or directly from the source code.

We survey a selection of the prominent approaches for test-case generation and present our own contribution to the field, a technique for generating inputs to highly constrained functions by specifying the expected behavior via *refinement types*.

1. Introduction

There are two core questions an automatic test-case generator must answer:

1. How do we generate input values?
2. How do we determine the correctness of an execution?

In general, the answer for (2) involves checking that the execution satisfies some property, e.g. crash-freedom. As such, we will primarily categorize systems by their answer to (1).

2. Enumerating Inputs

Perhaps the simplest method of automatically testing a program is to enumerate valid inputs and check whether the program behaves correctly on these inputs. Of course, enumerating *all* inputs is generally infeasible, so we must find some way of narrowing the search space. The two common solutions are enumerating *small* inputs and random sampling of the entire space.

2.1 Enumerating "small" inputs

The *small-scope hypothesis* [14] argues that if a property is invalid, there is likely a small counterexample, i.e. if a program contains a bug there is likely a small input that will trigger it. Thus, we can restrict our enumeration to "small" inputs and still gain a large degree of confidence in our program.

2.1.1 SmallCheck

2.1.2 Korat

2.2 Random sampling of inputs

2.2.1 QuickCheck

2.2.2 JCrasher

2.2.3 Randoop

2.3 Limitations

2.3.1 Preconditions

- need for programmer intervention to specify "smart" generators
- or fall back to generate-and-filter approach
- can be mitigated to some extent by lazy construction of inputs

3. Enumerating Code Paths

The drawback to explicit enumeration of input vectors is that many inputs will trigger similar behavior in the program under test. Indeed unit testing texts often advise programmers to first partition program inputs into *equivalence classes*, and then test a single input vector from each equivalence class, thereby minimizing the number of handwritten tests required [3]. So instead of enumerating inputs, perhaps we should enumerate program behaviors, i.e. paths through the program.

Tools that take this approach typically use *dynamic-symbolic execution*, which combines traditional symbolic execution with concrete execution, to quickly explore different paths through the program. The two main categories of dynamic-symbolic execution-based testing tools are concolic testing and execution-generated testing, both introduced independently in 2005 [5, 13].

3.1 Symbolic Execution

Symbolic execution as a method of testing programs is not a new idea, it was introduced in 1976 by King [15]. The key difference in between symbolic and concrete execution is that instead of mapping program variables to *values*, a symbolic executor maps them to *symbolic expressions*. For example, given the simple program

```
int f (int x, int y) {
  return 2 * (x + y);
}
```

a concrete execution may begin with input vector $\{x \mapsto 1, y \mapsto 2\}$ and return 6. A symbolic execution, however, will begin with an input vector $\{x \mapsto \alpha_1, y \mapsto \alpha_2\}$ – where α_i are symbolic variables – and return $2 * (\alpha_1 + \alpha_2)$, thereby precisely describing *all* possible executions of `f`.

Another key difference of symbolic execution is its handling of conditionals. Consider the first conditional in the following program.

```
int f (int x) {
  if (x > 0) {
    if (x == 0) {
      abort();
    }
  }
  return 0;
}
```

With the input vector $\{x \mapsto \alpha_1\}$, the symbolic executor does not know which direction of the branch it should take, as it knows nothing about the symbolic variable α_1. Therefore it must follow both directions! When following a branch, the symbolic executor

records the symbolic expression associated with the chosen direction in its *path constraint*, which we will write as a sequence of expressions $\langle e_1, e_2, \ldots \rangle$. For example, in the outer conditional above, the "true" case would record $\langle \alpha_1 > 0 \rangle$ and the false case would record $\langle \neg(\alpha_1 > 0) \rangle$. Thus, it remembers what properties of the program inputs will trigger specific paths through the code. When the symbolic executor reaches a branch point, it consults the current path constraint to determine with directions are feasible. For example, upon reaching the inner conditional above, the symbolic executor will check whether $\alpha_1 = 0$ is consistent with the path condition $\langle \alpha_1 > 0 \rangle$, i.e. is the formula $\alpha_1 = 0 \wedge \alpha_1 > 0$ satisfiable? As the formula is clearly unsatisfiable, the symbolic executor will decide that the "true" branch is *unreachable*, and continue by only pursuing the "false" branch. Thus, a symbolic executor can statically determine that the `abort()` call above can *never* be executed.

While a powerful idea in theory, symbolic execution crucially relies on a theorem prover to solve the symbolic expressions it creates, and as such it went relatively unused until recent advances in constraint solving technology.

