Faster Creation of Smaller Test Suites (with SNAP)

Jianfeng Chen, Tim Menzies
Computer Science, North Carolina State University, USA
jchen37@ncsu.edu, tim@menzies.us

Abstract—State-of-the-art theorem provers, combined with smart sampling heuristics, can generate millions of test cases in just a few hours. But given the heuristic nature of those methods, not all of those tests may be valid. Also, test engineers may find it too burdensome to run all those tests.

Within a large space of tests, there can be redundancies (duplicate entries or similar entries that do not contribute much to overall diversity). Our approach, called SNAP uses specialized sub-sampling heuristics to avoid finding those repeated tests. By avoiding those repeated structures SNAP explores a smaller space of options. Hence, it is possible for SNAP to verify all its tests.

To evaluate SNAP, this paper applied 27 real-world case studies from a recent ICSE’18 paper. Compared to prior results, SNAP’s test case generation was 10 to 3000 times faster (median to max). Also, while prior work showed that their tests were 70% valid, our method generates 100% valid tests. Most importantly, test engineers would find it relatively easy to apply SNAP’s tests since our test suites are 10 to 750 times smaller (median to max) than those generated using prior work.

Index Terms—SAT solvers, test suite generation, mutation.

I. INTRODUCTION

This paper replicates and improves QuickSampler [1], a recent ICSE’18 paper which built test suites by applying theorem provers to logical formula extracted from procedural source code. We found that QuickSampler generated test suites with many repeated entries. After applying some redundancy avoidance heuristics (defined below), our new algorithm (called SNAP) runs much faster than QuickSampler and returns much smaller test suites. This is useful since smaller test suites are simpler to execute and maintain.

To generate tests from programs, they must first be converted into a logic formula. Fig. 1 shows how this might be done. Symbolic/dynamic execution techniques [2], [3] extract the possible execution branches of a procedural program. Each branch $B_i$ is a conjunction of conditions $B_i = C_x \land C_y \land \ldots$ so the whole program can be summarized as the disjunction $B_1 \lor B_2 \lor \ldots$. Using deMorgan’s rule [4] these clauses can be converted to conjunctive normal form (CNF) where:

- The inputs to the program are the variables in the CNF;
- A test is valid if uses input settings that satisfy the CNF;
- A test suite is a set of valid tests.
- One test suite is more diverse than another if it uses more variable within the CNF disjunctions. Diverse test suites are better since they cover more parts of the code.

Theorem provers like Z3, pycoSAT, MathSAT, or vZ [4]–[7] can use this CNF as follows:

$$P \lor Q \equiv (\neg P \land \neg Q) \quad \text{and} \quad \neg(P \land Q) \equiv \neg P \lor \neg Q.$$

In practice, generation can be very slow indeed. When translated to CNF, the case studies explored in this paper require 5000 to 500000 variables within 17,000 to 2.6 million clauses (median to max). Even with state-of-the-art theorem provers like Z3, generating a single test from these clauses can take as much as 20 minutes. Worse still, this process must be repeated many times to find enough tests to build a diverse test suite.

To address the runtime issue, many researchers try to minimize the calls to the theorem prover. In that approach, heuristics are used to generate most of the tests. For example, QuickSampler assumes that adding the deltas between valid tests $a, b$ to a valid test $c$ will produce a new valid test $d$; i.e.

$$d = c \oplus (a \oplus b)$$

(where $\oplus$ means “exclusive or”). Using Eq. 1 QuickSampler generates thousands, or even millions of test cases within hours. But there is a catch – heuristically generated tests may be invalid. According to Dutra et al., 30% of the tests built by QuickSampler are often invalid.

The research of this paper began when we observed that many of the test cases generated by QuickSampler contained duplicates. For example, in the blasted_case47 case study, QuickSampler generated more than $10^7$ samples within 2 hours. On these, there were only approximate $2.62 \times 10^5$ unique solutions (i.e. one in 40).

