State Space Exploration using Feedback Constraint Generation and Monte-Carlo Sampling.

Sriram Sankaranarayanan
NEC Laboratories America.
srirams@nec-labs.com

Richard M. Chang
University of Texas, Austin.
rchang@cs.utexas.edu

Guofei Jiang
NEC Laboratories America.
gfj@nec-labs.com

Franjo Ivančić
NEC Laboratories America.
ivancic@nec-labs.com

ABSTRACT

The systematic exploration of the space of all the behaviours of a software system forms the basis of numerous approaches to verification. However, existing approaches face many challenges with scalability and precision. We propose a framework for validating programs based on statistical sampling of inputs guided by statically generated constraints, that steer the simulations towards more "desirable" traces.

Our approach works iteratively: each iteration first simulates the system on some inputs sampled from a restricted space, while recording facts about the simulated traces. Subsequent iterations of the process attempt to steer the future simulations away from what has already been seen in the past iterations. This is achieved by two separate means: (a) we perform symbolic executions in order to guide the choice of inputs, and (b) we sample from the input space using a probability distribution specified by means of previously observed test data using a Markov Chain Monte-Carlo (MCMC) technique. As a result, the sampled inputs generate traces that are likely to be significantly different from the observations in the previous iterations in some user specified ways. We demonstrate that our approach is *effective*. It can rapidly isolate rare behaviours of systems that reveal more bugs.

Categories and Subject Descriptors: D.2.5 [Testing and Debugging]: Symbolic execution, D.2.4 [Software/Program Verification]: Model checking, Statistical methods.

General Terms: Verification, Reliability, Theory.

Keywords: Model-Checking, Statistical Sampling, Monte-Carlo, Verification.

1. INTRODUCTION

Dynamic verification of software consists of simulating (or executing) some or all of its behaviours in a systematic fashion. The simulation can be carried out directly by interpreting the source code, or indirectly by instrumenting and compiling it. Simulation-based approaches may be classified in many ways based on the underlying methodology. The simulation can be *symbolic* or *concrete*, *exhaustive* or *user guided*, *systematic* or *random*.

The main advantage of dynamic approaches is the reliability of the bugs found, since these bugs correspond to real program executions. On the other hand, they cannot provide any guarantees on the absence of bugs unless applied exhaustively. Exhaustive testing is intractable and cost-ineffective in practice. Its use is limited in practice to safety critical software with relatively small input spaces. In practice, various coverage metrics such as line and branch coverage serve to quantify the thoroughness of the testing process with respect to a given metric.

Static verification techniques use source code traversal along with symbolic reasoning techniques on system abstractions to prove properties, and hence expose bugs. Static analysis techniques can deduce the presence of bugs or prove their absence by using different forms of mechanized reasoning. In theory, static analysis can be sound and exhaustive, i.e, it can guarantee 100% coverage of the program paths and input values. In practice, soundness and exhaustive search are sacrificed for scalability. Nevertheless, the success of the many popular static analysis tools such as CoVerity [20], FindBugs [19], PreFix [3], PolySpace [21] and so on are mainly due to their independence from an actual running environment.

Our approach to validation uses symbolic execution and input sampling directed by properties learned from the test output. At each iteration of our technique, the tests from the previous iterations are run and different aspects of the resulting traces are fed to learning modules to record general properties of the test outputs so far. These aspects may include standard notions such as statement coverage, branch coverage, function call sequences and various types of predicates on the program state. The goal of future iterations is to steer the simulation away from what has already been observed to expose "new" behaviours of the code. The learned facts define what has been seen so far, and therefore, the types of new behaviours that we are interested in observing.

The properties learned are used to guide future simulations in two ways. Some of the properties learned are fed into a symbolic executor to produce constraints on inputs that are more likely to yield unseen program behaviours. Secondly, the learned facts are used to specify a *bias func-*

ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.

tion that measures the " distance" of a given program trace from the previously observed runs. Such a function is used to reward runs which are *dissimilar* to the previously seen behaviors while punishing *similar* behaviors.

Our simulator combines the constraints and the bias functions by sampling inputs satisfying the constraints according to the bias function. The sampling is performed using the *Metropolis-Hastings algorithm*, and is based on a class of techniques called *Markov Chain Monte-Carlo* (MCMC) methods [6, 1]. In the long run, the algorithm ensures that inputs are sampled in proportion to the value of their bias functions. Therefore, inputs with higher bias values are sampled more often than inputs with lower bias values. As a result, the sampled traces exhibit higher bias values, or more distance from the learned facts than uniform random sampling of inputs. The resulting samples are again fed to the the learning modules, which add to the pool of learned facts from previous iterations.

In order to further enhance its effectiveness, our technique employs a lightweight static analysis front end that processes the input program and constructs a CFG representation. This representation is repeatedly simplified using property-specific slicing and constant folding. We then perform static analysis using abstract interpretation to prove as many properties of the program as possible, eliminating them from further consideration. The remaining properties are analyzed using our approach to obtain concrete inputs that drive the system to violate them. The front end makes the problem small enough to improve the overall efficacy of our technique on the parts of the code which matter to the property being checked.

We have implemented our technique as a part of the F-Soft framework to check common runtime safety properties of C programs including array overflows, pointer access violations, string usage patterns and memory leak checks. Our experiments on small and medium sized examples show that our technique is *effective*. As compared to uniform random testing, the combination of sampling and constraint generation achieves more coverage and finds more violations in the same amount of time. Furthermore, we show that the constraint generation and sampling are both essential to ensuring this performance.

Organization. Section 2 describes the basic components of our approach. Section 3 describes the simulation and the Metropolis-Hastings Algorithm. Section 4 describes the learning schemes used, Section 5 sketches the constraint generation and Section 6 describes the related work. Section 7 describes some of the implementation details. The results of experiments are described in Section 8.

2. APPROACH

Our framework consists of four basic parts as depicted in Figure 1.

Front End. The program is first passed through a static analysis front end. This constructs a CFG representation, instruments and simplifies the resulting CFG using slicing, constant folding and powerful static analyses using abstract interpretation for proving properties, thus removing many of the properties from further consideration. As a result of these simplifications, the only statements preserved are those that have a bearing on the property being checked.