3.2 Concolic Testing

3.2.1 DART

In 2005 Godefroid et al. introduced the notion of *concolic testing*, which combines symbolic and concrete execution, with DART [13]. DART instruments a C program to execute each instruction both concretely and symbolically, then performs a depth-first search of all paths through the program, starting with a random input vector. At each branch point, DART records the branch condition and the direction taken, thereby building a *path constraint*. For example, suppose DART is testing the following C program with initial inputs $\{x \mapsto 5, y \mapsto 6\}$.

```
int f (int x, int y) {
  if (x == 5) {
    if (2 * y == x) {
      abort();
    }
  }
  return 0;
}
```

This execution will satisfy $x = 5$ but not $2y = x$, thus the path constraint will be $\langle x = 5,\ 2y \neq x \rangle$. Next, DART will negate the last (right-most) predicate in the path constraint and query a constraint solver for a solution to $x = 5 \wedge 2y = x$, in order to produce a new input vector. There is only one solution to this constraint, $\{x \mapsto 5, y \mapsto 10\}$, which will force execution through the *true* branch of both conditionals, right into the erroneous `abort()` call. Since the concrete execution reached the `abort()` call, we know it is a real bug as opposed to a false positive that could come from a purely symbolic approach, i.e. DART *soundly* reports bugs.

When confronted with an expression that it cannot reason about symbolically, e.g. multiplication of two variables or a dereference of a pointer that depends on program input, DART will fall back to recording the result of the concrete evaluation. For example, given

```
int f (int x, int y) {
  if (x == y*y) {
    abort();
  }
  return 0;
}
```

and starting inputs $\{x \mapsto 5, y \mapsto 2\}$, DART will produce a path constraint $\langle x \neq 4 \rangle$ for the first execution. Refuting this path constraint will *not* produce an input vector that is guaranteed to take the *true* branch – indeed the solver may return the original input vector – thus DART suffers a severe loss of precision when the program veers outside the language of the constraint solver. In effect, this means DART degenerates to brute-force enumeration of inputs, as in Sec 2.

Furthermore, DART's depth-first enumeration of paths means that it may fail to discover all paths when presented with recursive programs, e.g. a program that checks the ordering invariant of a binary-search tree. In this case DART will loop forever, generating increasingly deep trees whose right sub-trees are always `NULL` (assuming the program checks the left sub-tree first).

3.2.2 CUTE

Sen et al. introduced CUTE [17] later that year, an extension of DART that adds support for testing complex datatypes. CUTE enhances DART's technique by adding support for (dis)equality constraints on pointers, and by switching to a *bounded* depth-first search.

Pointer (dis)equality Whereas DART maintained a single map of memory locations to symbolic expressions, CUTE maintains two maps of memory locations: (1) $\mathcal{A}$ to arithmetic expressions and (2) $\mathcal{P}$ to pointer expressions. $\mathcal{A}$ contains the usual linear arithmetic expressions as in DART; however, $\mathcal{P}$ contains expressions of the form $x_p \cong y_p$ where x_p is either a symbolic variable or the constant symbol `NULL` and $\cong \in \{=, \neq\}$. When solving a pointer constraint, CUTE partitions the variables in $\mathcal{P}$ into equivalence classes and applying the arithmetic constraints to all members of the equivalence class. For example, given

```
int f (int *x, int *y) {
  if (x == y) {
    if (*x == 5) {
      return 0;
    }
  }
  return 0;
}
```

and the path constraint $\langle x = y,\ *x \neq 5 \rangle$, when CUTE refutes the $*x \neq 5$ conjunct, the value of $*y$ will *also* be forced to 5 as x and y are in the same equivalence class.

Bounded Depth-First Search In order to avoid an infinite loop from the repeated inlining of a loop body or recursive call, CUTE places a configurable bound k on the number of predicates in the path constraint. Once the path constraint is full, CUTE stops recording any further nested branch conditions, thereby forcing the refutation process to negate an earlier constraint. For example, given

```
int f (int n) {
  for (int i = 0; i < n; i++ ) {
    ...
  }
  return 0;
}
```

and $k = 4$, CUTE will never force more than four iterations of the loop body, as the path constraint will be cut off at $\langle i_0 < n,\ i_1 < n,\ i_2 < n,\ i_3 < n \rangle$. Negating the last conjunct will force $n \leq 3$, and CUTE will begin to backtrack through the path constraint until it terminates. While this tactic forces broad rather than deep coverage, it also means that CUTE may miss bugs deep in the execution graph of the program, e.g. if the loop body above were `if (i == 5) abort();`.

