Compositional Dynamic Test Generation

(Extended Abstract)

Patrice Godefroid [*]

Microsoft Research

pg@microsoft.com

Abstract

Dynamic test generation is a form of dynamic program analysis that attempts to compute test inputs to drive a program along a specific program path. Directed Automated Random Testing, or DART for short, blends dynamic test generation with model checking techniques with the goal of systematically executing all feasible program paths of a program while detecting various types of errors using run-time checking tools (like Purify, for instance). Unfortunately, systematically executing *all* feasible program paths does not scale to large, realistic programs.

This paper addresses this major limitation and proposes to perform dynamic test generation *compositionally*, by adapting known techniques for interprocedural static analysis. Specifically, we introduce a new algorithm, dubbed *SMART* for *Systematic Modular Automated Random Testing*, that extends DART by testing functions in isolation, encoding test results as function summaries expressed using input preconditions and output postconditions, and then re-using those summaries when testing higher-level functions. We show that, for a fixed reasoning capability, our compositional approach to dynamic test generation (SMART) is both sound and complete compared to monolithic dynamic test generation (DART). In other words, SMART can perform dynamic test generation compositionally without any reduction in program path coverage. We also show that, given a bound on the maximum number of feasible paths in individual program functions, the number of program executions explored by SMART is linear in that bound, while the number of program executions explored by DART can be exponential in that bound. We present examples of C programs and preliminary experimental results that illustrate and validate empirically these properties.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Verification, Algorithms, Reliability

Keywords Software Testing, Automatic Test Generation, Scalability, Compositional Program Analysis, Program Verification

[*] This work was done in part when the author was still affiliated with Bell Laboratories. It is also funded in part by NSF CCR-0341658. This version has been slightly revised compared to the POPL'07 Proceedings.

1. Introduction

Given a program P, say a C program with a million lines of code, with a set of input parameters, wouldn't it be nice to have a tool that could automatically generate a set of input values that would exercise, say, even only 50% of the code of P?

This problem is called the test generation problem, and has been studied since the 70's (e.g., [Kin76, Mye79]). Yet, effective solutions and tools to address this problem have proven elusive for the last 30 years. What happened?

There are several possible explanations to the current lack of practically-usable tools addressing this problem. First, the expensive sophisticated program-analysis techniques required to tackle the problem, such as symbolic execution engines and constraint solvers, have only become computationally affordable in recent years thanks to the increasing computational power available on modern computers. Second, this steady increase in computational power has in turn enabled recent progress in the engineering of more practical software model checkers, more efficient theorem provers, and, last but not least, more precise yet scalable static analysis tools. Indeed, automatic code inspection tools based on static program analysis are increasingly being used in the software industry (e.g., [BPS00, HCXE02]).

Recently, there has been a renewed interest on automated test generation from program analysis (e.g., [BKM02, BCH$^+$04, VPK04, CS05, GKS05, CE05]). Work in this area can roughly be partitioned into two groups: *static* versus *dynamic* test generation.

Static Test Generation is Often Ineffective

Static test generation (e.g., [Kin76]) consists of analyzing a program P statically, by exclusively using symbolic execution techniques to attempt to compute inputs to drive P along specific execution paths or branches, *without ever executing the program*. The idea is to symbolically explore the tree of all computations the program exhibits when all possible value assignments to input parameters are considered. For each *control path* ρ, that is, a sequence of control locations of the program, a *path constraint* ϕ_ρ is constructed that characterizes the input assignments for which the program executes along ρ. All the paths can be enumerated by a search algorithm that explores all possible branches at conditional statements. The paths ρ for which ϕ_ρ is satisfiable are *feasible* and are the only ones that can be executed by the actual program. The solutions to ϕ_ρ exactly characterize the inputs that drive the program through ρ. Assuming that the theorem prover used to check the satisfiability of all formulas ϕ_ρ is sound and complete, this use of static analysis amounts to a kind of symbolic testing.

Unfortunately, this approach does not work whenever the program contains statements involving constraints outside the scope of reasoning of the theorem prover, i.e., statements "that cannot be reasoned about symbolically". This limitation is illustrated by the following example.

```
int obscure(int x, int y) {
  if (x == hash(y)) return -1;   // error
  return 0;                      // ok
}
```

Assume that the function hash cannot be reasoned about symbolically. Formally, this means that it is in general impossible to generate two values for inputs x and y that are guaranteed to satisfy (or violate) the constraint x == hash(y). (For instance, if hash is a hash or cryptographic function, it has been mathematically designed to prevent such reasoning.) In this case, static test generation cannot generate test inputs to drive the execution of the program obscure through either branch of the conditional statement: static test generation is helpless for a program like this.

The practical implication of this simple observation is significant: static test generation as proposed by King 30 years ago and much discussed since then (e.g., see [Mye79, Edv99, BCH$^+$04, VPK04, XMSN05, CS05]) is doomed to perform poorly whenever symbolic execution is not possible. Unfortunately, this is frequent in practice due to complex program statements (pointer manipulations, arithmetic operations, etc.) and calls to operating-system and library functions that are hard or impossible to reason about symbolically with good enough precision.

Dynamic Test Generation is More Powerful

A second approach to test generation is *dynamic test generation* (e.g., [Kor90, GMS00]): it consists of executing the program P, typically starting with some random inputs, gathering symbolic constraints on inputs gathered from predicates in branch statements along the execution, and then using a constraint solver to infer variants of the previous inputs in order to steer the next execution of the program towards an alternative program branch. This process is repeated until a given final statement is reached or a specific program path is executed.

Directed Automated Random Testing [GKS05], or DART for short, is a recent variant of dynamic test generation that blends it with model checking techniques with the goal of systematically executing *all* feasible program paths of a program while detecting various types of errors using run-time checking tools (like Purify, for instance). In DART, each new input vector attempts to force the execution of the program through some new path. By repeating this process, such a *directed search* attempts to force the program to sweep through all its feasible execution paths, in a style similar to *systematic testing* and *dynamic software model checking* [God97]. In practice, DART typically achieves much better coverage than pure random testing (see [GKS05]).