The CFG built through static analysis is then simulated in our framework to find inputs that lead to bugs. Our framework is iterative with each round consisting of an application of the simulator, learner and symbolic executor.

Table 1: Useful facts learned for different properties.

Array Overflow	Pointer Access
node/branch Coverage	node/branch Coverage
Function Call Sequence	Function Call Sequence
Variable Ranges	Null Pointer Parameters
Pointer Aliases	Pointer Aliases
...	...

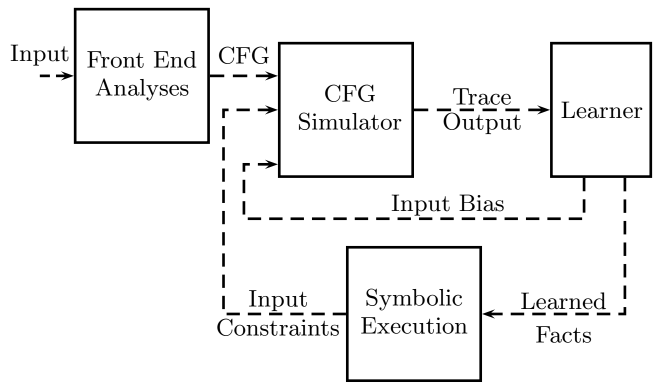


Figure 1: Block diagram summarizing our simulation framework.

Simulator. The simulator runs the CFG on selected inputs given a constraint restricting the set of inputs and a bias function guiding the selection from amongst the ones that satisfy the input constraints. The simulator works by sampling the input space using a Monte Carlo Markov Chain (MCMC) approach [6, 1]. The resulting trace data is sent to the learner module.

Learner. The learner module is responsible for accumulating facts about the traces that have been observed so far. These properties can be used to guide the simulation through constraints on the input and *bias functions* for sampling their solutions. The nature of the learner may be specific to the type of program being analyzed and property being checked. Table 1 shows some of the possible properties of the test output that we may infer automatically. These properties are used to generate test cases. Note that the properties listed therein also happen to be commonly inferred static properties when the program is statically analyzed for verification. Such properties can be obtained by learning algorithms of different levels of sophistication. Properties of interest may include standard notions of coverage, such as line coverage and branch coverage, the sequence of function calls and returns, the ranges of function parameters at the entry to a function, the points-to relationships between chosen points in the heap at different points in the program, or even the text printed out by the program.

Symbolic Execution. In order to capture some unexplored behaviours of the program, we use symbolic execution to infer constraints on inputs. Such inputs could exercise previously unseen function call sequences, visit specified program points with parameters that lie outside the inferred ranges, or with different pointer alias sets. In our approach, we use abstraction and static analysis to generate such constraints.

3. SIMULATION & SAMPLING

In this section, we describe techniques for simulating the CFG and sampling from a set of inputs.

CFG Simulation. Given a CFG representation of the program and some initial values, we simulate the program on the given input values in order to produce a trace. One of the issues with simulations is the potential for nontermination especially if the inputs do not respect the environment constraints or implicit assumptions at function entry. This is remedied by simulating the CFG for a large but fixed number of steps and artificially terminating the rest of the execution. It is also possible to encounter dead-ends in the CFGs as a result of simplification by static analysis techniques such as slicing and constant folding. The simulation automatically stops upon reaching a dead end.

Sampling: Metropolis-Hastings Algorithm

Input sampling is the most important part of the simulation process. We assume that we are given a constraint φ on the input space, and that it is possible to solve φ and obtain some solution s. The constraint arises from symbolic execution, and ideally, it guarantees that all its solutions produce distinct behaviours in some desired aspect of the output trace. In reality, however, the constraints are generated using approximations for efficiency. Therefore, not all their solutions may guarantee the desired behaviour. The objective of statistical sampling is to compensate for these approximations by controlling the choice of solutions.

In the ensuing discussion, let Ω represent the space of inputs of a program satisfying the constraint φ. For simplicity, we assume *deterministic execution*, i.e., a one-to-one mapping between each input and the resulting program trace. Our results hold even if this assumption is not satisfied.

In our framework, the learner produces a *bias function* v using data learned from previous iterations. Given a program trace T, the bias function v returns a *positive* real value $v(T) > 0$ that measures the "distance" of this trace from previously seen traces. Higher bias values for a trace indicates larger distance from previously seen behaviours. Under the deterministic execution model, this bias function may be indirectly applied on inputs s by first simulating them and evaluating the distance function on the resulting trace. If the execution is not deterministic, we may define $v(s)$ by averaging over a large number of distinct traces that running s may produce.

We now wish to sample from the input space according to the bias function $v(\cdot)$ over the inputs. This is achieved by means of a variation of the popular *simulated annealing* algorithm called the *Metropolis-Hastings* Algorithm [6, 1]. Algorithm 1 shows the schematic implementation of the algorithm. At each step, the algorithm generates a new proposal s' from the current sample s using some *proposal scheme* defined by the user. Subsequently, we compute the ratio $\alpha = v(s')/v(s)$ and accept the proposal randomly, with probability α. Note that if $\alpha \geq 1$, i.e., $v(s') \geq v(s)$, then the proposal is accepted with certainty. If the proposal is accepted then s' becomes a new sample. Failing this, s remains the current sample and no new sample is produced.

A proposal scheme is defined by a probability distribution $P(s'|s)$ that specifies the probability of proposing input s' given the current sample s. For a technical reason (known as *detailed balance*, see [6]), our version of the algorithm requires that $P(s'|s) = P(s|s')$. Furthermore, given any two inputs s, s', it should be possible with nonzero probability to generate a series of proposals $s, s_1, \ldots, s'$ that takes us from input s to s'.

Algorithm 1: Metropolis-Hastings Algorithm.

```
Input: Ω: Input Space, v(·): Bias Function,
       ProposalScheme(·): Proposal Scheme
Result: Samples ⊆ Ω
begin
    Choose some initial input s ∈ Ω.
    while (···) do        /* Require More Samples?  */
        /* Select s' using ProposalScheme          */
        s' ← ProposalScheme(s)
        α ← v(s')/v(s)
        r ← UniformRandomReal(0, 1)
        if (r ≤ α) then    /* Accept proposal?      */
            s ← s'
            Samples ← Samples ∪ {s}
        else
            /* Reject & seek fresh proposal         */
end
```

Suppose Algorithm 1 was run many times to generate a sufficiently large number $N > 0$ of samples. Let $N(s)$ denote the number of times an input s was sampled.

