Automated Specification-Based Testing

Eric Seidel

Abstract

Program testing is a crucial yet tedious part of software development. 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 automatic 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

Software testing is a time-consuming and expensive task. The National Institute of Standards & Technology (NIST) reports that an average of 50-70% of development time for a software product is spent on testing [42]. Furthermore, the cost of inadequate testing can be severe, NIST estimates the annual economic cost of inadequate software testing infrastructure at $22.2 - $59.5 billion. Thus there is a strong incentive to develop improved testing methods and tools.

At a high level, software testing can be divided into three levels, based on the object being tested [4]. *Unit testing* involves testing individual components of a system – often at the level of individual functions or modules – in isolation, and is almost always performed by the programmers themselves. *Integration testing* refers to testing the interactions between units of a system. Finally, *system testing* exercises the entire assembled software product, and may be performed by developers or target users. In this report we will focus on automating unit testing, though the techniques we present could also be applied to the other levels.

Two important questions that an automated test-case generation technique must answer are:

1. How shall we determine the success of a test run?
2. How shall we generate appropriate input vectors?

Characterizing correctness At the unit testing level, a common practice is to collect a set of input/output pairs that characterize the expected behavior of the system, and test that given an input the system produces the expected output. However, on its own this does not scale well to automatic test-generation (how do we determine *a priori* the expected output?). Thus, the systems we investigate will all use some notion of *specification-based testing* [31]. A specification is a formal statement of the expected behavior of the system under test, and may range from the simple "this program should not crash" to the relatively complex "this program takes as input a sorted list xs and an element x, and returns a new sorted list containing all elements of xs as well as x". In fact, a complex safety specification can be reduced to crash-freedom by the function under scrutiny in another function that dynamically checks the safety condition and crashes if it is violated. Thus, in this report we will focus more on the method of generating input vectors than the specification mechanism.

Input generation There are two broad approaches to constructing test vectors for a system. *Functional* or *black-box* testing treats the system as opaque and selects input candidates based solely on the external specification. On the other hand, *structural* or *white-box* testing uses knowledge of the system's internal to select test vectors (e.g. to optimize statement coverage or attempt to steer control-flow to a dangerous location). Both categories are widely used, and in this report we will investigate techniques that fall into both.

1.1 Organization

The rest of this report is organized as follows. In Sec. 2 we describe black-box techniques that explicitly enumerate input vectors. In Sec. 3 we investigate white-box techniques that attempt to enumerate *paths* through the system, thus limiting the number of concrete test-cases required. In Sec. 4 we examine white-box techniques that make use of a theorem prover to filter out program paths that can be statically proven correct. Finally, in Sec. 5 we present our own contribution to the field, a black-box technique for testing programs with highly-constrained input domains.

2. Enumerating Inputs

Perhaps the simplest method of automatically testing a program is to enumerate valid inputs based on its external interface – an instance of *black-box testing* [1] – 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* [26] 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

SmallCheck [39] is a testing library for Haskell programs that does the simplest thing possible. Programmers specify how to enumerate input values using Haskell's type-class mechanism [46] and provide a boolean-valued function that describes the property they wish to check, and SmallCheck enumerates all possible inputs up to some depth, validating the property on each input vector. The input generators are expected to produce values in increasing size (indeed, such a generator can be automatically derived from a datatype definition), thus SmallCheck is guaranteed to find the *minimal* counterexample if it lies within the depth-bound.

Naturally, such a brute-force enumeration of input vectors is bound to be wasteful (i.e. it will produce many inputs that trigger the

same code path), thus the authors also introduce a *lazy* variant that takes advantage of Haskell's inherent laziness to prune the search space. The key distinction of Lazy SmallCheck is that it produces *partially-defined* values instead of fully-defined values, i.e. Lazy SmallCheck will initially leave each component of a value uninitialized, so that the Haskell runtime will thrown an exception if the component is read. Only when a value is demanded will Lazy SmallCheck fill in the "hole" with an actual value. Thus, Lazy SmallCheck reduces the search space by dynamically discovering which parts of a value are relevant to the property being tested.

Initially, Lazy SmallCheck had several disadvantages compared to SmallCheck. It could not generate functional values (commonly used in Haskell), and it could not print the partially-defined counterexample, instead it would have to concretize the input vector, thus *losing* information. These were both added later as enhancements [38]. A remaining concern is that programmers must be careful about the order in which they conjoin predicates, giving precedence to the lazier predicate. For example, a binary-search tree implementation will want to check the ordering invariant before balancedness; the former will often fail within the first few nodes whereas the latter will force the entire spine.