A key observation from [GKS05] is that *imprecision in symbolic execution can be alleviated using concrete values and randomization*: whenever symbolic execution does not know how to generate a constraint for a program statement depending on some inputs, one can always simplify this constraint using the concrete values of those inputs.

Let us illustrate this important point with an example. Consider again the program obscure given above. Even though it is *statically* impossible to generate two values for inputs x and y such that the constraint x == hash(y) is satisfied (or violated), it is easy to generate, for a fixed value of y, a value of x that is equal to hash(y) since the latter is known at runtime. By picking randomly and then fixing the value of y, we can, in the next run, set the value of the other input x either to hash(y) or to something else in order to force the execution of the then or else branches, respectively, of the test in the function obscure. (DART does this automatically [GKS05].)

In summary, static test generation is totally helpless to generate test inputs for the program obscure, while dynamic test generation can easily drive the executions of that same program through all its feasible program paths!

Dynamic test generation can thus be viewed as extending static test generation with additional runtime information, and is thus more general and powerful. Indeed, it can use the same symbolic execution engine *and* use concrete values to simplify constraints outside the scope of the constraint solver. This is why we believe that dynamic test generation is our only hope of one day providing effective, practical test generation tools that are applicable to real-life software. And the purpose of the present paper is to discuss how to make this possible for *large* software applications.

SMART = Scalable DART

Obviously, systematically executing *all* feasible program paths does not scale to large, realistic programs.

This paper addresses this major limitation and proposes to perform dynamic test generation *compositionally*, by adapting known techniques for interprocedural static analysis (e.g., [RHS95]) that have been used to make static analysis scalable to very large programs (e.g., [BPS00, DLS02, HCXE02, CDW04]). Specifically, we introduce a new algorithm, dubbed *SMART* for *Systematic Modular Automated Random Testing*, that extends DART by testing functions in isolation, encoding test results as function summaries expressed using input preconditions and output postconditions, and then re-using those summaries when testing higher-level functions. We show that, for a fixed reasoning capability, our compositional approach to dynamic test generation (SMART) is both sound and complete compared to monolithic dynamic test generation (DART). In other words, SMART can perform dynamic test generation compositionally without any reduction in program path (and hence branch) coverage. We also show that, given a bound on the maximum number of feasible paths in individual program functions, the number of program executions explored by SMART is linear in that bound, while the number of program executions explored by DART can be exponential in that bound. We present examples of C programs and preliminary experimental results that illustrate and validate empirically these properties. To the best of our knowledge, SMART is the *first algorithm for compositional dynamic test generation*. We claim that a SMART search is *necessary* to make the "DART approach" scalable to large programs.

2. The DART Search Algorithm

In this section, we briefly recall the DART search algorithm first introduced in [GKS05], later re-phrased in [SMA05] and (independently) in [CE05]. We present here a simplified version to facilitate the exposition, see [GKS05] for additional details.

Like other forms of dynamic test generation (e.g., [Kor90]), DART consists of running the program P under test both concretely, executing the actual program, and symbolically, calculating constraints on values stored in program variables and expressed in terms of input parameters. These side-by-side executions require the program P to be instrumented at the level of a RAM (Random Access Memory) machine. The *memory* $\mathcal{M}$ is a mapping from memory addresses m to, say, 32-bit words. The notation $+$ for mappings denotes updating; for example, $\mathcal{M}' := \mathcal{M} + [m \mapsto v]$ is the same map as $\mathcal{M}$, except that $\mathcal{M}'(m) = v$. We identify *symbolic variables* by their addresses. Thus in an expression, m denotes either a memory address or the symbolic variable identified by address m, depending on the context.

The program P manipulates the memory through *statements* that are specially tailored abstractions of the machine instructions actually executed. A statement can be a *conditional statement* **c** of the form if (e) then goto ℓ' (where e is an expression over symbolic variables and ℓ' is a statement label), an *assignment statement* **a** of the form $m \leftarrow e$ (where m is a memory address),

abort, corresponding to a program error, or **halt**, corresponding to normal termination. The function *get_next_statement*() specifies the next statement to be executed.

The concrete semantics of the RAM machine instructions of P is reflected in $evaluate\_concrete(e, \mathcal{M})$, which evaluates expression e in context $\mathcal{M}$ and returns a 32-bit value for e. A program P defines a sequence of *input addresses* $\vec{M}_0$, the addresses of the input parameters of P. An *input vector* $\vec{I}$ associates a value to each input parameter and defines the initial value of $\vec{M}_0$ and $\mathcal{M}$.[1]

Let C be the set of conditional statements and A the set of assignment statements in P. A *program execution* w is a finite[2] sequence in $\mathsf{Execs} := (\mathsf{A} \cup \mathsf{C})^*(\textbf{abort} \mid \textbf{halt})$. The concrete semantics of P at the RAM machine level allows us to define for each input vector $\vec{I}$ an execution sequence: the result of executing P on $\vec{I}$ (the details of this semantics is not relevant for our purposes). Let $\mathsf{Execs}(P)$ be the set of such executions generated by all possible $\vec{I}$. By viewing each statement as a node, $\mathsf{Execs}(P)$ forms a tree, called the *execution tree*. Its assignment nodes have one successor; its conditional nodes have one or two successors; and its leaves are labeled **abort** or **halt**. The goal of DART is to explore all paths in the execution tree $\mathsf{Execs}(P)$.