Based on this observation, we conjectured:

**If a slow test generator has some redundancy, then to build a faster generator, avoid that redundancy.**

To test this conjecture, we built SNAP. Like QuickSampler, SNAP uses a combination of Eq. 1 (sometimes) and Z3 (at other times). SNAP also includes some specialized sub-sampling heuristics that strive to avoid the redundant tests.
1 int mid(int x, int y, int z) {
  2 if (x < y) {
  3   if (y < z) return y;
  4   else if (x < z) return z;
  5   else return x;
  6 } else if (x < z) return x;
  7 else if (y < z) return z;
  8 else return y;
}

The code above has the six branches shown below. Each branch can be modeled as a logical constraint \( C_1 \lor C_2 \lor \ldots \lor C_6 \). A valid test selects \( x, y, z \) such that it satisfies these constraints.

- path 1: \[ C_1: x < y < z \] L2->L3
- path 2: \[ C_2: x < z < y \] L2->L3->L4
- path 3: \[ C_3: z < x < y \] L2->L3->L4->L5
- path 4: \[ C_4: y < x < z \] L2->L6
- path 5: \[ C_5: y < z < x \] L2->L6->L7
- path 6: \[ C_6: z < y < x \] L2->L6->L7->L8

By convention, the disjunction \( \lor C_i \) is transformed into the conjunction normal form (CNF) \( C'_1 \lor C'_2 \ldots \). A valid assignment to the CNF; i.e. the assignment that fulfills all clauses, is corresponding to a test case, covering some branch of code. When translated into the input required for Z3, these conjunctions look like:

\[ 1 p \ cnf 11511 41411 \\
2 \ldots \\
3 \neg 11507 11510 0 \\
4 \neg 11510 11504 11507 11502 0 \\
5 \ldots \\
\ldots \land (\neg 11504 \lor 11510) \land (\neg 11510 \lor 11504 \lor 11507 \lor 11502) \land \ldots \]

Fig. 1: From source code (top left) to constraint solver. Note one detail: as \( x, y \) and \( z \) are integers, we could use (say) 7 bits to representing them. If so, then tools like SMT \([8]\) could find subsets of continuous ranges that satisfy branch conjunctions. That said, this paper is comparing its methods to those of \textit{QuickSampler} study. Accordingly, we use the same conventions as \textit{QuickSampler}; i.e. our constraints are just boolean variables.

Those heuristics are described later in this paper. For now, all we need say is that \textit{SNAP} explores and generates a much smaller sample of solutions than \textit{QuickSampler}. Hence it runs faster, and generates smaller test suites.

This paper evaluates \textit{SNAP} via four research questions.

\textbf{RQ1: How reliable is the Eq. 1 heuristic?} One reason we advocate \textit{SNAP} is that, unlike \textit{QuickSampler}, our method verifies each test (with Z3). But is that necessary? How often does Eq. 1 produce invalid tests? In our experiments, we can confirm Dutra \textit{et al.}'s estimate that the percent of invalid tests generated by Eq. 1 is usually 30% or less. However, that median result does not quite characterize the variability of that distribution. We found that in a third of case studies, the percent of valid tests generated by Eq. 1 is 25% to 50% (median to max). Hence we say:

\begin{center}
\textbf{Conclusion \#1:} Eq. 1 should not be used without a verification of the resulting test.
\end{center}

One useful feature of \textit{SNAP} is that its test suites are so small that it is possible to quickly verify all our final candidate test suites. That is, unlike \textit{QuickSampler}, all \textit{SNAP}'s tests are valid.

