Compositional Dynamic Test Generation

(Preliminary Version – July 14, 2006)

Patrice Godefroid

Bell Laboratories, Lucent Technologies

god@bell-labs.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 exploring *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, Program Analysis, Scalability, Compositional Analysis, Program Verification

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profi t or commercial advantage and that copies bear this notice and the full citation on the fi rst page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specifi c permission and/or a fee.

WXYZ '05 date, City.

Copyright © 2006 ACM [to be supplied] . . . $5.00.

1. Introduction

Given a program P, say a C program with a million lines of code, with a set of input parameters I, 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?

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, industrial-strength 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]).

Progress on practical test-generation tools based on automatic program analysis has been arguably slow, until recently where there has been a renewed interest for this problem (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 (e.g., [Kin76]) consists of analyzing the 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*. Unfortunately, the applicability of this approach is severely limited when the program P contains statements whose behavior cannot be predicted statically with symbolic execution.

Dynamic test generation (e.g., [Kor90]) consists of executing the program P, typically starting with some random inputs, gathering symbolic constraints on inputs 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 a specific program statement. This process is repeated until a given final statement is reached or a specific program path is executed. As observed in [GKS05] and explained in detail in the Section 2, given some fixed symbolic execution and constraint solving capabilities, dynamic test generation generalizes static test generation, and can alleviate the inherent limitations of symbolic execution by using concrete values, available at runtime, to simplify symbolic expressions that are beyond the scope of the given constraint solver. In practice, dynamic test generation seems more effective than static test gen-

eration as it can sometimes easily handle programs for which static test generation is totally helpless (see Section 2).

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). Unfortunately, systematically exploring *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, HCXE02, DLS02, 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.

This paper is organized as follows. In Section 2, we compare in detail the static and dynamic approaches to test generation. Then, we recall the DART search algorithm of [GKS05] in Section 3. We then introduce the SMART search algorithm and discusses its correctness and complexity. Section 5 discusses examples of programs and experimental results comparing the performance of DART and SMART. We conclude with some additional remarks in Section 6 and by discussing other related work in Section 7.

2. Background

2.1 Problem Definition

Consider a sequential, deterministic program P consisting of a set of program statements S (assignments, tests, loops, etc.). Given an input I, P computes a unique value $v = P(I)$ (also called *output*) where $P(I)$ denotes the execution of P induced by I. The problem of *test input generation* is defined as follows.

DEFINITION 1 (Test Input Generation). *Given a statement $s \in S$ of a program P, compute an input I such that the execution $P(I)$ of P induced by I reaches statement s.*

A variant of the previous definition is as follows.

DEFINITION 2 (Test Suite Generation). *Given a program P with a set S of statements, compute a set of inputs I_S such that, for all statement s in S, there is an input I in I_S such that the execution $P(I)$ of P induced by I reaches statement s.*

It is interesting to observe how effective random testing can be in solving each of these two problems. *Random testing* simply means choosing an input randomly. It is a simple and well-known technique (e.g., [BM83]), which in practice can sometimes be remarkably effective at finding software bugs [FM00].

PROPOSITION 1. *If $\mathcal{P}_n(S)$ denotes the probability of reaching* any one *statement in S with n random inputs, we have*

$$S_1 \subseteq S_2 \Rightarrow \mathcal{P}_n(S_1) \leq \mathcal{P}_n(S_2)$$

(where $\Rightarrow$ denotes logical implication).

This proposition implies that random testing is in general more likely to solve the second problem of test suite generation than the first problem of test input generation.

Still, the power of random testing is limited to address the latter problem.

PROPOSITION 2. *If $\mathcal{P}_n^{all}(S)$ denotes the probability of reaching* all *the statements in S with n random inputs, we have*

$$S_1 \subseteq S_2 \Rightarrow \mathcal{P}_n^{all}(S_1) \geq \mathcal{P}_n^{all}(S_2)$$

Intuitively, the larger the set S of statements, the less likely random testing will be able to cover them all.

Although obviously $\mathcal{P}_n^{all}(S) < \mathcal{P}_m^{all}(S)$ if $n < m$, increasing n does not necessarily increase $\mathcal{P}_n^{all}$ much in practice. For instance, the `then` branch of the conditional statement "`if (x==10) then ...`" has only one chance to be exercised out of 2^{32} if `x` is a 32-bit integer program input that is randomly initialized. In practice, this means that random testing has virtually no chance to exercise the `then` branch of that conditional statement. The limitations of random testing are well-known (e.g., [OH96]).

In what follows, *test generation* will denote both problems unless otherwise specified.