To simplify the following discussion, we assume that we are given a theorem prover that decides a theory T (for instance, including integer linear constraints, pointer constraints, array/string constraints, bit-level operation constraints, etc.). DART maintains a *symbolic memory* $\mathcal{S}$ that maps memory addresses to expressions. Initially, $\mathcal{S}$ is a mapping that maps each $m \in \vec{M}_0$ to itself. Expressions are evaluated symbolically with the function *evaluate_symbolic*$(e, \mathcal{M}, \mathcal{S})$. When an expression falls outside the theory T, *DART simply falls back on the concrete value* of the expression, which is used as the result. In such a case, we also set a flag *complete* to 0, which we use to track completeness. With this evaluation strategy, symbolic variables of expressions in $\mathcal{S}$ are always contained in $\vec{M}_0$.

To carry out a systematic search through the execution tree, our instrumented program is run repeatedly. Each run (except the first) is executed with the help of a record of the conditional statements executed in the previous run. For each conditional, we record a *done* value, which is 0 when only one branch of the conditional has executed in prior runs (with the same history up to the branch point) and is 1 otherwise. This information associated with each conditional statement of the last execution path is stored in a list variable called *stack*, kept between executions. For i, $0 \leq i < |stack|$, $stack[i]$ is thus the record corresponding to the $i+1$th conditional executed.

More precisely, the DART test driver *run_DART* is shown in Figure 1 where the two lines marked by (*) should be *ignored*. This driver combines random testing (the repeat loop) with directed search (the while loop). If the instrumented program throws an exception, then a bug has been found. The completeness flag *complete* holds unless a "bad" situation possibly leading to incompleteness has occurred. Thus, if the directed search terminates—that is, if *directed* of the inner loop no longer holds—then the outer loop also terminates provided the completeness flag still holds. In this case, DART terminates and safely reports that all feasible program paths have been explored. But if the completeness flag has been turned off at some point, then the outer loop continues forever.

The instrumented program itself is described in Figure 2 where the lines marked by (*) should again be ignored for now (ˆ denotes list concatenation). It executes as the original program, but with interleaved gathering of symbolic constraints. At each conditional statement, it also possible to check whether the current execution path matches the one predicted at the end of the previous execution and represented in *stack* passed between runs. How to do this is described in the function *compare_and_update_stack* of [GKS05].

When the original program halts, new input values are generated in *solve_path_constraint*, shown in Figure 3 while ignoring again all the lines marked with (*), to attempt to force the next run to execute the last[3] unexplored branch of a conditional along the stack. If such a branch exists and if the path constraint that may lead to its execution has a solution $\vec{I}'$, this solution is used to update the mapping $\vec{I}$ to be used for the next run; values corresponding to input parameters not involved in the path constraint are preserved (this update is denoted $\vec{I} + \vec{I}'$).

The main property of DART is stated in the following theorem, which formulates (a) soundness (of error founds) and (b) a form of completeness.

THEOREM 1. *[GKS05] Consider a program P as previously defined. (a) If run_DART prints out "Bug found" for P, then there is some input to P that leads to an abort. (b) If run_DART terminates without printing "Bug found," then there is no input that leads to an abort statement in P, and all paths in $\mathsf{Execs}(P)$ have been exercised. (c) Otherwise, run_DART will run forever.*

Proofs of (a) and (c) are immediate. The proof of (b) rests on the assumption that any potential incompleteness in DART's search is (conservatively) detected by setting the flag *complete* to 0.

3. The SMART Search Algorithm

We now present an alternative search algorithm that does not compromise search completeness but is typically much more efficient than the DART search algorithm. The general idea behind this new search algorithm is to perform dynamic test generation *compositionally*, by adapting (dualizing) known techniques for interprocedural static analysis to the context of automated dynamic test generation. Specifically, we introduce a new algorithm, dubbed *SMART* for *Systematic Modular Automated Random Testing*, that tests functions in isolation, collects testing results as function summaries expressed using preconditions on function inputs and postconditions on function outputs, and then re-use those summaries when testing higher-level functions.

We assume we are given a program P that consists of a set of functions. If a function f is part of P, we write $f \in P$. In what follows, we use the generic term of *function* to denote any part of a program P that we want to analyze in isolation and then summarize its observed behaviors. Obviously, any other kinds of program fragments such as program blocks or object methods can be treated as "functions" as done in this paper.

To simplify the presentation, we assume that the functions in P do not perform recursive calls, i.e., that the call-flow graph of P is acyclic. (This restriction can be lifted using dynamic programming techniques to compute function summaries, as is standard in interprocedural static analysis and pushdown system verification [RHS95, ABE+05].) As previously stated, we also assume that all the executions of P terminate. Note that both of these assumptions do not prevent P from possibly having infinitely many executions paths, as is the case if P contains a loop whose number of iterations may depend on some unbounded input.

[1] To simplify the presentation, we assume that $\vec{M}_0$ is the same for all executions of P.

[2] We thus assume that all program executions terminate; in practice, this can be enforced by limiting the number of execution steps.

[3] A depth-first search is used for exposition, but the next branch to be forced could be selected using a different strategy, e.g., randomly or in a breadth-first manner.

3.1 Definition of Summaries

For a given theory T of constraints, a function summary ϕ_f for a function f is defined as a formula of propositional logic whose propositions are constraints expressed in T. ϕ_f can be computed by successive iterations and defined as a disjunction of formulas ϕ_w of the form $\phi_w = pre_w \wedge post_w$, where pre_w is a conjunction of constraints on the inputs of f while $post_w$ is a conjunction of constraints on the outputs of f. ϕ_w can be computed from the path constraint corresponding to the execution path w as will be described shortly. An input to a function f is any address (memory location) that can be read by f in some of its execution, while an output of f is any address that can be written by f in some of its executions and later read by P after f returns.