2.1.2 TestEra / Korat

TestEra [28, 33] provides (bounded) exhaustive testing of Java methods against an Alloy [25] specification. Unlike SmallCheck, which enumerates input vectors by unfolding each possible data constructor and enumerating its arguments, TestEra begins with a pre-defined universe of atomic values (e.g. primitive types and uninitialized objects), and uses the Alloy Analyzer [24] to enumerate all valid *relations* between the atoms, i.e. setting the pointers appropriately such that the method's preconditions are satisfied. Thus, an important optimization that TestEra implements is filtering *isomorphic* inputs. For each symbolic input vector produced by Alloy, TestEra *concretizes* the input to a set of Java objects, executes the method, and then *abstracts* the result back to an Alloy value. Finally, it queries the Alloy Analyzer again to determine whether the output satisfies the method's postcondition, then proceeding to the next input vector. Unlike SmallCheck, TestEra is not guaranteed to find a minimal counterexample as the testing order is left up to Alloy.

Korat [5] is a rewrite of TestEra that uses a custom enumeration algorithm instead of calling out to Alloy. Korat enumerates all input candidates in lexicographic order, and runs an instrumented version of a programmer-supplied `repOk` method (to check class invariants and preconditions) that tracks field accesses. If `repOk` returns *true* Korat immediately outputs all (non-isomorphic) input vectors that share the current valuation of the *accessed* fields, as the valuation of the unread fields cannot have affected the outcome of `repOk`. More importantly, if `repOk` returns *false* Korat immediately backtracks, skipping all variant candidates that share the current valuation of the accessed fields, as *none* of them can be valid. Thus, Korat takes advantage of the inherent laziness of invariant-checking predicates to quickly prune the search space of invalid inputs, in a very similar manner to Lazy SmallCheck.

2.2 Random sampling of inputs

The obvious drawback to the approaches described above is that they will not detect bugs that only present on "large" inputs. Thus, a common alternative to enumeration of small inputs is random selection of input vectors from the entire search space.

2.2.1 QuickCheck

QuickCheck [11, 12] enables randomized testing of Haskell programs by providing an embedded domain-specific language for generating *arbitrary* values of a given datatype. As with SmallCheck, QuickCheck properties are boolean-valued Haskell functions whose inputs can be (randomly) generated, and the input-generators (usually) operate by unfolding a specific data constructor and generating sub-values for the constructor's fields. Unlike SmallCheck it is impractical to automatically derive QuickCheck generators for datatypes, as one must take care to ensure the generator covers a uniform distribution of values. For example, a generator for a simple "list" type

```
data List a = Nil | Cons a (List a)
```

that chooses between `Nil` and `Cons` with equal probability is highly unlikely to generate lists with more than a handful of elements. A further concern arising from random testing is that the returned counterexample may be quite large, as demonstrated by Pike [37]. Thus, subsequent iterations of QuickCheck introduced support for *shrinking* counterexamples [23]. Once QuickCheck has found a counterexample it will invoke a user-defined `shrink` function on the input vector, which will return a list of smaller inputs. QuickCheck will test each small candidate in turn and repeat the shrinking process on any new counterexamples, finally returning the smallest counterexample it could find. Notably, the shrinking process is not guaranteed to find a *minimal* counterexample.

Pike [37] builds on top of QuickCheck with SmartCheck, which provides automatically-derivable shrinking definitions that are shown to perform favorably compared to handwritten `shrink` functions, and produce smaller counterexamples. More interestingly, SmartCheck also introduces *counterexample generalization*, which attempts to produce a universal property describing a class of counterexamples. For example, Pike shows that SmartCheck can reduce a large counterexample like

```
StackSet
 (Screen (Workspace 0 (-1)
   (Just (Stack 'S' "" ""))) 1 1)
 [Screen (Workspace 2 (-1) Nothing) 2 (-1),
  Screen (Workspace 3 (-1) Nothing) 0 (-1)]
 [Workspace 1 (-1) (Just (Stack 'NUL' "" "")),
  Workspace 4 (-1) (Just (Stack 'I' "" ""))]
 (fromList [])
```

to a comparatively simple formula

```
forall values x0 x1 x2 x3:
  StackSet
    (Screen (Workspace x0 (-1) (Just x1)) 1 1)
    x2 x3 (fromList [])
```

thus abstracting away the irrelevant portions of the counterexample. The key insight is that if one can replace a sub-value by another arbitrary value without affecting the test outcome, then the sub-value must not affect the outcome. Thus, SmartCheck systematically replaces all sub-values of the counterexample with other random values and generalizes the counterexample accordingly.