2.2 Static Test Generation is Often Ineffective

The first approach to test generation was proposed about 30 years ago [Kin76]. It consists of analyzing the program P *statically*, by exclusively using symbolic execution techniques to try to compute appropriate inputs I to drive P along specific execution paths or branches, *without ever executing the program*.

More precisely, the idea is to symbolically explore the tree of all computations a 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.

```
void 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)`. Note that, if `hash` is

a hash or cryptographic function, it has been mathematically designed specifically so that such reasoning is impossible.

In this case, static test generation cannot generate test inputs to drive the execution of this program through either branch of the conditional statement in line 5: static test generation is totally 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 discussed in many papers since then (e.g., see [Mye79, Edv99, BCH$^+$04, VPK04, XMSN05, CS05]) is doomed to perform poorly whenever symbolic execution is not possible.

The above limitation would not be severe if conditional statements beyond the scope of theorem provers, like `x == hash(y)`, were rare in practice. Unfortunately, such statements are extremely frequent in systems code written in programming languages like C and C++. Indeed, those programs typically involve complex program statements (pointer manipulations, arithmetic and bit-vector operations, etc.), and many calls to operating-system and library functions that are hard or impossible to reason about symbolically with good enough precision.

2.3 Dynamic Test Generation is More Powerful

A second approach to test generation is *dynamic test generation* (e.g., [Kor90, GMS00]): the program is executed concretely, possibly on a first input chosen at random, and symbolic constraints are gathered along the execution. Then, a constraint solver is used to infer variants of the first input, and the process is repeated until a given statement is reached. This is the classic definition of "dynamic test generation" which addresses the test input generation problem defined in Section 2.1, and has been studied extensively in the testing research literature (e.g., [Kor90, GMS00]).

Recently, dynamic test generation has been adapted to also deal with the other problem of test suite generation. DART [GKS05] and subsequent related papers [SMA05, CE05, YST$^+$06] address this second problem for the set S of all program statements. Starting with a random input, a DART-instrumented program calculates during each execution an input vector for the next execution. This vector contains values that are the solution of symbolic constraints gathered from predicates in branch statements during the previous execution. The new input vector attempts to force the execution of the program through a 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].

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 handle a program statement involving some input variables, one can always simplify those constraints by using the concrete values of those inputs. Let us illustrate this important point with an example.

Consider again the example of the program `obscure` presented in Section 2.2. 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 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 claim dynamic test generation provides the only hope of one day providing effective, practical test generation tools that are applicable to real-life software. And this is why we believe the topic addressed in this paper is so central to reaching that goal for *large* software applications.

3. The DART Search Algorithm

In this section, we recall the DART search algorithm first introduced in [GKS05], and later re-implemented in [SMA05] and (independently) in [CE05]. We present here a simplified version to facilitate the exposition. The reader is referred to [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 at memory locations 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. A *symbolic expression*, or just expression, e can be of the form m, c (a constant), $*(e', e'')$ (a dyadic term denoting multiplication), $\leq (e', e'')$ (a term denoting comparison), $\neg(e')$ (a monadic term denoting negation), $*e'$ (a monadic term denoting pointer dereference), etc. Thus, the symbolic variables of an expression e are the set of addresses m that occur in it. Expressions have no side-effects.

The program P manipulates the memory through *statements* that are specially tailored abstractions of the machine instructions actually executed. There is a set of numbers that denote instruction addresses, that is, statement labels. If ℓ is the address of a statement (other than **abort** or **halt**), then $\ell + 1$ is guaranteed to also be an address of a statement. The initial address is ℓ_0. 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 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. Additionally, the function *statement_at*$(\ell, \mathcal{M})$ specifies the next statement to be executed. For an assignment statement, this function calculates, possibly involving address arithmetic, the address m of the left-hand side, where the result is to be stored; in particular, indirect addressing, e.g., stemming from pointers, is resolved at runtime to a corresponding absolute address.[1]

A program P defines a sequence of *input addresses* $\vec{M}_0$, the addresses of the input parameters of P. An *input vector* $\vec{I}$, which

[1] We do this to simplify the exposition; left-hand sides could be made symbolic as well.