Theorem 3.1. *For inputs* $s_1, s_2 \in \Omega$, $\lim_{N\to\infty} \frac{N(s_1)}{N(s_2)} = \frac{v(s_1)}{v(s_2)}$.

As a direct consequence, one may conclude, for instance, that a state s_1 with $v(s_1) = 100$ is ten times more likely to be sampled than a state s_2 with $v(s_2) = 10$ in the *long run*. In theory, it is possible to prove assertions about the number N of samples required for a particular scheme to produce samples in proportion to their bias values. This number, also known as the *mixing time*, is invariably large, depending on the size of Ω. Our use of the sampling scheme simply draws for a fixed number of samples, or alternatively, until the maximum value function so far exceeds the initial value by some factor λ.

Proposal Schemes. It is relatively simple to arrive at viable schemes for generating new proposals. For instance, it suffices to simply choose an input s uniformly at random from among the inputs. However, designing a good sampling scheme needs some insight into the nature of the input space and the bias function. For our application domain, we rely on a *locality* principle for sampling the input space. We assume that if an input s is suboptimal, there exists a "nearby" input $s + \Delta$, that improves the value of the bias function. Furthermore, such a nearby input generally provides some directional information for guiding the sampling of new inputs.

One of the main complications that arises in our application is the input domain used for the sampling. The space of inputs (domain) is specified in our scheme by a constraint φ, generated by a symbolic execution. Therefore, we require proposal schemes that sample from the space of solutions to these constraints. We first consider bounding-box domains wherein φ is of the form $\bigwedge_i\ x_i \in [l_i, u_i]$, where l_i and u_i are the lower and upper bounds on the input variable x_i. Under such conditions, we may use a proposal scheme known as the *Gibbs Sampler*. Let s be the current state wherein each variable x_i lies in the range $[l_i, u_i]$.

1. Choose an input variable x_i uniformly at random among the inputs,

2. With equal probability, we increment or decrement the chosen variable by 1, i.e., $x_i' := x_i \pm 1$. Furthermore, we may force the result in the range by wrapping around. If $x_i' \geq u_i$ then $x_i' := l_i$ and vice versa.

The Gibbs Sampler chooses next inputs that differ from the current input by ± 1 over exactly one input variable. Such a sampler always works for bounding boxes and produces good convergence results in many cases, depending on the nature of the bias function used.

On the other hand, the Gibbs Sampler poses problems in input domains that are not bounding boxes. One such domain that will be used in our application is that of linear constraints. Consider the constraint

$$\psi: \; x_1 + 2x_2 = 0 \wedge x_1 \in [-10, 10] \; \wedge \; x_2 \in [-10, 10]$$

over input variables x_1, x_2. The state $\langle x_1, x_2 \rangle = \langle 0, 0 \rangle$ is a valid input satisfying ψ. On the other hand, all the neighbouring states $\langle 0, \pm 1 \rangle, \langle \pm 1, 0 \rangle$ that will be considered by the Gibbs Sampler lie outside $[[\psi]]$. Therefore, the Gibbs sampler will fail to sample faithfully in such a case.

In order to sample from an arbitrary domain φ, we may form a bounding box φ_b that overapproximates φ. Subsequently, one may sample from φ_b and accept only those samples that satisfy φ. Whereas sampling from bounding boxes is computationally inexpensive, the loss in samples due to the overapproximation may be quite high. The bounding box approximation $\psi_b : x_1, x_2 \in [-10, 10]$ of ψ shown above has $\sim$ 400 input states, whereas the domain ψ itself has $\sim$ 10 possible inputs. Therefore, using the bounding box approximation leads to one viable sample in the domain ψ for every 40 samples from the bounding box, on an average. This problem is exacerbated when the number of inputs increases.

On the other hand, it is possible to sample directly from the input domain itself, albeit at a higher computational cost. Let us assume a randomized constraint solver C that is able to return a solution s to a given constraint φ at random by using some random seeds provided by the user in the constraint solving process. We may modify the Gibbs sampler to sample from the input domain using such a constraint solver in the proposal scheme. Let s be the current input satisfying the domain constraint φ. The constraint

$$\varphi_s : \; \varphi \wedge \bigwedge \left(s(x_i) - \Delta \leq x_i \leq s(x_i) + \Delta \right),$$

for some constant $\Delta > 0$, strengthens φ. Furthermore, any solution s' of φ_s is obtained by an increment (decrement) of at most Δ to each variable in the current input state s.

A key drawback to this scheme is the ability to build a good randomized constraint solver. Techniques such as WalkSat can be generalized to extract satisfiable solutions to SAT and CSP problems in a randomized fashion, approaching uniform random sampling [35, 16]. Our implementation builds randomized solver for linear constraints by using the simplex algorithm with a randomly generated objective function.

Input: (x, y): Integers, $(x, y) \in [-100, 100] \times [0, 10]$
begin
  int $z = \textbf{multiply}(x, y)$
  assert$(z \neq 210)$
end

Figure 2: A program with an assertion violation.

Bias Functions. We now consider the specification and computation of bias functions. The bias needs to measure the "distinctness" of a trace from previously observed ones. There are many measures of distinctness imaginable. However, we may combine many different bias functions $v_1, \ldots, v_m$ into a single function by simply multiplying the values of the individual biases on a particular input, i.e., $v(s) = \Pi_{i=1}^{m} v_i(s)$. Secondly, there are numerous ways of providing values to a trace for some given concept. However, the sampling algorithm considers only the ratio of bias values for two different input values rather than the actual values. Therefore, the scale of the valuations does not matter. We demonstrate some possible bias functions corresponding to some of the facts shown in Table 1 such as node/edge coverage, variable ranges and function sequences:

Coverage Bias: Coverage is computed by recording the nodes (statements) and edges (branches) visited so far. Let s be a proposal with its simulated trace π. Let m_π be the number of *new* nodes visited by π. A simple bias value is given by the fraction $v(s) : \; m_\pi / |\pi|$. On the other hand, we may wish to reward traces that visit an unseen node earlier in their execution. Let l be the earliest position in the trace where a previously unseen node is visited. A bias function of the form $v(\cdot) : \; \frac{1}{2^l}$ rewards traces that visit a fresh CFG node earlier in their execution. Similar notions may be formulated for the uncovered CFG edges. Note that this function changes for each iteration as the nodes and edges covered also change. Also, a trace that is indistinct from a previously seen trace gets the least possible value under this scheme.