Another tactic CUTE employs to quickly achieve high coverage is branch prediction. Since CUTE only refutes the final conjunct of the path constraint, the outcomes of the previous branches should remain the same. Deviation from the previous path at an earlier branch indicates an imprecision in the symbolic executor; in this case CUTE will decide to restart execution with random inputs instead of allowing the loss of precision.

3.2.3 PEX

Tillman and Halleaux further extended concolic testing with Pex [18] in 2008, adding heuristics to improve path-selection, modeling of interactions with the environment, and a richer constraint language.

Richer constraints Whereas previous systems had limited constraint languages – linear arithmetic for DART, with the addition of pointer equality for CUTE – Pex takes advantage of the rich constraint language offered by Z3 [9]. Pex supports linear arithmetic, bit-vectors, arrays directly via Z3. Pex further supports floating-point numbers with an approximation to rational numbers.

Improving path-selection Instead of performing a depth-first search of all program paths, Pex maintains a tree of all branch conditions it has encountered. After exploring a path, Pex will choose a new unexplored path from the unexplored leaves of the execution tree, using several heuristics to partition branches into equivalence classes and then choosing a new branch from the least-often chosen class. Thus, Pex favors a more breadth-oriented search than DART or CUTE, while avoiding randomness in its path-selection.

Dealing with the environment Pex builds a model of the environment by recording the inputs and outputs of function calls where the source code is unavailable. This allows Pex to increase its precision when determining the feasibility of a path, but it also makes Pex unsound as the model is necessarily an under-approximation.

3.3 Execution-Generated Testing

Instead of performing symbolic and concrete execution in tandem, *execution-generated testing* [5] begins with pure symbolic execution and lazily generates concrete inputs on demand. When a dangerous operation (e.g. division or memory read/write) is about to be executed, the system will insert an implicit branch denoting the possibility of an error (e.g. divide-by-zero or out-of-bounds write). If the error branch is deemed feasible, the system will then solve the path constraint for an input vector designed to trigger the error condition. Similarly, function calls into uninstrumented code, e.g. library functions or system calls, will induce a call to the constraint solver for a concrete set of inputs designed to trigger the call. When the external call returns, the system will continue execution with the concrete result, thus improving precision over pure-symbolic approaches that would have to somehow model the interaction with the external world (often simply assuming nothing about the result).

3.3.1 EXE

Cadar et al. introduced execution-generated testing with EXE [6]. EXE models program memory as arrays of bitvectors, enabling bit-precise reasoning about the C programs it tests via the co-developed constraint solver STP [12]. This crucial distinction from DART and CUTE allows EXE and STP to view program values in the same way as the systems software they test, as untyped bytes.

At each branch EXE forks execution for each direction of the branch that is deemed feasible. The child processes add their direction to the path contraint and go to sleep. A master process then decides which child (path) should continue executing, using a combination of depth-first and best-first search. The master process chooses the child blocked on the instruction with the lowest execution count and runs it and its children in DFS for some period of time. Then it picks another best candidate and repeats the process.

An important optimization of EXE is *aggressive concretization*. If the operands are all concrete (i.e. constant values), EXE will simply perform the operation and record the resulting concrete value. This helps simplify the queries sent to STP, such that the only symbolic variables in a query will have a data dependence on one of the initial symbolic variables.

3.3.2 KLEE

In 2008, Cadar et al. rewrote EXE as KLEE [4], which symbolically executes LLVM IR [16] and provides several enhancements over EXE.

Compact process representation Whereas EXE processes relied on the host OS to share memory and was thus limited to page-level granularity, KLEE implements sharing with a granularity of individual objects, thus tracking many more processes than EXE could with the same memory limit. This optimization enabled KLEE to scale up to testing all of GNU Coreutils.

Random path selection and Coverage-optimized search KLEE employs two path selection strategies in round robin to prevent either one from getting stuck. *Random path selection* maintains a tree of all branches KLEE has encountered. It starts at the root and randomly picks a child node until it hits a leaf, and schedules the corresponding process for execution. This favors broad and shallow coverage, while still allowing for deep paths to be chosen. *Coverage-optimized search* weights each process according to some heuristics, e.g. distance to an unexecuted instruction, and biases the choice accordingly.

Environment modeling KLEE models the environment at the level of system calls, by replacing the actual system call with a simplified C implementation. Thus there is no "foreign" code and the developers can model interactions with the external world with as much precision as they desire. The drawback is that KLEE must now additionally reason about the mock system calls (as well as any library code leading up to them).

4. Tests from Counterexamples

In the previous section we discussed approaches whose aim was to achieve high program coverage, i.e. to execute as many instructions as possible in a short period. However even this may seem wasteful in the presence of tools that can *prove* a program correct.