```
evaluate_symbolic (e, M, S) =
  match e:
    case m:    //the symbolic variable named m
      if (m ∈ domainS) then return S(m)
      else return M(m)
    case op(param-list):    //some operation
      if (op(param-list) ∈ T) then
        return op(param-list)    //this constraint is in theory T
      else    // otherwise fall-back on concrete value
        complete = 0
        return evaluate_concrete(e, M)
```

Figure 1. Symbolic evaluation

```
run_DART () =
  complete = 1
  repeat
    stack = ⟨⟩; I⃗ = [] ; directed = 1
    while (directed) do
      try (directed, stack, I⃗) =
        instrumented_program(stack, I⃗)
      catch any exception →
        print "Bug found"
        exit()
  until complete
```

Figure 2. Test driver

associates a value to each input parameter, defines the initial value of $\vec{M}_0$ and hence $\mathcal{M}$.[2]

Let $\mathbf{C}$ be the set of conditional statements and $\mathbf{A}$ the set of assignment statements in P. A *program execution* w is a finite[3] sequence in $\mathbf{Execs} := (\mathbf{A} \cup \mathbf{C})^*(\mathbf{abort} \mid \mathbf{halt})$. We prefer to view w as being of the form $\alpha_1 \mathbf{c}_1 \alpha_2 \mathbf{c}_2 \ldots \mathbf{c}_k \alpha_{k+1} \mathbf{s}$, where $\alpha_i \in \mathbf{A}^*$ (for $1 \leq i \leq k+1$), $\mathbf{c}_i \in \mathbf{C}$ (for $1 \leq i \leq k$), and $\mathbf{s} \in \{\mathbf{abort}, \mathbf{halt}\}$. 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 $\mathbf{Execs}(P)$ be the set of such executions generated by all possible $\vec{I}$. By viewing each statement as a node, $\mathbf{Execs}(P)$ 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 $\mathbf{Execs}(P)$.

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.). This will allow us to explain how we handle the transition from constraints within the theory T to those that are outside.

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 as described in Figure 1. 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 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 in a file between executions. For i, $0 \leq i < |stack|$, $stack[i]$ is thus the record corresponding to the $i+1$th conditional executed.

More precisely, our test driver *run_DART* is shown in Figure 2. 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 the completeness flag has been turned off at some point, then the outer loop continues forever (modulo resource constraints not shown here).

```
instrumented_program(stack, I⃗) =
  // Random initialization of uninitialized input parameters in M⃗0
  for each input x with I⃗[x] undefined do I⃗[x] = random()
  Initialize memory M from M⃗0 and I⃗
  // Set up symbolic memory and prepare execution
  S = [m ↦ m | m ∈ M⃗0]
  ℓ = ℓ0 // Initial program counter in P
  k = 0 // Number of conditionals executed
  // Now invoke P intertwined with symbolic calculations
  s = statement_at(ℓ,M)
  while (s ∉ {abort, halt}) do
    match (s)
      case (m ← e):
        S= S + [m ↦ evaluate_symbolic(e, M, S)]
        v = evaluate_concrete(e, M)
        M = M + [m ↦ v]; ℓ = ℓ + 1
      case (if (e) then goto ℓ′):
        b = evaluate_concrete(e, M)
        c = evaluate_symbolic(e, M, S)
        if b then
          path_constraint = path_constraint ˆ ⟨c⟩
          ℓ = ℓ′
        else
          path_constraint = path_constraint ˆ ⟨neg(c)⟩
          ℓ = ℓ + 1
        if (k = |stack|) then stack = stack ˆ ⟨0⟩
        k= k + 1
    s =statement_at(ℓ,M)    // End of while loop
  if (s==abort) then
    raise an exception
  else // s==halt
    return solve_path_constraint(k,path_constraint,stack)
```

Figure 3. Instrumented_program

The instrumented program itself is described in Figure 3 (where ˆ denotes list concatenation). It executes as the original program,

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

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

```
solve_path_constraint(k,path_constraint,stack) =
  j = k − 1
  while (j ≥ 0)
    if (stack[j] = 0) then
      path_constraint[j] = neg(path_constraint[j])
      if (path_constraint[0, ... , j] has a solution I') then
        stack[j] = 1
        return (1, stack[0..j], I + I')
      else j = j − 1
    else j = j − 1
  return (0, _, _) // This directed search is over
```

Figure 4. Solve_path_constraint

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], which is not recalled here.

When the original program halts, new input values are generated in *solve_path_constraint*, shown in Figure 4, to attempt to force the next run to execute the last[4] 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* **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 directed search is (conservatively) detected and recorded by setting the flag *complete* to 0.

4. 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 could be treated as "functions" for the purpose of 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. (It is conceptually easy to lift this restriction by using dynamic programming techniques to compute function summaries; this is standard in interprocedural static analysis and pushdown system verification as described in [RHS95, ABE$^+$05], among many others.) 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.

4.1 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. In what follows, ϕ_f will typically be defined as a disjunction of formulas ϕ_w of the form $\phi_w = (pre_w \Rightarrow post_w)$, 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 $(x > 0 \Rightarrow ret = 1) \vee (x \leq 0 \Rightarrow ret = 0)$ (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.[5]

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

[4] 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.

[5] 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.