Preconditions in function summaries are expressed in terms of constraints on function inputs instead of program inputs in order to avoid duplication of identical summaries in equivalent but different calling contexts. For instance, in the following program

```
int is_positive(int x) {
  if (x > 0) return 1;
  return 0;
}
void top(int y, int z) {
  int a,b;
  a = is_positive(y);
  b = is_positive(z);
  if (a && b) then [...]
  [...]
}
```

the summary for the function `is_positive` could be $(x > 0 \wedge ret = 1) \vee (x \leq 0 \wedge ret = 0)$ (if T includes linear arithmetic) where ret denotes the value returned by the function. This summary is expressed in terms of the function input `x`, independently of specific calling contexts which may map `x` to different program inputs like `y` and `z` in this example.[4]

Whenever a constraint on some input cannot be expressed within T, no constraint is generated. For instance, consider the following function `g`:

```
1  int g(int x) {
2    int y;
3    if (x < 0) return 0;
4    y = hash(x);
5    if (y == 100) return 10;
6    if (x > 10) return 1;
7    return 2;
8  }
```

Assuming the constraint `(hash(x)==100)` cannot be expressed in T, the summary ϕ_w of the execution path w corresponding to taking all the else branches at the three conditional statements in function `g` is then $(x \geq 0 \wedge x \leq 10 \wedge ret = 2)$.

A precondition defines an equivalence class of concrete executions. All the concrete executions corresponding to concrete inputs satisfying the same precondition are guaranteed to execute the same program path *only provided that all the constraints along that path are in* T. In the example above, if the path w that takes all the else branches in function `g` was explored with a random concrete value, say, $x = 5$, another value satisfying the same precondition $(x \geq 0 \wedge x \leq 10)$, say $x = 6$ is *not* guaranteed to yield the same program path, because of the presence of the unpredictable conditional statement in line 5 (as $hash(6)$ could very well be 100). The execution of this conditional statement makes a DART search incomplete (the flag *complete* is then set to 0). In that case, all the preconditions in a function summary may no longer be mutually exclusive: a given concrete state may satisfy more than one precondition in a function summary when the function contains conditional statements whose corresponding constraints are outside T.

[4] Remember that symbolic variables are associated with program or function inputs, i.e., memory locations where inputs are being read from. When syntactic program variables uniquely define where those inputs are stored, like variables `x`, `y` and `z` in the above example, we merely write "an input `x`" in the text to simplify the presentation.

3.2 Computing Summaries

Function summaries can be computed by successive iterations, one path at a time. When the execution of the function terminates, the DART-computed path constraint for the current path w in the function can be used to generate a precondition pre_w for that path: pre_w is obtained by simplifying the conjunction of branch conditions on function inputs in the path constraint for w.

If the execution of the function terminates on a return statement, a postcondition $post_w$ can be computed by taking the conjunction of constraints associated with memory locations $m \in Write(f, \vec{I_f}, w)$ written during the execution of f during the last execution w generated from a context (set of inputs) $\vec{I_f}$. Precisely, we have

$$post_w = \bigwedge_{m \in Write(f, \vec{I_f}, w)} (m = \mathit{evaluate\_symbolic}(m, \mathcal{M}, \mathcal{S}))$$

Otherwise, if the function terminates on a `halt` or `abort` statement, we define $post_w = false$ to record this in the summary for possible later use in the calling context, as described later.

A summary for the execution path w in f is then $\phi_w = pre_w \wedge post_w$. The process is repeated for other DART-exercised paths w in f, and the overall summary for f is defined as $\phi_f = \bigvee_w \phi_w$.

By default, the above procedure can always be used to compute function summaries path by path. But more advanced techniques, such as automatically-inferred loop invariants, could also be used (see Section 4). Note that pre_w can always be approximated by $false$ (the strongest precondition) while $post_w$ can always be approximated by $true$ (the weakest postcondition) without compromising the correctness of summaries, and that any technique for generating provably correct weaker preconditions or stronger postconditions can be used to improve precision.

Given the call-flow graph G_P of a program P (which we have previously assumed to be acyclic) and a topological sort of the functions in G_P computed starting from the top-level function of the program, function summaries can then be computed in either a *bottom-up* or *top-down* strategy.

With a bottom-up strategy, one starts testing functions at the deepest level in G_P, one computes summaries for those, and then moves up the topological sort to functions one-level up while reusing the summaries for the functions below (as described in the next subsection), and so on up to the top-level function of the program. While the bottom-up strategy is conceptually the easiest to understand, it suffers from two major limitations that make its implementation problematic in the context of compositional dynamic test generation.

First, testing lower-level functions in isolation for all possible contexts (i.e., for all possible input values) is likely to trigger unrealistic behaviors that may not happen in the specific contexts in which the function can actually be called by higher-level program functions; this analysis can be prohibitively expensive and will likely generate an unnecessarily large number of spurious summaries that will never be used subsequently. Thus, *too many* summaries are computed.

Second, because of the inherent limitation of symbolic execution to reason about constraints outside the given theory T, summaries computed in bottom-up fashion may be incomplete in pres-

```
run () =
     complete = 1
(*)  summary = $[f \mapsto \emptyset \mid f \in P]$ // Set of summaries
     repeat
         stack = $\langle\rangle$; $\vec{I}$ = [] ; directed = 1
(*)      context_stack = $\langle(\_, \_, 0)\rangle$ // Stack of contexts
         while (directed) do
             try (directed, stack, $\vec{I}$) =
                 instrumented_program(stack, $\vec{I}$)
             catch any exception $\rightarrow$
                 print "Bug found"; exit()
     until complete
```

Figure 1. run_DART and (*) run_SMART test drivers

ence of statements involving constraints outside T. For instance, in the case of function g presented in Section 3.1, analyzing g in isolation using DART techniques will probably *not* be able to exercise the then branch of the conditional statement on line 5, i.e., to randomly find a value of x such that `hash(x) == 100`. However, in its actual calling contexts within the program P, it is possible that the function g is often called with values for x that satisfy this constraint. In this case, *too few* summaries are pre-computed, and it is necessary to compute later in the search a summary for the case where `hash(x) == 100` is satisfied.