Program verification is the process of analyzing a program and constructing a formal proof that it satisfies some correctness condition. As before we will use crash-freedom as our correctness condition, as high-level safety properties can be rewritten in terms of crash-freedom. A verifier is considered *sound* if it never reports a false positive, i.e. if the verifier claims your program is bug-free, it truly is. The converse does not generally hold; even if your program is bug-free the verifier may still report a possible bug, as it often has to *over-approximate* program behavior in order to achieve soundness. For instance, many verifiers struggle with non-linear arithmetic, i.e. they would be unable to verify

```
int f (int x, int y) {
  if (x > 0 && y > 0) {
    return 1 / (x * y);
  }
}
```

because the underlying theorem prover cannot handle multiplication of two variables. Thus, when a verifier reports a potential bug,

the programmer must still manually inspect the verifier's output to determine if the bug is genuine or fictitious. Luckily, many theorem provers produce a counterexample when verification fails. The insight of the tools we discuss in this section is that these counterexamples can be transformed into concrete test cases designed to trigger the erroneous behavior. Thus, one only need test the paths that cannot be statically proven safe.

Check'n'Crash [7] builds on top of JCrasher and the ESC/Java contract checker [11]. It runs ESC/Java on the supplied program and then solves the constraint system arising from a counterexample for concrete program input. Check'n'Crash can solve constraints involving integer arithmetic, object aliases, and multidimensional arrays, and can always fallback to the purely random testing of JCrasher if it cannot solve the constraint system. It then uses JCrasher to automatically generate test methods from the solutions. Note that a counterexample may assign program variables to symbolic expressions instead of concrete values, e.g. in the above the counter example would be $x > 0 \wedge y > 0$, thus Check'n'Crash must enumerate all possible solutions to the counterexample to be sure the bug does not exist.

DSD-Crasher [8] extends Check'n'Crash by first running the Daikon [10] invariant detection tool on the program. The inferred invariants are translated into JML specifications so that ESC/Java can digest them and avoid paths that would be triggered by invalid inputs. Thus, DSD-Crasher is able to generate test-suites with fewer false positives than Check'n'Crash, as it infers the programmer's intent. The drawback, however, is that Daikon requires a sizeable test-suite to infer precise invariants, so the prospective user of DSD-Crasher is left with something of a chicken-and-egg problem.

Beyer et al. [1] take a slightly different approach, using the BLAST [2] model-checker to generate test vectors that drive execution to each location where a user-supplied predicate p holds. They use BLAST to translate a C program into a control-flow automaton, which it then traverses to generate all traces that satisfy p at the final location. These traces are sequences of assignments and assumptions about the program state (e.g. from taking a specific direction of a branch), and must be converted into concrete test vectors before they can be executed. BLAST then translates these traces into logical formulae encoding constraints on the program variables and queries a theorem prover for a satisfying assignment, which finally represents a concrete test vector.

An advantage of this approach over the Check'n'Crash approach is that the user can supply any predicate they wish and BLAST will find states that satisfy it, whereas Check'n'Crash will only find states that ESC/Java deems unsafe. (One could insert explicitly failing assertions in specific locations to guide Check'n'Crash, but this is more work for the user.)

5. Type-Targeted Testing

- *Filtered* enumeration of inputs

References

[1] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 326–335, Washington, DC, USA, 2004. IEEE Computer Society.

[2] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, September 2007.

[3] Ilene Burnstein. *Practical Software Testing: A Process-Oriented Approach*. Springer Science & Business Media, June 2003.

[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[5] Cristian Cadar and Dawson Engler. Execution generated test cases: How to make systems code crash itself. In Patrice Godefroid, editor, *Model Checking Software*, number 3639 in Lecture Notes in Computer Science, pages 2–23. Springer Berlin Heidelberg, 2005.

[6] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[7] Christoph Csallner and Yannis Smaragdakis. Check 'n' crash: Combining static checking and testing. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 422–431, New York, NY, USA, 2005. ACM.

[8] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. DSD-crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2):8:1–8:37, May 2008.

[9] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] M.D. Ernst, J. Cockrell, William G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[11] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[12] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, number 4590 in Lecture Notes in Computer Science, pages 519–531. Springer Berlin Heidelberg, 2007.

[13] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[14] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[15] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[16] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*, pages 75–86, March 2004.

[17] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[18] Nikolai Tillmann and Jonathan de Halleux. Pex–white box test generation for .NET. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs*, number 4966 in Lecture Notes in Computer Science, pages 134–153. Springer Berlin Heidelberg, January 2008.