QuickCheck does not currently have good support for testing properties with preconditions, due to the low probability of randomly generating a value that satisfies the precondition. [10] describes an algorithm for random generation of constrained inputs based on [18], by defining a function to index into a uniform distribution of constrained values, and then generating random indices, but the work has not yet been incorporated into QuickCheck.

2.2.2 JCrasher

Instead of constructing input vectors directly, JCrasher [13] constructs them indirectly via *method chaining*. Given a set of Java

classes, JCrasher constructs a parameter graph, where the nodes are public methods, constructors, and primitive values, and the edges run from method parameters to nodes producing values of the needed type. By randomly choosing paths through the graph starting from the method under test, JCrasher creates sequences of method and constructor calls that should produce valid input vectors. JCrasher executes these method chains followed by the target method, under the assumption that public methods and constructors should not produce inputs that will crash a program.

There is some subtlety in the use of "crash", as Java methods will frequently throw exceptions when given invalid input parameters. We should not consider precondition violations as "crashes" as the responsibility of providing valid inputs rests with the *caller*, not the *callee* [34]. Thus, JCrasher includes a number of heuristics to determine whether a thrown exception should be considered a bug. For example, an `IllegalArgumentException` can be considered a bug if it was thrown by a transitively-called method, but not if it was thrown directly by the method under test, as that would indicate that our test vector was at fault. On the other hand, an `ArithmeticException` (e.g. divide-by-zero) can always be classified as a bug, as the method under test should have caught and handled it.

2.2.3 Randoop

Randoop [36] extends the method-chaining approach of JCrasher by incorporating feedback from previously generated test vectors. The main insight is that if a sequence of methods s results in a crash, there is no point in checking any sequences that include s as a prefix. Thus, Randoop iteratively constructs longer chains of method calls, only extending existing chains if they do not cause the program to crash. Furthermore, Randoop applies some filters to the generated sequences before testing them to further reduce the search space, e.g. by discarding sequences that immediately throw an exception or produce a value that equal to the result of an existing sequence.

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 [6]. So instead of enumerating inputs, perhaps we should enumerate program behaviors, i.e. paths through the program. This necessarily requires knowledge of the internal structure of the program under test, thus tools that take this approach will fall in the category of *white-box testing* [1].

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 [8, 22].

3.1 Symbolic Execution

Symbolic execution as a method of testing programs is not a new idea, it was introduced in 1976 by King [29]. 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

Godefroid et al. introduced *concolic testing* in 2005 [22]. Concolic testing performs symbolic and concrete execution of a program in tandem. Thus, when confronted with a program expression that the symbolic executor cannot reason about, a concolic tester can fall back to the concrete value and continue execution with more precision than a purely symbolic approach.

3.2.1 DART

DART [22] 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 path-constraint 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 [40] 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 arithmetic 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 [43] 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 [16]. 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* [8] 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 [9]. 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 [21]. 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 [7], which symbolically executes LLVM IR [30] 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. Filtered Enumeration of Code Paths

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 [35]. 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 [14] builds on top of JCrasher and the ESC/Java contract checker [20]. 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 [15] extends Check'n'Crash by first running the Daikon [19] 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. [2] take a slightly different approach, using the BLAST [3] 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

In this section we describe our own contribution to automatic test-case generation, a black-box approach for testing programs with highly-constrained input domains. Our approach, which we call *type-targeted testing* – abbreviated to Target – uses *refinement types* to describe function contracts and then automatically generates all inputs, up to a given depth, that satisfy the function's precondition.

5.1 Refinement Types

A refinement type – written `{v:T | p}` – refines a base type `T` with a logical predicate `p` that all values `v` of the refined type must satisfy. For example, the following types

```
type Nat   = {v:Int | 0 <= v}
type Pos   = {v:Int | 0 <  v}
type Rng N = {v:Int | 0 <= v && v < N}
```

describe the set of integers that are non-negative, strictly positive, and in the range $[0, N)$ respectively. We can also construct refined collection and function types by refining the individual components.