```
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  }
```

Assume that the constraint (hash(x)==100) corresponding to the conditional statement of line 5 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 $((x \geq 0) \wedge (x \leq 10)) \Rightarrow ret = 2$.

A precondition pre_w of a summary is thus a propositional formula built from propositions representing constraints expressible in T on function inputs. 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. Otherwise, concrete inputs satisfying the same precondition are not guaranteed to follow the same path. 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).

Note that all the preconditions in a function summary are not necessarily 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.

4.2 Computing Summaries

Function summaries can be computed by successive iterations, one path at a time. Starting with a random path, one dynamically tracks which memory locations are being read by the function and which are being written. Those locations correspond to the function inputs and outputs, respectively.

When the execution of the function terminates (either on a return statement or on a halt or abort statement), 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 $m \in Write(f, \vec{I}, w)$ written during the execution of f during the last execution w generated from a context (set of input values) $\vec{I}$. Precisely, we have

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

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 in what follows.

A summary for the execution path w in f is then $\phi_w = (pre_w \Rightarrow post_w)$. The process is repeated for successive 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. We will discuss this point further in Section 6. 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, unction 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, one computes summaries for those, and then moves up the topological sort to functions one-level up while re-using the summaries for the functions below (as described in the next subsection). This process is repeated 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 of the given theory T, summaries computed in bottom-up fashion may be incomplete in presence of statements involving constraints outside of T. For instance, in the case pf function g presented in Section 4.1, 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 very well possible that the function g is often called with values for x that satisfy this constraint. In this case, *too few* summaries are precomputed, 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.

4.3 Algorithm

A top-down SMART search algorithm is presented in Figures 5, 6 and 7. The pseudo-code shown in those figures is similar to the one presented earlier in Figures 2, 3 and 4, respectively, 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 in the previous sections. Those summaries are dynamically computed in a top-down manner following a topological sort of the call-flow graph G_P of P.

```
run_SMART () =
  complete = 1
  (*) summary = [f ↦ ∅ | f ∈ P] // Set of summaries
  repeat
    stack = ⟨⟩; I⃗ = [] ; directed = 1
    (*) context_stack = ⟨(_, _, 0)⟩ // Stack of contexts
    while (directed) do
      try (directed, stack, I⃗) =
        SMART_instrumented_program(stack, I⃗)
      catch any exception →
        print "Bug found"
        exit()
  until complete
```

Figure 5. SMART test driver

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 strategy differs from the bottom-up strategy sketched in the previous subsection since an initial calling context is determined by an initial top-down traversal and the order in which backtracking points are considered is different. This search order is also different from DART's search order. For instance, following a first execution where g calls f and then f returns, the first backtracking point considered by SMART will typically be in f (if f contains a conditional statement, etc.), while the first backtracking point considered by DART will be at the last conditional statement executed before this first run terminates (assuming a depth-first DART's search order), and will typically be after the first call to f returns (if the execution that follows contains a conditional statement, etc.).

Our algorithm starts by executing the procedure *run_SMART* described in Figure 5. The only differences with the procedure *run_DART* of Figure 2 is the initialization of a set of summaries and of a context stack, which is used to record the sequence of calling contexts for which summaries still need to be computed for the current execution, and is also used to resume execution in a previously visited calling context.

The main functionality of SMART is presented in Figure 6. 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, *SMART_instrumented_program* checks whether a summary for f is already available for the current calling context. This is done by checking whether the current concrete function input assignment satisfies one of the preconditions currently recorded in the summary for f.

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

```
SMART_instrumented_program(stack, I⃗) =
  // Random initialization of uninitialized input parameters in M⃗0
  for each input x with I⃗[x] undefined do I⃗[x] = random()
  Initialize memory M from M⃗0 and I⃗
  // Set up symbolic memory and prepare execution
  S = [m ↦ m | m ∈ M⃗0]
  ℓ = ℓ0 // Initial program counter in P
  k = 0 // Number of conditionals executed
  (*) backtracking = 1 // By default, backtrack at all branch points
  // Now invoke P intertwined with symbolic calculations
  s = statement_at(ℓ,M)
  while (s ∉ {abort, halt}) do
    match (s)
      case (m ← e):
        S= S + [m ↦ evaluate_symbolic(e, M, S)]
        v = evaluate_concrete(e, M)
        M = M + [m ↦ v]; ℓ = ℓ + 1
      case (if (e) then goto ℓ'):
        b = evaluate_concrete(e, M)
        c = evaluate_symbolic(e, M, S)
        (*) if backtracking then
          if b then
            path_constraint = path_constraint ˆ ⟨c⟩
            ℓ = ℓ'
          else
            path_constraint = path_constraint ˆ ⟨neg(c)⟩
            ℓ = ℓ + 1
          if (k = |stack|) then stack = stack ˆ ⟨0⟩
          k= k + 1
      (*) case (f : call):    // call of function f
        if backtracking then
          if (I⃗ ∈ summary(f)) then
            // We have a summary for f in context I⃗
            path_constraint = path_constraint ˆ ⟨summary(f)⟩
            // Execute f without backtracking until it returns
            backtracking = 0
            if (k = |stack|) then stack = stack ˆ ⟨1⟩
            k= k + 1
          else
            // Proceed to compute a summary for f in context I⃗
            Push (f, I⃗, k) onto context_stack
        ℓ = ℓ + 1
      (*) case (f : return):    // return of function f
        if backtracking then
          // Stop the search in f
          // Generate summary for the current path
          add_to_summary(f,path_constraint)
          return SMART_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
            M = M + [m ↦ m | m ∈ post(summary(f))]
          ℓ = ℓ + 1
      s =statement_at(ℓ,M)    // 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 SMART_solve_path_constr(k,path_constraint,stack)
```

Figure 6. SMART_instrumented_program