To avoid these two limitations, we recommend and adopt a top-down strategy for computing summaries on a demand-driven basis. A complete algorithm for doing this is described next.

3.3 Algorithm

A top-down SMART search algorithm is presented in Figures 1, 2 and 3. The pseudo-code for SMART is similar to the one for DART with the exception of the new additional lines marked by (*). Indeed, SMART strictly generalizes DART and reduces to it in the case of programs consisting of a single function.

A SMART search performs dynamic test generation compositionally, using function summaries as defined previously. Those summaries are dynamically computed in a top-down manner through the call-flow graph G_P of P. Starting from the top-level function, one executes the program (initially on some random inputs) until one reaches a first function f whose execution terminates on a return or `halt` statement. One then backtracks inside f as much as possible using DART, computing summaries for that function and each of those DART-triggered executions. When this search (backtracking) in f is over, one then resumes the original execution where f was called, this time treating f essentially as a black-box, i.e., without analyzing it and re-using its previously computed summary instead. The search proceeds similarly, with the next backtracking point being in some lower-level function, if any, called after f returns, or in the function g that called f otherwise, or some other higher-level function that called g if the search in g is itself over. This search order is thus different from DART's search order.

A SMART search starts by executing the procedure *run_SMART* described in Figure 1. The only differences with the procedure *run_DART* is the initialization of a set of summaries and of a context stack that records the sequence of calling contexts for which summaries still need to be computed along the current execution, and is also used to resume execution in a previous context.

The main functionality of SMART is presented in Figure 2. The key difference with DART is that function calls and returns are now instrumented to trigger and organize the computation of function summaries. Whenever a function f is called, a SMART *instrumented_program* checks whether a summary for f is already available for the current calling context $\vec{I_f}$. This is done by check-

```
     instrumented_program(stack, $\vec{I}$) =
         // Random initialization of uninitialized input parameters in $\vec{M_0}$
         for each input x with $\vec{I}$[x] undefined do $\vec{I}$[x] = random()
         Initialize memory $\mathcal{M}$ from $\vec{M_0}$ and $\vec{I}$
         // Set up symbolic memory and prepare execution
         $\mathcal{S} = [m \mapsto m \mid m \in \vec{M_0}]$
         $k$ = 0 // Number of conditionals executed
(*)      backtracking = 1 // By default, backtrack at all branch points
         // Now invoke $P$ intertwined with symbolic calculations
         s = get_next_statement()
         while (s $\notin$ {abort, halt}) do
             match (s)
                 case ($m \leftarrow e$):
                     $\mathcal{S}$= $\mathcal{S}$ + [$m \mapsto$ evaluate_symbolic($e, \mathcal{M}, \mathcal{S}$)]
                     $v$ = evaluate_concrete($e, \mathcal{M}$)
                     $\mathcal{M}$ = $\mathcal{M}$ + [$m \mapsto v$]
                 case (if ($e$) then goto $\ell$):
                     $b$ = evaluate_concrete($e, \mathcal{M}$)
                     $c$ = evaluate_symbolic($e, \mathcal{M}, \mathcal{S}$)
(*)                  if backtracking then
                         if $b$ then
                             path_constraint = path_constraint ^ $\langle c\rangle$
                         else
                             path_constraint = path_constraint ^ $\langle neg(c)\rangle$
                         if ($k = |stack|$) then stack = stack ^ $\langle 0\rangle$
                         $k = k + 1$
(*)              case ($f$ : call):     // call of function $f$ with context $\vec{I_f}$
(*)                  if backtracking then
(*)                      if ($\vec{I_f} \in summary(f)$) then
(*)                          // We have a summary for $f$ in context $\vec{I_f}$
(*)                          path_constraint = path_constraint ^ $\langle summary(f)\rangle$
(*)                          // Execute $f$ without backtracking until it returns
(*)                          backtracking = 0
(*)                          if ($k = |stack|$) then stack = stack ^ $\langle 1\rangle$
(*)                          $k = k + 1$
(*)                      else // Compute a summary for $f$ in context $\vec{I_f}$
(*)                          Push $(f, \vec{I}, k)$ onto context_stack
(*)              case ($f$ : return):     // return of function $f$
(*)                  if backtracking then
(*)                      // Stop the search in $f$
(*)                      // Generate a summary for the current path
(*)                      add_to_summary($f$,path_constraint)
(*)                      return solve_path_constr($k$,path_constraint,stack)
(*)                  else
(*)                      if (Top(context_stack) = $(f, \_, \_)$) then
(*)                          backtracking = 1
(*)                          // Extend the set of inputs by the return values of $f$
(*)                          $\mathcal{M} = \mathcal{M} + [m \mapsto m \mid m \in post(summary(f))]$
             s =get_next_statement()
         od    // End of while loop
         if (s==abort) then
             raise an exception
         else // s==halt
(*)          if backtracking then
(*)              $(f, \_, \_) =$ Top(context_stack)
(*)              add_to_summary($f$,path_constraint)
             return solve_path_constr($k$,path_constraint,stack)
```

Figure 2. DART and (*) SMART instrumented_program

```
solve_path_constr($k$,path_constraint,stack) =
    $j = k - 1$; $k_f = 0$
(*) $(f, \vec{I}, k_f)$ = Top(context_stack)
    while ($j \geq k_f$) do
        if (stack[$j$] = 0) then
            $path\_constraint[j]$ = neg($path\_constraint[j]$)
            if ($path\_constraint[0,\ldots,j]$ has a solution $\vec{I}'$) then
                stack[$j$] = 1
                return $(1, stack[0..j], \vec{I} + \vec{I}')$
            else $j = j - 1$
        else $j = j - 1$
    od
(*) if ($k_f > 0$) then
(*)     Pop $(f, \vec{I}, k_f)$ from context_stack
(*)     return $(1, stack[0..(k_f - 1)], \vec{I})$
    return (0, _, _) // This directed search is over
```

Figure 3. DART and (*) SMART solve_path_constr

ing whether the current *symbolic*[5] calling context implies the disjunction of preconditions currently recorded in the summary for f.[6]

If so, this summary is added to the current path constraint, and the execution proceeds by turning backtracking off in f and any function below it in the call-flow graph of P. The latter is done through the use of a boolean flag *backtracking*. Backtracking is resumed later in the current execution path when f returns: this is done in the else branch of the conditional statement included in the `return` case, where the set of inputs (in the function calling f) is also extended with the set of return values appearing in the set $post(summary(f))$ of postconditions included in the summary $summary(f)$ currently available for f.