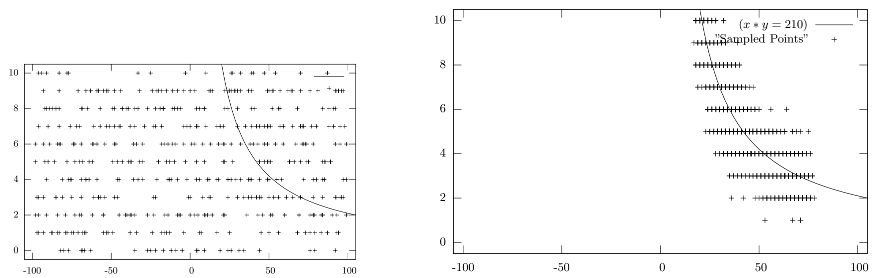
Distance to Violation: The bias function may be used to reward traces that "almost" cause an assertion in the code to be violated. Consider, for instance, an assertion that is violated whenever $e = 0$, for some arbitrary program expression e. Let T be some threshold value of e. A bias function of the form

$$v(\cdot) = \begin{cases} 0.5^{|e|}, & |e| > |T|, \\ 100.0, & |e| \leq |T| \end{cases}$$

can be used to reward traces that achieve a value of e in the range $[-T, T]$ with some large value. Example 3.1 demonstrates one such function. Assertions of the form $e_1 \langle relop \rangle \, e_2$ can be similarly treated for other relational operators. The learner may control the bias by narrowing the threshold for each iteration based on the learned data.

Range Bias: Given the ranges of selected variables at some instrumented program points, the range bias given a trace π is the number of times it visits an instrumented point with the value of some variable of interest lying outside the range given by the learner. Alternatively, one may formulate different notions of distance based on the actual deviation of the observed values from the ranges recorded so far. Once more, the bias function rewards inputs that visit instrumented points with new values for the variables.

In practice one may define numerous other bias functions



(a) (b)

Figure 3: Input simulation for example program using (a) uniform random sampling, and (b) Metropolis-Hastings algorithm. "+" denotes the sampled points and the solid curve shows the objective.

based on the property being checked and the output of the learner module.

Example 3.1. *Consider the program in Figure 2 with two inputs x, y in the range $(x, y) \in [-100, 100] \times [0, 10]$. The assertion $z = xy \neq 210$ is violated only for very few selected values of x, y. We compare two schemes for choosing the input (x, y): (a) sample uniformly at random from inside its range, and (b) use the Metropolis-Hastings algorithm with a bias function rewarding traces that get close to violating the assertion in question:*

$$v(x,y): \begin{cases} 0.9^{|xy-210|}, & \textit{if } xy \neq 210 \\ 100, & \textit{otherwise} \end{cases}$$

Figure 3 shows the scatter plots of the sampled points from a typical run of both the schemes. Note the clustering of the sampled points about the target points when the bias function is used.

The example also shows the benefit of using statistical techniques to handle nonlinear expressions such as $x * y$. These expressions cannot be handled by techniques relying purely on symbolic execution, since they typically give rise to intractable satisfiability problems.

4. LEARNING

The learning used in our framework simply records useful facts about what has been seen so far in the trace data. The learner is used to generate constraints on the input space and define the bias functions for the statistical sampling described in Section 3. In our framework, we consider learning schemes that go beyond faithfully recording output data. Our schemes also generalize from the traces seen so far to ensure that future iterations produce behaviours that are significantly different.

Coverage Learning. Coverage learning simply records if a node or an edge in the CFG has been visited in an earlier round or otherwise. This can be performed inexpensively by associating a Boolean flag with each node or edge. The resulting annotation can be used to compute a bias function that rewards traces which visit previously unseen nodes and edges. It can also be used to guide the symbolic execution process to ensure that previously unseen nodes and edges are visited in the future.

Function Call Sequences. Syntactic notions of coverage include function call sequences, that are key determinants of a program's behaviour. Many common causes of bugs including API usage errors are directly dependent on function call sequences encountered during execution. A simple scheme for "learning" such call sequences could be to store all encountered sequences.

We employ a more sophisticated approach that can generalize the call sequences to produce a finite state machine model of the calling sequences in the program [23]. The key to this approach is its use of NGRAMS to learn automata. An NGRAM is a word consisting of a sequence $N > 0$ letters. For example, "ABCD" and "EF" are 4-grams and 2-grams, respectively. NGRAMS are commonly used in natural language modeling and understanding. We use this concept to represent automata characterizing sub-sequences of a trace. Each state of the automaton is an NGRAM of length $n \in [1, N]$, denoting the occurrence of all the events denoted by the letters that make up the NGRAM. Automata edges connect these NGRAMS to represent a set of traces. Figure 4 shows an example of such an automaton. The NGRAMS in this automaton are "A", "B" and "CDE". The automaton generated from the call sequences defines a bias function based on the size of the path prefix that belongs to the language of the automaton.

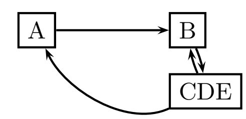


Figure 4: Ngram Automaton

Ranges. We record the ranges of variables of interest at specific program points by simply recording them. Nodes of interest include function entry and exit points. Variables of interest include parameter and return values, array indices and pointer dereferences. We also record the ranges of expressions at branch points and assertions. Note that recording the maximum and minimum values encountered rather than the individual values seen by each trace is the generalization involved in this form of learning. This enables us to compute bias functions that measure the distance to a particular assertion violation, as described earlier. Dynamically inferred invariants along the lines of tools such as Daikon can also be used as characterizations of previously seen traces [12].