Traditionally, refinement types have been used as a means of program verification [17, 41, 45, 47], by translating the program into a logical formula that can be efficiently analyzed by an SMT solver. SMT (*Satisfiability Modulo Theories*) solvers combine a satisfiability solver for propositional logic with decision procedures for various external theories, such as linear integer arithmetic, bitvectors, and uninterpreted functions. We are not going to discuss refinement types in the context of verification however; instead we will use refinement types as a high-level specification mechanism for generating exhaustive test-suites.

Why test instead of prove? Target enables *gradual verification*, which we find has several advantages over pure verification. First, we provide an incentive to write formal specifications by automatically translating the specifications into exhaustive test-suites, thus giving the programmer immediate gratification without the need for hints, tactics, or strengthened inductive invariants that verification tools inevitably require. This makes Target more suitable for the initial exploratory phase of designing a program. Second, even once the design has settled and formal verification begins, Target allows the programmer to test functions that are too complex to formally verify, without having to resort to a different specification mechanism. Thus, functions in the verified portion of the program can *assume* that the tested functions satisfy their specifications. Finally, as we explained in the previous section, the concrete counterexamples that Target generates can be invaluable for debugging programs rejected by the verifier.

5.2 Synthesizing Tests from Refinement Types

Our high-level strategy for generating test vectors is to: (1) derive a set of logical constraints from the input types and *query* an SMT solver for a satisfying assignment, (2) *decode* the model into concrete Haskell values, (3) *execute* the function to obtain the output, (4) *check* that the output satisfies the output type, (5) *refute* the model the generate a different test vector, and loop back to step 2 until we have exhausted all inputs of a given size. We now describe steps 1, 2, and 4 with a series of examples.

Primitive Types Let us begin with a simple function that takes a value in a given range and *scales* it to fit in another range. Using the type aliases from Sec. 5.1, we specify and define `rescale` as

```
rescale :: r1:Nat -> r2:Nat -> s:Rng r1 -> Rng r2
rescale r1 r2 s = s * (r2 `div` r1)
```

We encode preconditions on primitive types directly from the refinement predicates, conjoining the constraints for multiple inputs. Thus, we generate the following input constraint for `rescale`:

$$\mathsf{C}_0 \doteq 0 \leq r1 \wedge 0 \leq r2 \wedge 0 \leq s < r1$$

Suppose the SMT solver solves the above constraint with the model $[r1 \mapsto 1, r2 \mapsto 1, s \mapsto 0]$, Target will then execute `rescale 1 1 0`, which results in output `0`. Next we *validate* the output against the postcondition by encoding the conjunction of the output type and value as a constraint, and checking validity of

$$r2 = 1 \wedge v = 0 \wedge 0 \leq v \wedge v < r2$$

This formula is valid, so Target requests another input vector by conjoining C_0 with an explicit refutation of the last model:

$$\mathsf{C}_1 \doteq \mathsf{C}_0 \wedge (r1 \neq 1 \vee r2 \neq 1 \vee s \neq 0)$$

Suppose this time the solver returns $[r1 \mapsto 1, r2 \mapsto 0, s \mapsto 0]$. Target will execute `rescale 1 0 0 = 0`, which does *not* satisfy the postcondition as $0 < 0$ is not valid. Thus Target will report `1, 0, 0` as a counterexample. An easy fix here is to require strictly positive ranges, i.e.

```
rescale :: r1:Pos -> r2:Pos -> s:Rng r1 -> Rng r2
```

after which Target is assuaged and reports no counterexamples.

Containers Next, consider a function that computes a weighted average of some scores.

```
type Score = Rng 100

average :: [(Pos, Score)] -> Score
average []  = 0
average wxs = total `div` n
  where
    total   = sum [w * x | (w, x) <- wxs ]
    n       = sum [w     | (w, _) <- wxs ]
```

Now Target must generate lists of constrained values. A list is either "nil" or an element "cons"ed onto another list, so we generate constraints that represent *all* lists up to the given depth using *choice variables* to encode the choice between alternative constructors in the solver's logic. Each model returned by the solver will contain both concrete values for primitive types *and* choices of specific data constructors at each level of the list, enabling us to reconstruct a concrete Haskell value. For example, we represent values of type `[(Pos,Score)]` – up to depth 3 – with the conjunction of $\mathsf{C}_{\mathsf{list}}$ and