```
SMART_solve_path_constr(k,path_constraint,stack) =
    j = k − 1
    (*) (f, I⃗, k_f) = Top(context_stack)
    while (j ≥ k_f)
        if (stack[j] = 0) then
            path_constraint[j] = neg(path_constraint[j])
            if (path_constraint[0, ... , j] has a solution I⃗') then
                stack[j] = 1
                return (1, stack[0..j], I⃗ + I⃗')
            else j = j − 1
        else j = j − 1
    (*) if (k_f > 0) then
        Pop (f, I⃗, k_f) from context_stack
        return (1, stack[0..(k_f − 1)], I⃗)
    return (0, _, _) // This directed search is over
```

Figure 7. SMART_solve_path_constr

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, this calling context is saved by pushing it onto the context stack, and the algorithm will compute a summary for it 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 4.2. 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, *SMART_solve_path_constr*, presented in Figure 7, is called to determine where the algorithm should backtrack next. The only difference with the similar procedure used in DART is that when backtracking in a specific function f and calling context $\vec{I}$ is over, the search resumes in the last calling context saved in the context stack.

4.4 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 whose corresponding constraints are in a given decidable theory T (i.e., for which the flag *complete* always remains 1), 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.

THEOREM 2. *(Relative Soundness and Completeness) Given any program P and theory T, run_SMART terminates without printing "Bug found"* if and only if *run_DART terminates without printing "Bug found".*

Proof: (Sketch) The termination of either *run_SMART* or *run_DART* without printing "Bug found" implies that the flag *complete* remains equal to 1 throughout the search (otherwise the search does not terminate). This in turn implies that all the program statements in P generate constraints that are within the scope of the theory T used by both search algorithms.

Let $d(w)$ denote the maximum depth of an execution path w. Thus, $d(w)$ is always less or equal than the depth of the (acyclic) call-flow graph G_P of program P. The proof is by induction on $d(w)$. The base case for executions paths w such that $d(w) = 1$ is immediate since the SMART search algorithm then reduces to the DART search algorithm. The inductive case for executions paths w such that $d(w) = n$ is proved by showing that every symbolic execution performed by the DART search algorithm is simulated by a symbolic execution of the SMART search algorithm, and vice versa, given that whenever a lower-function f is called, we know by applying the inductive hypothesis to the part of w inside f that any symbolic execution performed by DART is represented by a SMART summary, and vice versa. ■

Even in the case of programs P containing exclusively statements whose corresponding constraints are all within the scope of the given decidable theory T (i.e., for which the flag *complete* always remains 1), it is in general not possible to guarantee that

> *run_SMART* terminates by printing "Bug found" *if and only if run_DART* terminates by printing "Bug found".

Indeed, if the number of execution paths is infinite, the different search order used by SMART and DART does not guarantee that both searches will eventually find the same `abort`. If there are multiple `abort`s, both searches could also find first different `abort`s. However, when the set **Execs**(P) of execution paths of P is finite, then both searches are guaranteed to find an `abort` if there is a reachable one. Moreover, by computing summaries for runs ending in an `abort` statement as for runs ending in a `halt` statement, all `abort`s can be then be found by both searches.

For programs P containing statements whose corresponding constraints are beyond the scope of the given decidable theory T, a search in its feasible program paths is in general incomplete. 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, and all paths in* **Execs**(P) *have been* covered *(but not necessarily all explicitly exercised). (c) Otherwise, run_SMART will run forever.*

Proof of (a) and (c) are immediate, while (b) follows directly from Theorem 2.

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.

4.5 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. Note that, in practice, a bound b can always be trivially 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 exploring 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 explore 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 by an example in the next section.

5. Example and Case Study

5.1 A Simple Example

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