5. SYMBOLIC EXECUTION

The symbolic executor utilizes the output of the learning module to generate constraints on inputs refuting the properties learned. In order to make the execution tractable, our symbolic executor treats nonlinear branches and assignments as nondeterministic. This ensures that the resulting constraints belong to tractable theories such as linear arithmetic, and therefore, are easy to solve. However, it is well known that an overapproximate symbolic execution along a path does not always lead to inputs that necessarily exercise the desired paths. Nevertheless, the constraints so obtained can improve the probability of observing the statistically rare behaviours upon sampling.

In general, the generation of constraints depends on the type of learning used for its input. Different types of learning require different abstractions and reasoning engines. In this section, we restrict ourselves to test case generation corresponding to a few of the aspects described in Section 4.

Coverage Enhancement. We compute constraints to visit a node n or an edge e that has not been previously explored. The constraints are obtained by symbolic execution of paths in the code that reach the uncovered node/edge. The process is continued until we find a feasible path that traverses the node or edge, or enough paths have been explored and none found feasible.

Function call sequences can also guide static constraint generation through a process of *chaining* [25]. Given a desirable sequence of function calls, we may formulate a series of subgoals, each of which could consist of an extension of a CFG path through an uncovered edge. The sub-goal formulation is performed through data and control dependence analysis on the flow graph.

Range Enhancement. Let n be a node of interest and x be a variable that has so far been observed in some range $[\alpha, \beta]$. We choose paths that ends in n and execute them symbolically. In order to extend the range, we simply extend each path by adding an extra "virtual" condition $x < \alpha$ or $x > \beta$ to the end of the path. One may stop extending the range of the variable based on the results of a static range analysis. The resulting symbolic execution, if feasible, may enable a visit to the node n with a value of x lying outside the range $[\alpha, \beta]$.

A typical test generation problem has two aspects to it: (a) *Path Selection* and (b) *Symbolic Execution.*

Path Selection. Paths through the CFG may be chosen by systematically exploring the CFG using a search strategy. However, for large programs, there are numerous syntactic paths, a majority of which may be infeasible. Furthermore, an exhaustive search through the CFG may be time consuming. Our solution is to perform a best effort search for reachability using a database of previously explored feasible paths. Given a reachability objective, we consider minimal extensions to the prefixes of paths in our path database that enable us to reach the objective. For each such extension, we compute a *path constraint*, which is checked for feasibility. The process terminates once enough feasible paths are found.

The database of feasible paths can be built from two sources: (a) The paths exercised by previously seen traces from test data, and (b) paths found to be feasible by previous attempts at constraint generation. The paths in the database may be stored and accessed efficiently using a modification of the *trie* data-structure [24]. The trie representation may be kept small by merging nodes inside loops to prevent unnecessary replication of loop nodes along a path and also by summarizing straight line code.

Finally, our path selection scheme maintains the stack of function calls encountered along each path in order to match calls properly with returns. While this matching happens automatically for paths obtained from concrete executions, care must be taken while extending such paths to ensure that the extensions also produce interprocedurally valid paths. Thus, all the paths selected are interprocedurally valid.

Symbolic Execution. Our objective is to obtain input constraints to exercise a particular path. Such an input constraint can be computed using the *weakest precondition* (WPC) along the chosen path. The computation of WPC is common to numerous approaches in verification and test generation. Therefore, we omit the details of this computation from this discussion.

In order to simplify the weakest precondition, we first construct a linear abstraction of the program by replacing non linear operations such as multiplication, integer division, modulo and bitwise operations to nondeterministic choice. Given a CFG, computing its linear abstraction involves a simple transformation that can be achieved in linear time. The computation of the WPC involves substitution and syntactic manipulations. It can be performed in polynomial time in the size of the path and the number of variables involved. The linearization ensures that given a path the resulting WPC is a conjunction of linear inequalities. Such constraints can be solved using standard *linear programming*(LP) solvers.

Common operations over path constraints include solving them to obtain some sample points, and a proposal scheme to choose a new solution at random, given a current solution. Even though these operations can be performed in polynomial time for linear arithmetic assertions, doing so repeatedly may lead to a large overhead. The *bounding box* abstraction can be used to further over-approximate the constraint γ by means of a outer bounding box. To do so, we compute bounds $[\alpha_i, \beta_i]$ for every input variable x_i. This can be performed efficiently based on simple ideas from interval arithmetic solvers.

6. RELATED WORK

Using Feedback. Many approaches have directly or indirectly used the coverage information from the exploration so far to guide new exploration, using many different notions of coverage. Tasiran et al. use coverage data to modify the parameters of a Markov chain model of the system under test, in order to guide further test generation for functional testing of hardware [32]. Fine & Ziv use a Bayesian network to discover relationships between the inputs and the resulting coverage. The resulting network is then used to guide the further generation of test vectors for functional testing of hardware designs [13]. Ganai & Aziz use a notion of "rarity" based on the frequency of occurrence of different components of the system state to guide state space exploration for the circuit verification [14]. Edelstein et al. present a tool that explores different context switch patterns of a multithreaded program in order to generate race conditions in a coverage guided fashion [11].

Pacheco et al. use random test generation in combination with a set of heuristics that guides the extension of previously seen unit tests to produce new tests for Java programs. They demonstrate the superiority of their approach vis-a-vis other systematic approaches to the problem [30].

Random Testing. Numerous approaches to testing rely on inputs that are sampled randomly from fixed or changing probability distributions. *Random testing* samples uniformly at random from the space of inputs, essentially treating the program as a black box [17]. This approach has been implemented in tools such as JCrasher [9] and QuickCheck [7]. Simplicity and ease of implementation are the essential advantages to random testing. There are many disadvantages: inputs of interest frequently have a low probability of being discovered by chance alone.

Distance Metrics. Bias functions, or more generally, distance metrics have been used elsewhere to guide test generation. The work of Korel uses goal-directed exploration of program paths by minimizing a distance metric [25]. The distance metric is similar in form to the "distance to violation" metric discussed in Section 3. This work also uses the chaining method for using a sequence of subgoals to satisfy a larger goal. Michael et al. use a similar approach of formulating objective functions, which are minimized using genetic algorithms and gradient descent [27].