$\mathsf{C}_{\mathsf{data}}$.

$$
\begin{aligned}
\mathsf{C}_{\mathsf{list}} \doteq\ & (c_{00} \Rightarrow xs_0 = []) \wedge (c_{01} \Rightarrow xs_0 = x_1 : xs_1) \\
\wedge\ & (c_{10} \Rightarrow xs_1 = []) \wedge (c_{11} \Rightarrow xs_1 = x_2 : xs_2) \\
\wedge\ & (c_{20} \Rightarrow xs_2 = []) \wedge (c_{21} \Rightarrow xs_2 = x_3 : xs_3) \\
\wedge\ & (c_{30} \Rightarrow xs_3 = []) \wedge (c_{21} \Rightarrow c_{30}) \\
\wedge\ & (c_{00} \oplus c_{01}) \wedge (c_{01} \Rightarrow c_{10} \oplus c_{11}) \wedge (c_{11} \Rightarrow c_{20} \oplus c_{21})
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{C}_{\mathsf{data}} \doteq\ & (c_{01} \Rightarrow x_1 = (w_1, s_1)\ \wedge\ 0 < w_1\ \wedge\ 0 \leq s_1 < 100) \\
\wedge\ & (c_{11} \Rightarrow x_2 = (w_2, s_2)\ \wedge\ 0 < w_2\ \wedge\ 0 \leq s_2 < 100) \\
\wedge\ & (c_{21} \Rightarrow x_3 = (w_3, s_3)\ \wedge\ 0 < w_3\ \wedge\ 0 \leq s_3 < 100)
\end{aligned}
$$

$\mathsf{C}_{\mathsf{list}}$ encodes the *structural* constraints on the input list, i.e. that at each level i the solver must choose between "nil" (c_{i0}) and "cons" (c_{i1}). We encode [] and : as *uninterpreted* functions in the constraints, enabling efficient analysis by SMT solvers.

At each level we *guard* the constraints with the choice variables. Thus, if the solver picks "cons", i.e. sets c_{i1} to $true$, it must pick precisely one of the choice variables at the next level, i.e. $c_{i1} \Rightarrow c_{(i+1)0} \oplus c_{(i+1)1}$. Furthermore, the data constraints at level $i+1$ are only required if the solver chose "cons" at level i, i.e. we only constrain the values that will be *realized*. This restriction is essential to ensure that Target does not miss certain test vectors, as we will see in the next example.

When we decode a model of the above constraints, we use the valuation of the choice variables to build the list incrementally. A valuation $c_{i0} \mapsto true$ results in the empty list `[]`, and a valuation $c_{i1} \mapsto true$ means we have to further decode x_{i+1} and xs_{i+1} and "cons" the results.

Suppose the solver returns

$$[c_{00} \mapsto false, c_{01} \mapsto true, w_1 \mapsto 1, s_1 \mapsto 2, c_{10} \mapsto true, \dots]$$

which we decode to the Haskell value `(1,2):[]`. When we refute this model, we will add a constraint

$$c_{00} \neq false \vee c_{01} \neq true \vee w_1 \neq 1 \vee s_1 \neq 2 \vee c_{10} \neq true$$

which notably ignores all logical variables that did not contribute to the realized value. This is an important optimization, as if we had refuted the *entire* model, the solver could have just changed the value of, e.g., w_3, producing a new but equivalent model.

Ordered Containers Now that we have seen how Target generates test vectors with structured data, let us consider an example where the input domain is highly constrained, namely the `insert` function from the eponymous sorting algorithm. `insert` requires that its input list be already sorted, and guarantees that its output list will also be sorted.

```
insert :: Ord a => a -> Sorted a -> Sorted a
```

we capture the ordering invariant with a new list type that recursively requires each element of the tail be less than the head.

```
data Sorted a = []
              | (:) { h :: a
                    , t :: Sorted {v:a | h < v}
                    }
```

In practice we would use a regular Haskell list with an *abstract refinement* [44] to enforce ordering, but for the sake of exposition we will just create a new datatype here.