```
1  // locate index of first character c
   // in null-terminated string s
2  int locate(char *s, int c) {
3    int i=0;
4
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 $2n$ possible distinct execution paths for that function if `c` is non-zero (and at most n if `c` is zero). Those paths are ⟨line5:else⟩, ⟨line5:then; line6:then⟩, ⟨line5:then; line6:else; line5:else⟩, etc. In short, the set of all $2n$ possible execution paths can be denoted by the regular expression: ⟨(line5:then; line6:else)^i (line5:else | (line5:then; line6:then))⟩ for $0 \leq i \leq (n-1)$.

There are $n+1$ possible return values, namely -1 (for the n paths of the form ⟨(line5:then; line6:else)^i (line5:then; line6:then)⟩ for $0 \leq i \leq (n-1)$), and $0, 1, \ldots, (n-1)$, each returned by the path ⟨(line5:then; line6:else)^i line5:else⟩ where i is equal to the return value.

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

```
11  void top(char *input) {
      // assume input is null-terminated
12    int z;
13
14    z = locate(input,'a');
15    if (z == -1) return -1;           // error code
16    if (input[z+1] != ':') return 1; // success
17    return 0;                         // failure
18  }
```

In the function `top`, there are 3 possible execution paths: ⟨line15:then⟩, ⟨line15:else; line16:then⟩ and ⟨line15:else; line16:else⟩.

Following the call to `locate`, the outcome of the test at line 15 is completely determined by the return value from function `locate` stored in `z`. In contrast, the test at line 16 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 15, n executions terminate after the then branch of line 16, and $n-1$ executions terminate in line 17. 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 explore all possible execution paths and would thus perform $3n-1$ runs for this program. In contrast, a SMART search will systematically explore all possible execution paths of the function `locate` and `top` *separately*. Precisely, a SMART search computing function summaries as described in Section 4.2 would compute $2n$ path summaries for function `locate`, whose function summary ϕ_f would then be of the form

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

Then, the SMART search would explore the feasibility of the 3 paths of the function `top` using ϕ_f to summarize function `locate`. 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 when compared to DART, while still providing the same path coverage, hence guaranteeing the same branch coverage as well (100% branch coverage is achieved in this example).

5.2 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 in this section some preliminary experiments with a subset of a challenging example of program, which actually motivated the development of SMART in the first place. This example is oSIP, an open-source implementation of the increasingly popular protocol SIP. SIP, which stands for *Session Initiation Protocol*, is a protocol for call-establishment of multi-media sessions over IP networks (in-

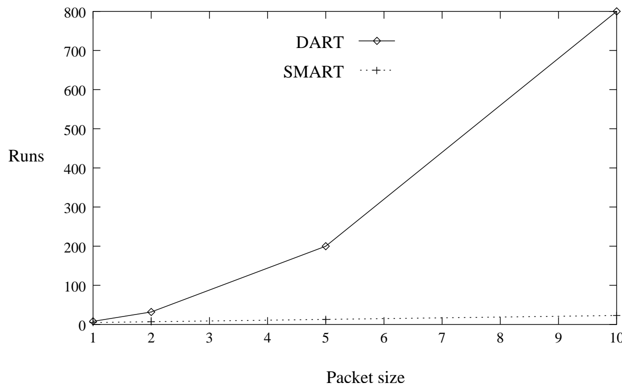

Figure 8. Experimental comparison between DART and SMART

cluding "Voice over IP"). oSIP is a C library freely available at `http://www.gnu.org/software/osip/osip.html`. It consists of about 30,000 lines of C code. Two typical applications are SIP clients (such as softphones to make calls over the internet from a PC) and servers (to route internet calls). SIP messages are transmitted as ASCII strings and a large part of the oSIP code is dedicated to parsing SIP messages.

When running the DART implementation described in [GKS05] on the oSIP parser as a whole, very low branch coverage is obtained: about 1%. (Extending the theory T used in that implementation to also reason about pointer in/equalities [SMA05] or bit-level operations [YST$^+$06] would not help here as the input is simply a (non-null) pointer to a packet whose content is unknown.) An examination of the branch coverage achieved and of the code which is never exercised reveals that this poor coverage is due to inability of this DART implementation to reason about strings. By extending the decision procedure T to include a simple theory of arrays and by instrumenting properly standard string-manipulation functions like `strcpy`, `strcmp`, etc., branch coverage can then be extended to about 30% after about 12 hours of DART search. After this, the DART search is stuck again around 30%, for another reason: the search is now "lost in space". Indeed, an execution profile of the DART search reveals that most of the backtracking performed is concentrated in a few low-level functions called over and over again by the oSIP parser. An example of such a function is `osip_clrspace(char *s)`, which essentially removes blank spaces (and also other characters such as `'\t'`, `'\r'`, `'\n'`, etc.) at the beginning of a string `s`. Such a function introduces an enormous amount of execution paths: whenever it is called, the size of the search space is multiplied by almost the number of its feasible execution paths, in a way similar to what is observed in the simple example discussed in 5.1.