Model Checking & Symbolic Execution. Numerous approaches use model checking and symbolic execution to explore the state space of programs either systematically or otherwise. Model checking for software has been implemented in tools for explicit state exploration such as Murφ [10], SPIN [18], JPF [33], CMC [28], among many others; and symbolic exploration in tools such as CBMC [8] and F-Soft [22]. Tools such as JPF [34], Symstra [36] and jCUTE [31] use symbolic execution in order to generate constraints exercising program paths.

Combining Static & Dynamic. Recently, there has been a slew of work on combining symbolic and concrete execution for test case generation [15, 4, 31]. The approach (termed *concolic testing* by Sen et al.) consists of instrumenting a running program to produce constraints along its execution path, so that the path exercised by the program can be symbolically executed to perform further test case generation. Our approach shares many similarities with concolic testing, including the idea of combining symbolic and concrete executions. On the other hand, whereas CUTE symbolically executes all paths up to some fixed depth cutoff, our approach uses a learning scheme to drive path exploration towards new paths for symbolic execution. Finally, rather than run the program on a few solutions to symbolic execution constraints, our approach samples many inputs from the constraints discovered by the symbolic execution according to a bias function learned from previously observed tests.

The recent work of Majumdar and Sen extends concolic testing with random simulation. This approach, called *Hybrid Concolic testing* uses a combination of random exploration of the state space along with directed symbolic execution to test software [26].

Learning. Numerous approaches combine random testing with static reasoning to obtain novel ways of proving properties or finding bugs in programs. In general, data from testing has been used to infer invariants, construct abstractions by learning predicates, specifications and likely invariants. The work of Ernst et al. uses test data to infer *likely invariants* [12]. These invariants have been used in a static analysis framework to prove program properties [29]. Commonly generated invariants include linear equalities among the variables in the program and properties of data structures such as linked lists.

Facts learned from test output have been used as formal specifications. The work of Yang et al. infers common function call sequences from the test data. These sequences are used as specifications for static analyzer such as ESP to discover anomalous call sequences [37]. Chang, et al. propose a similar technique that refines a set of candidate property specifications based on programs executions to efficiently infer a wider set of commonly occurring properties [5]. We use the approach of Jiang et al. [23] to learning call sequences. However, whereas the resulting automata in this approach is used in the aforementioned work to monitor the program and flag deviations, we use it to guide our simulation by means of a bias function.

7. IMPLEMENTATION

We have implemented an instance of our framework using the front-end infrastructure for checking C programs [22]. Our implementation checks for memory safety issues such as the absence of overflows, pointer access violations, double frees and so on. We can also check the usage of the standard library string functions for the C language. In this section, we discuss some of the implementation details.

Front-End & Static Analysis: We use various front-end features such as instrumentation for the insertion of extra variables to track allocated boundaries of arrays and pointers, heap and stack modeling, the automatic insertion of checks for array accesses, pointer dereferences, memory allocations, reallocations and frees. The model is then simplified by a series of transformations. Many of the checks inserted are eliminated through pointer analyses at different levels of precision, and abstract interpretation techniques such as *interval analysis*, *octagon analysis* and *polyhedral analysis*, along the lines of the Astreé project [2]. At each stage, we statically slice the CFG to eliminate code irrelevant to the properties being checked. The resulting CFG consists of the slice that corresponds to the uneliminated checks.

The front end transformations are faithful to the semantics of the original code. However, one important exception is the handling of the contents of large arrays. It is intractable to handle the contents of large arrays, especially those of unknown size, during the CFG construction phase. Therefore, we model a bounded number of positions as specified by the user, while soundly abstracting the remaining positions to nondeterministic choices. This limitation also carries over to dereferences of pointers which may point to unmodelled array positions, and heap data structures such as linked list, which frequently use arrays of pointers.

Simulator: We simulate the effect of the CFG nodes and branches along with those of many standard library functions such as memory management, string handling and other common utilities. Other functions with unknown source code are assumed to return random values. A future extension could enable the use of precompiled libraries for such functions.

The environment at the start of the simulation can be specified by function preconditions that are checked during simulation and incorporated into the symbolic execution. However, such preconditions are rarely available in a level of detail sufficient to recreate valid inputs. As a result of this, and other abstractions during the CFG building process, some of the bugs we produce during simulations are potentially false bugs. Table 2 summarizes some of the issues handled in the implementation of simulators.

Learning: Our implementation provides hooks for many types of learning modules that may be specific to the type

Table 2: Language and system issues that complicate simulation.

Issue	Solution
Long (nonterminating) simulations	User-defined limit on the number of steps
Unknown input environment	User-specified preconditions
System calls, library functions	Explicitly model common func. Rest are treated as nondeterminstic.
Pointer dereference of untracked address	Evaluate to a random value.
Large array indexing	Track array length, null termination and a bounded number of elements.
Data-structures	Enable preconditions on data structure shapes (not implemented).

of code being analyzed or the property being checked. Currently, we have implemented learning schemes for recording CFG node and edge coverage, function call sequences using a variant of the algorithm described by Jiang et al. [23], ranges for function parameters and expression ranges at assertion violations. Each learning module also implements bias functions along the lines of those discussed in Section 3, that can be called by the sampler to evaluate a proposed input. Furthermore, the coverage and range data are used to generate constraints by symbolic execution.

Symbolic Execution: As described in detail in Section 5, we search for interesting paths by extending existing paths in the CFG stored in a trie to satisfy coverage or range enhancement objectives. Our symbolic execution currently supports linear arithmetic (LP) and bounding box (BBOX) constraints.

Our implementation tightly couples the components described above. Each round of the procedure consists of sampling a fixed number of inputs from each of the constraints produced in the previous round. The traces resulting from the samples are added to the database. The bias function is updated using the learning output, and the test generator module is run to produce a new set of constraints. The overall process consists of iterating our procedure for a fixed number of rounds. We observe that a small number of rounds (typically 10) suffices for our test examples. The number of samples drawn per constraint can be varied depending on the estimated number of solutions to the constraint. For the purpose of our experiments, this number is fixed at 1000 samples.

8. EXPERIMENTS

We have applied our techniques to a few example C programs shown in Table 3. The benchmarks are typically implementations of modules offering various functionalities to the system as a whole. These functionalities are broadly classified into a series of initializations, operations and cleanup. The benchmarks include common data structure implementations used heavily in the F-Soft front-end (and our tool implementation), small application libraries and device drivers. In some cases harnesses were already available to exercise all the operations exported by the library. In other cases, we built our own harnesses to exercise some initialization sequence, operation sequence and cleanup based on input arguments to the harness function. Table 3 also shows the performance of the front end. Note that in many cases, the front end is able to prove a large fraction of the properties.