\textbf{RQ2: How diverse are the \textit{SNAP} test cases?} Since \textit{SNAP} explores far fewer tests than \textit{QuickSampler}, its tests suites could be less diverse. Nevertheless:

\begin{center}
\textbf{Conclusion \#2:} The diversity of \textit{SNAP}'s test suites are not markedly worse than those of \textit{QuickSampler}.
\end{center}

\textbf{RQ3: How fast is \textit{SNAP}?} \textit{SNAP} was motivated by the observation that \textit{QuickSampler} built many similar test cases; i.e. much of its analysis seemed redundant. If so, we would expect \textit{SNAP} to generate test cases much faster than \textit{QuickSampler} (since it avoids redundant analysis). This prediction turns out to be true. In our case studies:

\begin{center}
\textbf{Conclusion \#3:} \textit{SNAP} was 10 to 3000 times faster than \textit{QuickSampler} (median to max).
\end{center}

\textbf{RQ4: How easy is it to apply \textit{SNAP}'s test cases?} Finally, we end on a pragmatic note. The \textit{smaller} a test suite, the \textit{easier} it is for programmers to run those tests. Therefore it is important to ask which method produces fewer tests: \textit{QuickSampler} or \textit{SNAP}? We find that:

\begin{center}
\textbf{Conclusion \#4:} \textit{SNAP}'s test cases were 10 to 750 times smaller than those of \textit{QuickSampler} (median to max).
\end{center}

Hence, we argue that it would be easiest for an industrial practitioner to execute and maintain \textit{SNAP}'s test suite.

In summary, the unique contributions of this paper are:

- A novel mutation based sampling algorithm name \textit{SNAP}.
- Experiments on common case studies that compare \textit{SNAP} to a recent state-of-the-art sampler in method (\textit{QuickSampler}, from ICSE'18);
- Based on that comparison, we show that \textit{SNAP} generates much smaller solutions that the diverse as the prior state-of-the-art; and does so far faster. Further, 100% of our tests are valid (while other methods may only generate 70% valid tests, or less).
- A reproduction package for this paper, and \textit{SNAP}.

The rest of this paper is structured as follows: \S III introduces some related works in solving this problem. \S IV shows the core algorithm of \textit{SNAP}. \S IV addresses the details of ex-

2 Source code at https://github.com/ai-se/SatSpaceExpo
periments and the study cases. §V discusses the experiment results. Following that, §VI and §VII have further discussion and conclusions.

II. RELATED WORK

Using the methods of Fig. I software often generates CNF with three or more variables per clause. Since 3-SAT problem is NP-complete [18], so generating tests from these clauses is an inherently slow process.

This problem has been explored for decades. One way to solve the theorem proving problem is to simplify or decompose the CNF formulas. A recent example in this arena was GreenTire, proposed by Jia et al. [19]. GreenTire supports constraint reuse based on the logical implication relation among constraints. One advantage of this approach is its efficiency guarantees. Similar to the analytical methods in linear programming, they are always applied to specific class of problem. However, even with the improved theorem prover, such methods may be difficult to be adopted in large models. GreenTire was tested in 7 case studies. Each case study was corresponding to a small code script with tens lines of code, e.g. the BinTree in [20]. For the larger models, such as those explored in this paper, the following methods might do better.

Another approach, which we will call sampling, is to combine theorem provers Z3 with stochastic sampling heuristics. For example, given random selections for \( b, c \), Eq. [1] might be used to generate a new test suite, without calling a theorem prover. Theorem proving might then be applied to some (small) subset of the newly generated tests, just to assess how well the heuristics are working. Table I includes some of related works.

The earliest sampling tools were based on binary decision diagrams (BDDs) [21]. Yuan et al. [9], [11] build a BDD from the input constraint model and then weighted the branches of the vertices in the tree such that a stochastic walk from root to the leaf was able to generate samples with desired distribution.

In other work, Iyer proposed a technique named RACE which has been applied in multiple industrial solutions [10]. RACE (a) builds a high-level model to represent the constraints; then (b) implements a branch-and-bound algorithm for sampling diverse solutions. The advantage of RACE is its implementational simplicity. However, RACE, as well as the BDD-based approached introduced above, return highly biased samples, that is, highly non-uniform samples. For testing, this is not recommended since it means small parts of the code get explored at a much higher frequency than others.

Using a SAT solver WalkSat [22], Wei et al. [12] proposed SampleSAT. SampleSAT combines random walk steps with greedy steps from WalkSat. This method works well in small constraint models. However, due to the greedy nature of WalkSat, the performance of SampleSAT is highly skewed as the size of the constraint model increases.