Figure 8 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 containing the function `osip_clrspace` mentioned above. Experiments were performed for several, small packet sizes. Runtime is linear in the number of runs for those experiments. As is clear from Figure 8, 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.

6. Discussion

6.1 On Using Branch Coverage to Limit the Search

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. Indeed, each branch is attempted to be executed in a typically much smaller number of contexts (program paths), and some branches may fail to be exercised in those particular few contexts.

This point is illustrated with the following example.

```
  void foo(int x, int y) {
    int x_is_zero, y_is_zero;
1   if (x==0) then x_is_zero = 1;
2             else x_is_zero = 0;
3
4   if (y==0) then y_is_zero = 1;
5             else {
6               y_is_zero = 0;
7               if (x_is_zero) then abort();
8             }
9 }
```

If x and y are 32-bits integer inputs, the very first run of this program using random values for inputs x and y will likely have non-zero values for both x and y, and hence will take the else branches of all three tests. This first run is denoted by ⟨ line1:else; line4:else; line7:else ⟩. The last test on line 7 will not be backtracked as it does not involve an input. Thus, the next test to be backtracked will be the test on line 4, and the second run will force y to be zero to take the then branch of that test. The second run is thus: ⟨ line1:else; line4:then ⟩. The next backtracking point will be at the test on line 1, and the next run will force x to be zero to explore the then branch of that test. Assuming y is still zero, the third run is thus ⟨ line1:then; line4:then ⟩ where the then branch on line 4 is taken. With a "branch-coverage-driven" DART search, no backtracking will take place in line 4 after the third run since both branches on line 4 have already been covered. Thus, the fourth possible execution path ⟨ line1:then; line4:else; line7:then ⟩ will not be exercised. This means that the then branch of line 7 wil never been explored and the abort statement will be missed.

In summary, using branch coverage to guide backtracking is a simple technique for limiting "path explosion" but at the price of often making the search incomplete.

In contrast, SMART reduces the computational complexity of DART *without sacrificing* full path coverage and hence provably without reducing branch coverage.

6.2 Better Summaries with Loop Invariants

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 4.2.

For instance, consider again the function `locate` shown in Section 5.1. Instead of the function summary

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

a more compact and general function summary is $\phi_f =$

$$\begin{aligned}((\exists i \geq 0 : s[i] = c \wedge (\forall j < i : (s[j] \neq c) \wedge (s[j] \neq 0))) \Rightarrow ret = i) \\ \vee ((\exists i \geq 0 : s[i] = 0 \wedge (\forall j < i : s[j] \neq c)) \Rightarrow ret = -1)\end{aligned}$$

Indeed, the latter is independent of any maximum size n for the string `s`.

While inferring loop invariants is known to be a hard problem in general, it might be possible to design good heuristics to infer such invariants for frequent patterns of loops commonly used in specific types of programs, such as scanning an input packet for a specific value in a protocol implementation. Concrete values known at runtime could also be used in this context to facilitate the detection of "partial" loop invariants, i.e., simplified loop invariants that are valid only when some input variables are fixed.

7. 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 [God97], JavaPathFinder [VHBP00] and CMC [MPC+02], 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 as explained in Section 2. 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 often using concrete values and randomization helps in practice. More experiments with different examples of applications are needed to determine this.

Second, DART suffers from the "path explosion" problem: systematically exploring *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.

Our approach adapts known techniques for interprocedural static analysis to the context of dynamic test generation. It is worth noting that implementations of interprocedural static analysis are typically both incomplete (may miss bugs) and unsound (may generate false alarms) with respect to falsification [GK05]. In contrast, 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 – no "false alarm" is ever reported by DART or SMART, by design, as every compositional symbolic execution is grounded into some concrete execution. The only kind of imprecision in our approach is incompleteness with respect to falsification: we may miss bugs.

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.

In closing, we believe it is an exciting time to be doing research in program analysis: although most of the basic ideas behind program analysis and automatic test generation have often been published decades ago, the computational power available on modern computers makes the implementation and engineering of these old ideas feasible (or not so far-fetched) for the first time. As tremendous progress has been made during the last 10 years in the engineering of practically-usable automated code inspection tools using static analysis, we believe a big challenge and opportunity for the next 10 years is to automate software testing (as much as possible) using advances in program analysis, like those described in this paper, and taking advantage of those powerful computers.