We performed the following experiments to measure the impact of the various components of our framework on the coverage and the number of checks violated by our inputs:

(A) Symbolic execution & bias function sampling. Two versions of this experiment were run to compare symbolic evaluators using LP (linear arithmetic) and BBOX (bounding box) abstractions, respectively.

(B) Bias function sampling using MCMC, but no symbolic execution.

(C) Uniform random sampling of inputs (given as much time as Expt. A(LP)). This experiment serves as a baseline for comparison with the results of the other experiments.

Table 5: Bug Analysis for Experiments.

Name	Bug	Comment
ARRAYCR	4	Write to unallocated pointer field.
BIGINT	1	Double free.
VAR-SET	13	Handling of corner case.

Experiments A and B are each iterated for 10 rounds. Table 4 shows the results of the experiments. For each experiment, we report the time taken in seconds, number of warnings found and the coverage as a percentage of the total number of nodes and edges in the CFG. This coverage is measured on the sliced CFG and has no bearing on the coverage on the code as a whole. Secondly many of the nodes left in the CFG may never be reached. Therefore, 100% coverage may not always be possible on the sliced CFG.

We find that in almost all cases the presence of constraints and the sampling helps find bugs faster achieve more coverage on the simplified CFG. Comparing symbolic execution engines using linear arithmetic and bounding boxes, the LP engine seems to provide better coverage in all but one of the cases. However, this improved coverage is at the cost of an almost 2x extra overhead in terms of time. In practice, it is a straightforward matter to combine the relative merits of the experiments by running different option sets repeatedly and slicing the CFG each time based on the checks violated thus far.

Table 5 shows the bugs found by our techniques on the examples. Our tool found a bug in the ARRAYCR example due to the access of an uninitialized pointer. For the VAR-SET example, our tool found 13 distinct violations, one of which was due to the wrong usage of the API in our test harness code (insufficient documentation), and the rest due to the corner case of sets with zero elements in the universe. However, very few of these violations could be found by random simulation. The warnings issued by our tool in the other cases turned out to be false because of precondition violations, missing functions and heap modeling.

Finally, Table 6 shows the result of running the CUTE tool due to Sen et al. [31] on some of the smaller benchmarks. The larger benchmarks proved to be difficult to run in CUTE since they had numerous library calls for which the

Table 3: Description of programs used for experiments. Time taken for the front end is in seconds. #Prop: Total Checks, #Prf.: Checks eliminated by front-end, N/E: Nodes/Edges in the remaining CFG.

Name	LOC	#Funs	Description	Property Checked	Static Analysis			
					Time	#Prop.	#Prf	N/E
ARRAYCR	280	7	Arrays with constant time reset	Overflow	6	65	56	105/116
SKIPLIST	400	6	Skiplist implementation	Pointer Access	33	53	15	332/404
BIG-INT	663	9	Arithmetic on big integers	Overflow	66	90	65	359/443
VAR-SET	768	11	Sets with bit-vectors	Overflow	190	212	31	697/932
ST-TABLE	790	10	A hashtable implementation	Overflow	92	258	170	519/654
STRAPP1	2.1K	3	Voice recognition system	C String Usage	21	24	17	185/221
STRAPP2	2.4K	2	Voice recognition system	C String Usage	29	75	70	225/270
DRIVE1	394	6	An example block driver	Pointer Access	1.6	47	32	144/164
IPOIB	7.7K	11	A linux device driver	Pointer Access	730	155	114	1363/1749

Table 4: Summary of results of our experiments on the benchmark programs. T: running time in seconds, #E: Number of warnings, Cov.: Node/Edge Coverages in percentages.

Name	Expt. A(LP)			Expt. A(BBOX)			Expt. B (MC. Rand.)			Expt. C (Unif. Rand.)		
	T	#E	Cov.	T	#E	Cov.	T	#E	Cov.	T	#E	Cov.
ARRAYCR	20	4	94/90	14	4	94/90	5	4	94/90	20	2	92/86
SKIPLIST	64	3	29/26	64	3	29/26	60	3	29/26	66	3	29/26
DRIVE1	40	**7**	**89/83**	2	4	41/37	.2	1	13/10	40	0	35/30
BIGINT	9.5	2	33/29	9.5	2	33/29	9.5	2	33/29	9.5	0	8/7
VAR-SET	568	**16**	**75/62**	39	4	32/26	0.72	4	31/25	569	2	26/21
ST-TABLE	32	11	66/58	32	11	66/58	27	14	67/59	32	14	67/59
STRAPP1	39	2	51/47	15	2	51/46	7	2	51/47	39	1	15/14
STRAPP2	27	1	**54/51**	8	1	44/40	.4	0	26/23	27	0	17/15
IPOIB	214	20	71/69	209	21	71/69	95	21	71/69	214	19	70/68
Tot.	1120	74	65/59 (Avg)	492	60	55/50	216	59	51/47	1125	49	44/40

Table 6: Result of CUTE on the benchmarks.

Name	Bugs Found	Cov(%)	Time
ARRAYCR	0	-	< 1
SKIPLIST	0	-	< 1
VARSET	1	47	3
ST-TABLE	0	55	87
BIGINT	1	18	< 1

code was unavailable. Our front end automatically rewrites such calls to return nondeterministic values. The coverage reported by the tool cannot be directly compared to the CFG coverage. Our attempts to run CUTE directly on the CFG representation (output as a C program) have so far been unsuccessful.

9. CONCLUSION

We have presented a framework for state space exploration by combining static analysis with sampling based on learning and bias functions. Our approach has been experimentally validated and our preliminary results are quite promising. In the future, we hope to remove some of the hurdles towards making the tool practical by addressing some of its current limitations. We also plan to extend our approach to other domains such as the analysis of multi-threaded systems.

Acknowledgements. We thank Aswin Sankaranarayanan for providing many useful pointers to Monte-Carlo techniques. We thank the anonymous referees for their insightful reviews.