If no summary is available for the current calling context $\vec{I}_f$, the current input assignment $\vec{I}$ is saved by pushing it onto the context stack, and the algorithm will compute a summary for $\vec{I}_f$ by continuing the search deeper in the called function f. When backtracking is on and the inner-most function terminates either on a return statement or a `halt` statement, *add_to_summary*(f,*path_constraint*) computes a summary for f and the last path executed as discussed in Section 3.2. Note that a function summary for f includes in itself summaries of lower-level functions possibly called by f itself.

After computing a summary for the current function and execution path, *solve_path_constr*, presented in Figure 3, is called to determine where the algorithm should backtrack next. When backtracking in a specific innermost function f is over, the search resumes from the last input assignment $\vec{I}$ saved in the context stack.

3.4 Correctness

The correctness of the SMART search algorithm is defined with respect to the DART search algorithm, thus independently of a specific theory T representing the reasoning capability of symbolic execution. Specifically, we can prove that, for any program P containing exclusively statements $st(P)$ whose corresponding constraints are in a given decidable theory T (denoted $st(P) \subset T$), the SMART search algorithm provides *exactly the same program path coverage* as the DART search algorithm. Thus, for those programs P, *every feasible path that is exercised by DART is also "explored", albeit compositionally, by SMART*; and conversely, every compositional execution considered by SMART is guaranteed to correspond to a concrete full execution path. Formally, we have the following.

[5] Not "concrete" as incorrectly stated in the POPL'07 Proceedings version.

[6] Checking later that the output values of f for that run satisfy the corresponding postcondition in the summary is not mandatory for correctness but can increase precision and hence coverage.

THEOREM 2. [7] *(Relative Soundness and Completeness) Given any program P and theory T such that $st(P) \subset T$, run_SMART terminates without printing "Bug found"* if and only if *run_DART terminates without printing "Bug found".*

In practice, programs P typically contain statements corresponding to constraints outside T (whatever T is). The SMART and DART searches may then behave differently because their search order vary, and calls to the function random() to initialize undefined inputs may return different values, hence exercising the code randomly differently. Nevertheless, a corollary of the previous theorem is that the SMART search algorithm is *functionally equivalent* to DART, in the sense that it still satisfies the conditions identified in Theorem 1 characterizing the correctness of the DART search algorithm (and of its various implementations [GKS05, CE05, SMA05, YST$^+$06]). Formally, we can prove the following.

THEOREM 3. *Consider a program P as previously defined. (a) If run_SMART prints out "Bug found" for P, then there is some input to P that leads to an abort. (b) If run_SMART terminates without printing "Bug found," then there is no input that leads to an abort statement in P. (c) Otherwise, run_SMART will run forever.*

In summary, SMART is functionally equivalent to DART and, typically, whatever test inputs DART can generate, SMART can too, although possibly much more efficiently. How much more efficient (hence scalable) can SMART be compared to DART? This question is addressed next.

3.5 Complexity

Let b be a bound on the maximum number of distinct execution paths that can be contained in any function f of the program P. If a function f does not contain any loop, such a bound is guaranteed to exist, although it can be exponential in the number of statements in the code describing f. If f contains loops whose number of iterations may depend on an unbounded input, the number of execution paths in f could be infinite, and such a bound b may not exist. In practice, a bound b can always be enforced by simply limiting the number of execution paths explored in a function, i.e., by limiting the size of summaries; this heuristics has been shown to work well in the context of interprocedural static analysis (e.g., see [BPS00]).

Given such a bound b, it is easy to see that the number of execution paths considered by a SMART search (while the flag *directed* is kept to 1) will be at most nb, where n is the number of functions f in P, and is therefore *linear* in nb. In contrast, the number of execution paths considered by a DART search (while the flag *directed* is kept to 1) can be *exponential* in nb, as DART does not exploit program hierarchy and treats a program as a single, "flat" function. This reduction in the number of explored paths from exponential to polynomial in b is also observed with compositional verification algorithms for hierarchical finite-state machines [AY98].

Although SMART can avoid systematically executing all the possibly exponentially many feasible program paths in P, it does require the use of formulas ϕ_f representing function summaries which can be of size linear in b, and the use of theorem proving techniques to check satisfiability of those formulas, with decision procedures which can, in the worst case, be exponential in the size of those formulas, i.e., exponential in b. However, while DART can be viewed as always trying to systematically execute all possible execution paths, i.e., *all* possible disjuncts in $\phi_f = \bigvee_w \phi_w$, SMART will try to check the satisfiability of ϕ_f in conjunction with additional constraints generated from a calling context of f, and hence try to find just *one* way to satisfy the resulting formula using a logic constraint solver. This key point is illustrated next.

[7] Edited for completeness and clarity from the POPL'07 Proceedings.