As we unfold the `Sorted` argument, we will generate the same $\mathsf{C}_{\mathsf{list}}$ constraints as above, augmented with

$$\mathsf{C}_{\mathsf{ord}} \doteq (c_{11} \Rightarrow x_1 < x_2)\ \wedge\ (c_{21} \Rightarrow x_2 < x_3\ \wedge\ x_1 < x_3)$$

to enforce the ordering constraint. Now we can see why the guards are necessary for completeness, suppose we had instead left them out, producing

$$\mathsf{C}_{\mathsf{ord}'} \doteq x_1 < x_2\ \wedge\ x_1 < x_3\ \wedge\ x_2 < x_3$$

In practice, the constraints will also include conjuncts of the form $-N \leq x_i \leq N$ where N is the depth. Thus at depth 3, the solver would never be able to produce a model where $x_1 \mapsto 3$, i.e. we would never generate the test vector $\{xs \mapsto [3]\}$, which would be a valid input.

Structured Containers Finally, let us consider an example with constraints on the *structure* of the input vector. The `best` function returns the k highest scores in the given list.

```
best :: k:Nat -> {v:[Score] | k <= len v}
     -> {v:[Score] | k = len v}
best k xs = take k $ reverse $ sort xs
```

We encode the length of the input and output lists using a logical *measure function* [27].

```
measure len :: [a] -> Nat
len []     = 0
len (x:xs) = 1 + len xs
```

Each time we unfold a list constructor we instantiate its measure definition, producing a complete constraint that the SMT solver can reason about.

$$
\begin{aligned}
\mathsf{C}_{\mathsf{size}} \doteq\ & (c_{00} \Rightarrow len\ xs_0 = 0) \wedge (c_{01} \Rightarrow len\ xs_0 = 1 + len\ xs_1) \\
\wedge\ & (c_{10} \Rightarrow len\ xs_1 = 0) \wedge (c_{11} \Rightarrow len\ xs_1 = 1 + len\ xs_2) \\
\wedge\ & (c_{20} \Rightarrow len\ xs_2 = 0) \wedge (c_{21} \Rightarrow len\ xs_2 = 1 + len\ xs_3) \\
\wedge\ & (c_{30} \Rightarrow len\ xs_3 = 0)
\end{aligned}
$$

Inside the logic, len is just an uninterpreted function, the structural properties are spelled out explicitly by the constraints. To search for valid inputs, we add the top-level constraints

$$0 \leq k \wedge k \leq len\ xs_0$$

thus restricting Target to produce lists with at least k elements.

5.3 Implementation and Evaluation

We have implemented Target as a testing library for Haskell programs. Target takes as input a function with a refined type, providing the specification, and exhaustively enumerates all *valid* input vectors up to a given depth, verifying that each input produces a valid output. As with QuickCheck and SmallCheck, the user must provide a test-case generator for the input types and predicates describing the desired pre- and postconditions. Like SmallCheck (but *unlike* QuickCheck) the generators are extremely mechanical and can in fact be automatically derived from the data definition via a generics library like `GHC.Generics` [32]. Furthermore, as Target uses an SMT solver to filter out invalid values, we only require a *single* generator per input type, whereas SmallCheck often requires custom generators for each set of input constraints.

We have compared Target against QuickCheck and (Lazy) SmallCheck on a series of functions with complex preconditions. In order to evaluate our claim that Target's symbolic enumeration of inputs would fair better than concrete enumeration (even lazily), we restricted each system to a single test-case generator per type. We checked properties of a custom red-black tree implementation, height-balanced trees from the `Data.Map` library, and the core `StackSet` datatype of the `XMonad` tiling window manager. In all of our examples we found that: (1) QuickCheck was unable to randomly generate suitably constrained inputs without a custom gen-

erator, (2) SmallCheck was substantially faster than Target at generating *very small* inputs, and (3) Target was consistently able to explore *deeper* inputs than (Lazy) SmallCheck before timing out.

Regrettably, we know of no concolic testing libraries for Haskell, and so were unable to perform a direct comparison of our approach with concolic testing. We did however port a simplified version of the height-balanced tree example to `F#` to test Pex. We found that Pex was unable to generate any non-trivial (i.e. containing more than one element) trees, as the symbolic executor ended up enumerating paths through the *precondition*, unable to reach the actual function we were testing.