10. REFERENCES

[1] Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I. Jordan, *An introduction to MCMC for machine learning*, Machine Learning **50** (2003), 5–43.

[2] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jerome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival, *A static analyzer for large safety-critical software*, ACM SIGPLAN PLDI'03, vol. 548030, ACM Press, June 2003, pp. 196–207.

[3] William R. Bush, Jonathan D. Pincus, and David J. Sielaff, *A static analyzer for finding dynamic programming errors.*, Softw., Pract. Exper. **30** (2000), no. 7, 775–802.

[4] Cristian Cadar and Dawson R. Engler, *Execution Generated Test Cases: How to make systems code crash itself.*, SPIN, LNCS, vol. 3639, Springer–Verlag, 2005, pp. 2–23.

[5] Richard M. Chang, George S. Avrunin, and Lori A. Clarke, *Property inference from program executions*, 2006, University of Massachusetts, Amherst, Tech report number: UM-CS-2006-26.

[6] Siddhartha Chib and Edward Greenberg, *Understanding the Metropolis-Hastings algorithm*, The American Statistician **49** (1995), no. 4, 327–335.

[7] Koen Claessen and John Hughes, *Quickcheck: a lightweight tool for random testing of haskell programs.*, ICFP, 2000, pp. 268–279.

[8] Edmund Clarke, Daniel Kroening, and Flavio Lerda, *A tool for checking ANSI-C programs*, TACAS, LNCS, vol. 2988, Springer, 2004, pp. 168–176.

[9] Christoph Csallner and Yannis Smaragdakis, *Jcrasher: an automatic robustness tester for Java.*, Softw., Pract. Exper. **34** (2004), no. 11, 1025–1050.

[10] David L. Dill, *The murφ verification system*, Conference on Computer-Aided Verification, LNCS, Springer-Verlag, July 1996, pp. 390–393.

[11] Orit Edelstein, Eitan Farchi, Yarden Nir, Gil Ratsaby, and Shmuel Ur, *Multithreaded java program test generation.*, IBM Systems Journal **41** (2002), no. 1, 111–125.

[12] Michael D. Ernst, *Dynamically discovering likely program invariants*, Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[13] Shai Fine and Avi Ziv, *Coverage directed test generation for functional verification using bayesian networks*, DAC'03, ACM Press, 2003, pp. 286–291.

[14] Malay K. Ganai and Adnan Aziz, *Rarity based guided state space search.*, ACM Great Lakes Symposium on VLSI, 2001, pp. 97–102.

[15] Patrice Godefroid, Nils Klarlund, and Koushik Sen, *DART: Directed Automated Random Testing*, ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05), 2005, pp. 213–223.

[16] Vaibhav Gogate and Rina Dechter, *A new algorithm for sampling csp solution uniformly at random*, CP'07, 2007.

[17] R. Hamlet, *Random testing*, Encyclopedia of Software Engineering (J.Marciniak, ed.), Wiley, 1994, pp. 970–978.

[18] Gerard J. Holzmann, *The SPIN Model Checker: Primer and reference manual*, Addison-Wesley, 2004.

[19] David Hovemeyer and William Pugh, *Finding bugs is easy*, ACM SIGPLAN Notices **39** (2004), no. 12, 92–106.

[20] CoVerity Inc., *Coverity verification toolsuite*, `http://www.coverity.com`.

[21] PolySpace Inc., *Polyspace verification toolsuite*, `http://www.polyspace.com`.

[22] Franjo Ivančić, Zijiang Yang, Malay K. Ganai, Aarti Gupta, and Pranav Ashar, F-Soft*: Software verification platform*, Computer-Aided Verification (CAV 2005), LNCS, vol. 3576, Springer–Verlag, 2005, pp. 301–306.

[23] Guofei Jiang, Haifeng Chen, Cristian Ungureanu, and Kenji Yoshihira, *Multi-resolution abnormal trace detection using varied-length N-grams and automata.*, ICAC, IEEE Computer Society, 2005, pp. 111–122.

[24] Donald E. Knuth, *The art of computer programming, vol. 3*, Addison-Wesley, 1997.

[25] Bogdan Korel, *Automated test data generation for programs with procedures*, ACM SIGSOFT Software Engineering Notices **21** (1996), no. 3, 209–215.

[26] Rupak Majumdar and Koushik Sen, *Hybrid concolic testing*, ICSE'07, 2007, pp. 416–426.

[27] Christoph C. Michael, Gary McGraw, and Michael A. Schatz, *Generating software test data by evolution*, IEEE Trans. Software Engineering **27** (2001), no. 12, 1085–1108.

[28] Madanlal Musuvathi, David Park, Andy Chou, Dawson R. Engler, and David L. Dill, *CMC: A Pragmatic Approach to Model Checking Real Code*, OSDI, Dec 2002, pp. 75–88.

[29] Jeremy W. Nimmer and Michael D. Ernst, *Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java.*, Electr. Notes Theor. Comput. Sci. **55** (2001), no. 2.

[30] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball, *Feedback-directed random test generation*, ICSE'07, 2007, To Appear.

[31] Koushik Sen, Darko Marinov, and Gul Agha, *Cute: A concolic unit testing engine for c*, ESEC/FSE'05, ACM Press, 2005.

[32] Serdar Tasiran, F. Fallah, D. G. Chinnery, S.J. Weber, and K. Keutzer, *A functional validation technique: Biased random simulation guided by observability-based coverage*, Proc. IEEE Conf. on Computer Design (ICCD'01), IEEE press, 2001, pp. 82–88.

[33] Willem Visser, Klaus Havelund, G. Brat, S. Park, and Flavio Lerda, *Model checking programs*, Automated Software Engineering Journal **10** (2003), no. 2.

[34] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid, *Test input generation with java pathfinder.*, ISSTA, 2004, pp. 97–107.

[35] Wei Wei, Jordan Erenrich, and Bart Selman, *Towards efficient sampling: Exploiting random walk strategies*, AAAI'04, 2004.

[36] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin, *Symstra: A framework for generating object-oriented unit tests using symbolic execution.*, TACAS, LNCS, vol. 3440, 2005, pp. 365–381.

[37] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das, *Perracotta: Mining temporal API rules from imperfect traces*, Proc. of ICSE, 2006.