4. Example, Case Study, Discussion

A Simple Example

Consider the function `locate` whose code is as follows:

```
1  // locate index of first character c
2   // in null-terminated string s
3  int locate(char *s, int c) {
4    int i=0;
5    while (s[i] != c) {
6      if (s[i] == 0) return -1;
7      i++;
8    }
9    return i;
10 }
```

Given a string `s` of maximum size n (i.e, `s[n]` is always zero), there are at most $2n$ distinct execution paths for that function if `c` is non-zero (and at most n if `c` is zero). Those $2n$ paths can be denoted by the regular expression: $\langle$(line5:then; line6:else)^i (line5:else | (line5:then; line6:then))$\rangle$ for $0 \leq i \leq (n-1)$. There are $n+1$ possible return values: -1 (for the n paths $\langle$(line5:then; line6:else)^i (line5:then; line6:then)$\rangle$ for $0 \leq i \leq (n-1)$), and $0, 1, \ldots, (n-1)$, each returned by the path $\langle$(line5:then; line6:else)^i line5:else$\rangle$ where i is equal to the return value.

Now, consider the function `top` which calls the function `locate`:

```
11  int top(char *input) {
      // assume input is null-terminated
12    int z;
13    z = locate(input,'a');
14    if (z == -1) return -1;          // error code
15    if (input[z+1] != ':') return 1; // success
16    return 0;                        // failure
17  }
```

In the function `top`, there are 3 possible execution paths: $\langle$line14:then$\rangle$, $\langle$line14:else; line15:then$\rangle$ and $\langle$line14:else; line15:else$\rangle$.

Following the call to `locate`, the outcome of the test at line 14 is completely determined by the return value from function `locate` stored in `z`. In contrast, the test at line 15 constraints the next element `input[z+1]` in the string `input` and its outcome depends on the value stored at that address. That input value could either be equal to `':'` or not, except for `input[`n`]` which we assumed to be zero. Therefore, for the whole program P composed of the two functions `top` and `locate`, there are $3n-1$ possible execution paths: n executions terminate after the then branch of line 14, n executions terminate after the then branch of line 15, and $n-1$ executions terminate in line 16. Thus, the number of feasible paths in P is (roughly) the *product* of the number of paths in its functions `locate` and `top`.

A DART search attempts to systematically execute all possible execution paths and would thus perform $3n-1$ runs for this program. In contrast, a SMART search will systematically execute all possible execution paths of the function `locate` and `top` *separately*. Precisely, a SMART search computing function summaries as described in Section 3.2 would compute $2n$ path summaries for function `locate`, whose function summary ϕ_f would then be of the form

$$\begin{aligned}\phi_f \quad = \quad & (s[0] = c \wedge ret = 0) \\ & \vee (s[0] \neq c \wedge s[0] = 0 \wedge ret = -1) \\ & \vee (s[0] \neq c \wedge s[0] \neq 0 \wedge s[1] = c \wedge ret = 1) \\ & etc.\end{aligned}$$

Then, the SMART search would explore the feasibility of the 3 paths of the function `top` using ϕ_f to summarize function `locate`.

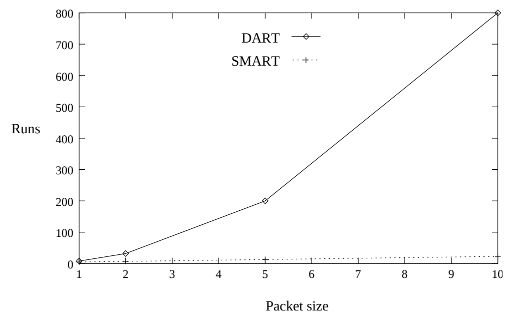

Figure 4. Experimental comparison between DART and SMART

For this example, SMART would then perform $2n+3$ runs, i.e., the *sum* of the number of paths in its functions `locate` and `top`.

Observe how the address `z+1` is defined relative to `z` and that its absolute value "does not matter" (as long as $z+1 \neq n$) when proving the satisfiability of the constraint generated from the test `input[z+1] != ':'` and of its negation. This is captured by the SMART algorithm, which will not attempt to try *all* possible ways to satisfy/violate these constraints (as DART would), but will only find *one* way to satisfy those. This observation explains intuitively the significant speed-up that SMART can provide compared to DART, while guaranteeing the same path (and hence branch) coverage (100% branch coverage is achieved in this example).

Case Study

We have developed an implementation of the SMART search algorithm for the C programming language, extending the DART implementation described in [GKS05]. We report here preliminary experiments comparing the efficiency of DART and SMART on the oSIP example discussed in [GKS05]. oSIP is an open-source C library implementing the SIP protocol and consisting of about 30,000 lines of C code. SIP messages are transmitted as ASCII strings and a large part of the oSIP code is dedicated to parsing SIP messages.

Figure 4 presents the number of runs needed by DART and SMART to fully explore all the feasible program paths in a subset of the oSIP parser code. Experiments were performed for several, small packet sizes. Runtime is linear in the number of runs for those experiments. As is clear from Figure 4, SMART can fully cover all the feasible program paths of this example much more efficiently than DART. In fact, for this example, the SMART search is *optimal* in the sense that its number of runs (and runtime) grows in a linear way with the size of the input packet.

Discussion

Another way to limit the "path explosion" problem in a DART search is simply to allow backtracking only at branches of conditional statements that have never been executed so far. If B denotes the number of conditional statements in a program P, the number of execution paths (runs) explored by such a "*branch-coverage-driven*" DART search is trivially bounded by $2B$, i.e., is linear in the program size. The drawback of this naive solution is obviously that *full feasible path coverage* is no longer guaranteed, even for programs containing only statements with constraints in T. This, in turn, typically reduces overall branch coverage itself, and thus chances of finding bugs. In contrast, SMART reduces the computational complexity of DART *without sacrificing* full path coverage and hence provably without reducing branch coverage.