Acknowledgments

This work was funded in part by NSF CCR-0341658.

References

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

[AY98] R. Alur and M. Yannakakis. Model Checking of Hierarchical State Machines. In *Proceedings of the Sixth ACM Symposium on the Foundations of Software Engineering (FSE'98)*, pages 175–188, 1998.

[BCH+04] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating Tests from Counterexamples. In *Proceedings of ICSE'2004 (26th International Conference on Software Engineering)*. ACM, May 2004.

[BKM02] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of ISSTA'2002 (International Symposium on Software Testing and Analysis)*, pages 123–133, 2002.

[BM83] D. Bird and C. Munoz. Automatic Generation of Random Self-Checking Test Cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[BPS00] W.R. Bush, J.D. Pincus, and D.J. Sielaff. A static analyzer for fi nding 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 *Proceedings of NDSS'04 (2004 Network and Distributed System Security Symposium)*, 2004.

[CE05] C. Cadar and D. Engler. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *Proceedings of SPIN'2005 (12th International SPIN Workshop on Model Checking of Software)*, volume 3639 of *Lecture Notes in Computer Science*, San Francisco, August 2005. Springer-Verlag.

[CG06] A. Chakrabarti and P. Godefroid. Software Partitioning for Effective Automated Unit Testing. In *Pro-*

ceedings of EMSOFT'2006 (6th Conference on Embedded Software), Lecture Notes in Computer Science, Seoul, October 2006. Springer-Verlag. Available from `http://cm.bell-labs.com/who/god/`.

[CS05] C. Csallner and Y. Smaragdakis. Check'n Crash: Combining Static Checking and Testing. In *Proceedings of ICSE'2005 (27th International Conference on Software Engineering)*. ACM, May 2005.

[DLS02] M. Das, S. Lerner, and M. Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. In *Proceedings of PLDI'02 (2002 ACM SIGPLAN Conference on Programming Language Design and Implementation)*, 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.

[FM00] J. E. Forrester and B. P. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th USENIX Windows System Symposium*, Seattle, August 2000.

[GK05] P. Godefroid and N. Klarlund. Software Model Checking: Searching for Computations in the Abstract or the Concrete (Invited Paper). In *Proceedings of IFM'2005 (Fifth International Conference on Integrated Formal Methods)*, volume 3771 of *Lecture Notes in Computer Science*, pages 20–32, Eindhoven, November 2005. Springer-Verlag.

[GKS05] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of PLDI'2005 (ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation)*, pages 213–223, Chicago, June 2005.

[GMS00] N. Gupta, A. P. Mathur, and M. L. Soffa. Generating Test Data for Branch Coverage. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, pages 219–227, September 2000.

[God97] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of POPL'97 (24th ACM Symposium on Principles of Programming Languages)*, pages 174–186, Paris, January 1997.

[God05] P. Godefroid. The Soundness of Bugs is What Matters (Position Paper). In *Proceedings of BUGS'2005 (PLDI'2005 Workshop on the Evaluation of Software Defect Detection Tools)*, Chicago, June 2005.

[HCXE02] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A System and Language for Building System-Specific Static Analyses. In *Proceedings of PLDI'02 (2002 ACM SIGPLAN Conference on Programming Language Design and Implementation)*, pages 69–82, 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*, pages 311–317, San Diego, November 1990.

[MPC+02] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of OSDI'2002*, 2002.

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

[OH96] J. Offutt and J. Hayes. A Semantic Model of Program Faults. In *Proceedings of ISSTA'96 (International Symposium on Software Testing and Analysis)*, pages 195–200, San Diego, January 1996.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symp. on Princ. of Prog. Lang.*, pages 49–61, New York, NY, 1995. ACM Press.

[SMA05] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of FSE'2005 (13th International Symposium on the Foundations of Software Engineering)*, Lisbon, September 2005.

[VHBP00] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proceedings of ASE'2000 (15th International Conference on Automated Software Engineering)*, Grenoble, September 2000.

[VPK04] W. Visser, C. Pasareanu, and S. Khurshid. Test Input Generation with Java PathFinder. In *Proceedings of ACM SIGSOFT ISSTA'04 (International Symposium on Software Testing and Analysis)*, Boston, July 2004.

[XMSN05] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution. In *Proceedings of TACAS'05 (11th Conference on Tools and Algorithms for the Construction and Analysis of Systems)*, volume 3440 of *LNCS*, pages 365–381. Springer, 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.