6. Conclusion

In this report we presented various techniques for automatically generating test vectors. First, we looked at black-box explicit enumeration of inputs, both by randomly searching the input domain and by exhaustively searching for "small" inputs. The drawback to such systems is that we generate many input vectors that trigger equivalent behavior from the system under test. Thus we next looked at white-box techniques that enumerate paths through the system, producing (ideally) only a single input vector per behavior. Such systems are limited by the expressiveness of their underlying constraint solver and by the fact that preconditions on the input are often given as code, thereby forcing the system to enumerate paths through the precondition. A further drawback is that many program paths can be statically proven safe, thus we also investigated techniques that use a program verifier to isolate the potentially unsafe paths and guide the search further. Finally, we presented type-targeted testing, a black-box technique for testing programs with complex preconditions by symbolically enumerating valid inputs.

References

[1] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, verification, and testing of computer software. *ACM Comput. Surv.*, 14(2):159–192, June 1982.

[2] 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.

[3] 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.

[4] Pierre Bourque, R. E Fairley, and IEEE Computer Society. *SWEBOK: guide to the software engineering body of knowledge*. IEEE Computer Society, [Los Alamitos, CA], 2014.

[5] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 123–133, New York, NY, USA, 2002. ACM.

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

[7] 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.

[8] 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.

[9] 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.

[10] Koen Claessen, Jonas Duregård, and Michal H. Palka. Generating constrained random data with uniform distribution. FLOPS '14, 2014.

[11] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[12] Koen Claessen and John Hughes. Testing monadic code with QuickCheck. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 65–77, New York, NY, USA, 2002. ACM.

[13] Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for java. *Software: Practice and Experience*, 34(11):1025–1050, September 2004.

[14] 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.

[15] 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.

[16] 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.

[17] Joshua Dunfield. Refined typechecking with stardust. In *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification*, PLPV '07, pages 21–32, New York, NY, USA, 2007. ACM.

[18] Jonas Duregård, Patrik Jansson, and Meng Wang. Feat: Functional enumeration of algebraic types. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 61–72, New York, NY, USA, 2012. ACM.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] John Hughes. QuickCheck testing for fun and profit. In Michael Hanus, editor, *Practical Aspects of Declarative Languages*, number 4354 in Lecture Notes in Computer Science, pages 1–32. Springer Berlin Heidelberg, 2006.

[24] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the alloy constraint analyzer. In *Proceedings of the 2000 International Conference on Software Engineering, 2000*, pages 730–733, 2000.

[25] Daniel Jackson. Automating first-order relational logic. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications*, SIGSOFT '00/FSE-8, pages 130–139, New York, NY, USA, 2000. ACM.

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

[27] Ming Kawaguchi, Patrick Rondon, and Ranjit Jhala. Type-based data structure verification. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 304–315, New York, NY, USA, 2009. ACM.

[28] Sarfraz Khurshid and Darko Marinov. TestEra: Specification-based testing of java programs using SAT. *Automated Software Engineering*, 11(4):403–434, 2004.

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

[30] 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.

[31] Gilbert Thomas Laycock. *The Theory and Practice of Specification Based Software Testing*. PhD thesis, University of Sheffield, 1993.

[32] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. A generic deriving mechanism for haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 37–48, New York, NY, USA, 2010. ACM.

[33] Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of java programs. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 22–31. IEEE, 2001.

[34] B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, October 1992.

[35] Charles Gregory Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, Stanford, CA, USA, 1980. AAI8011683.

[36] C. Pacheco, S.K. Lahiri, M.D. Ernst, and T. Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering, 2007. ICSE 2007*, pages 75–84, May 2007.

[37] Lee Pike. SmartCheck: Automatic and efficient counterexample reduction and generalization. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 53–64, New York, NY, USA, 2014. ACM.

[38] Jason S. Reich, Matthew Naylor, and Colin Runciman. Advances in lazy SmallCheck. In Ralf Hinze, editor, *Implementation and Application of Functional Languages*, Lecture Notes in Computer Science, pages 53–70. Springer Berlin Heidelberg, January 2013.

[39] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy smallcheck: Automatic exhaustive testing for small values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, pages 37–48, New York, NY, USA, 2008. ACM.

[40] 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.

[41] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 266–278, New York, NY, USA, 2011. ACM.

[42] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007(011), 2002.

[43] 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.

[44] Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. Abstract refinement types. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 209–228, Berlin, Heidelberg, 2013. Springer-Verlag.

[45] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. LiquidHaskell: Experience with refinement types in the real world. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 39–51, New York, NY, USA, 2014. ACM.

[46] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.

[47] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 249–257, New York, NY, USA, 1998. ACM.