In the presence of loops, loop invariants could be used to generate more general and compact function summaries than those generated by the path-by-path procedure for computing summaries presented in Section 3.2. For instance, considering again the function `locate`, a more compact and general function summary is

$\phi_f = ((\exists i \geq 0 : s[i] = c \wedge (\forall j < i : (s[j] \neq c) \wedge (s[j] \neq 0))) \wedge ret = i) \vee ((\exists i \geq 0 : s[i] = 0 \wedge (\forall j < i : s[j] \neq c)) \wedge ret = -1)$, which is independent of any maximum size n for the string `s`. Concrete values known at runtime could be used to detect "partial" loop invariants, i.e., simplified loop invariants that are valid only when some input variables are fixed.

5. Conclusion

DART [GKS05], and closely related work (e.g., [SMA05, CE05, YST$^+$06]), is a promising new approach to automatically generate tests from program analysis. Actually, DART can be viewed [GK05] as one way of combining static (interface extraction, symbolic execution) and dynamic (testing, run-time checking) program analysis with model-checking techniques (systematic state-space exploration) in order to address one of the main limitations of previous dynamic, concrete-execution-based software model checkers (such as VeriSoft, JavaPathFinder and CMC, among others), namely their inability to automatically deal with input data nondeterminism.

But DART suffers from two major limitations. First, its effectiveness critically depends on the symbolic reasoning capability T available. Whenever symbolic execution is not possible, concrete values can be used to simplify constraints and carry on with a simplified, partial symbolic execution. Randomization can also help by suggesting concrete values whenever automated reasoning is impossible or difficult. Still, it is currently unknown whether dynamic test generation is really that superior to static test generation, that is, how effective using concrete values and randomization help symbolic execution for testing purposes in practice. More experiments with various kinds of examples are needed to determine this.

Second, DART suffers from the "path explosion" problem: systematically executing *all* feasible program paths is typically prohibitively expensive for large programs. This paper addresses this second limitation in a drastic way, by performing dynamic test generation compositionally and eliminating path explosion due to interprocedural program paths (i.e., paths across function boundaries) without sacrificing overall path or branch coverage. A SMART search can be viewed as exploring the set of feasible whole program paths *symbolically*, i.e., by exploring simultaneously *sets* of such paths, instead of executing those one by one.

Our approach adapts known techniques for interprocedural static analysis to the context of dynamic test generation. While implementations of interprocedural static analysis are typically both incomplete (may miss bugs) and unsound (may generate false alarms) with respect to falsification [GK05], our compositional dynamic test generation is performed in such a way to preserve the soundness of bugs [God05]: any error path found is guaranteed to be sound, as every *compositional symbolic execution* is grounded, by design, into some concrete execution. The only imprecision in our approach is incompleteness with respect to falsification: we may miss bugs by failing to exercize some executable program paths and branches.

The idea of compositional dynamic test generation was already suggested in [GK05]; the motivation of the present paper is to investigate this idea in detail. Other recent related work includes [CG06], which proposes and evaluates several heuristics based on light-weight static analysis of function interfaces to partition large software applications into groups of functions, called units. Those units can then be tested in isolation without generating too many false alarms caused by unrealistic inputs being injected at interfaces between units. In contrast with the present work, no summarization of unit testing, nor any global analysis is ever performed in [CG06]. Both types of techniques can actually be viewed as complementary. We refer the reader to [GKS05] for a detailed discussion of other automated test generation techniques and tools, and to [GK05] for a discussion of other possible DART extensions.

References

[ABE$^+$05] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of Recursive State Machines. *TOPLAS*, 27(4):786–818, July 2005.

[AY98] R. Alur and M. Yannakakis. Model Checking of Hierarchical State Machines. In *FSE'98*.

[BCH$^+$04] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating Tests from Counterexamples. In *ICSE'2004*.

[BKM02] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA'2002*.

[BPS00] W.R. Bush, J.D. Pincus, and D.J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7):775–802, 2000.

[CDW04] H. Chen, D. Dean, and D. Wagner. Model Checking One Million Lines of C Code. In *NDSS'04*.

[CE05] C. Cadar and D. Engler. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *SPIN'2005*.

[CG06] A. Chakrabarti and P. Godefroid. Software Partitioning for Effective Automated Unit Testing. In *EMSOFT'2006*.

[CS05] C. Csallner and Y. Smaragdakis. Check'n Crash: Combining Static Checking and Testing. In *ICSE'2005*.

[DLS02] M. Das, S. Lerner, and M. Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. In *PLDI'2002*.

[Edv99] J. Edvardsson. A Survey on Automatic Test Data Generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering*, pages 21–28, Linkoping, October 1999.

[GK05] P. Godefroid and N. Klarlund. Software Model Checking: Searching for Computations in the Abstract or the Concrete (Invited Paper). In *IFM'2005*.

[GKS05] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI'2005*.

[GMS00] N. Gupta, A. P. Mathur, and M. L. Soffa. Generating Test Data for Branch Coverage. In *ASE'2000*.

[God97] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *POPL'97*.

[God05] P. Godefroid. The Soundness of Bugs is What Matters (Position Paper). In *BUGS'2005*.

[HCXE02] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A System and Language for Building System-Specific Static Analyses. In *PLDI'2002*.

[Kin76] J. C. King. Symbolic Execution and Program Testing. *Journal of the ACM*, 19(7):385–394, 1976.

[Kor90] B. Korel. A Dynamic Approach of Test Data Generation. In *IEEE Conference on Software Maintenance*, 1990.

[Mye79] G. J. Myers. *The Art of Software Testing*. Wiley, 1979.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL'95*.

[SMA05] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *FSE'2005*.

[VPK04] W. Visser, C. Pasareanu, and S. Khurshid. Test Input Generation with Java PathFinder. In *ISSTA'2004*.

[XMSN05] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution. In *TACAS'2005*.

[YST$^+$06] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically Generating Malicious Disks using Symbolic Execution. In *Proceedings of IEEE Security and Privacy'2006*, Oakland, 2006.