For seeking diverse samples, universal hashing [23] techniques have been proposed. These algorithms were designed for strong guarantees of uniformity. Meel et al. [17] provided an overview of key ingredients of integration of universal hashing and SAT solvers; e.g. with universal hashing, it is possible to guarantee uniform solutions to a constraint model. These hashing algorithms can be applied to the extreme large models (with near 0.5M variables). More recently, several improved hashing-based techniques have been purposed to balance the scalability of the algorithm as well as diversity (i.e. uniform distribution) requirements. For example, Chakraborty et al. proposed an algorithm named UniGen [15], following by the Unigen2 [16]. UniGen provides strong theoretical guarantees on the uniformity of generated solutions and has applied to constraint models with hundreds of thousands of variables. However, UniGen suffered from a large computation resource requirement. Later work explored a parallel version of this approach. Unigen2 achieved near linear speedup of the number of CPU cores.

To the best of our knowledge, the state-of-the-art technique

| Reference | Year | Citation | Sampling methodology                | Case study size | Verifying samples | Distribution/diversity reported |
|-----------|------|----------|-------------------------------------|-----------------|-------------------|--------------------------------|
| [9]       | 1999 | 105      | Binary Decision Diagram             | ≈1.3K           | ○                 | ○                              |
| [10]      | 2003 | 50       | Interval-propagation-based         | 200             | ○                 | ○                              |
| [11]      | 2004 | 54       | Binary Decision Diagram            | < 1K            | ○                 | ○                              |
| [12]      | 2004 | 141      | Random Walk + WalkSAT              | No experiment conducted | ○ | ○ |
| [13]      | 2011 | 88       | Sampling via determinism           | 6k              | ○                 | ○                              |
| [14]      | 2012 | 25       | MAXSAT + Search Tree               | Experiment details not reported | ○ | ○ |
| [15]      | 2014 | 29       | Hashing based                      | 400K            | ○                 | ○                              |
| [16]      | 2015 | 28       | Hashing based (paralleling)        | 400K            | ○                 | ○                              |
| [17]      | 2016 | 29       | Universal hashing                  | 400K            | ○                 | ○                              |
| [11]      | 2018 | 5        | Z3 + Eq. 1 flipping                | 400K            | ○                 | ○                              |
| SNAP      | 2019 | this paper   | Z3 + Eq. 1 + local sampling        | 400K            | ●                 | ●                              |

○ / ● : the absence / presence of corresponding item  ● : only partial case studies (the small case studies) were reported.
for generating test cases using theorem provers is QuickSampler [1]. QuickSampler was evaluated on large real-world case studies, some of which have more than 400K variables. At ICSE’18, it was shown that QuickSampler outperforms Unigen2 as well as another similar technique named SearchTreeSampler [14]. QuickSampler starts from a set of valid solutions generated by Z3. Next, it computes the differences between the solutions using Eq. [1]. New test cases generated in this manner are not guaranteed to be valid. According to Dutra et al.’s experiments, the percent of valid tests found by QuickSampler can be higher than 70%. The percent of valid tests found by SNAP, on the other hand, is 100%. Further, as shown below, SNAP builds those tests with enough diversity much faster than QuickSampler.

III. ABOUT SNAP

As stated in the introduction, SNAP uses the Z3 theorem prover combined with Eq. [1]. Also, SNAP uses specialized sub-sampling heuristics to avoid redundant tests. Just to say the obvious, we have no formal proofs that any of the following are useful. Instead, these heuristics are based on hunches we acquired while working with QuickSampler.

Heuristic #1: Instead of computing some deltas between many tests, SNAP restrains mutation to many deltas between a few tests. Specifically, SNAP builds a pool of 10,000 deltas from \( N = 100 \) valid tests (note that this process requires calling the theorem prover only \( N = 100 \) times). SNAP uses this pool as a set of candidate “mutators” for existing tests (and by “mutator”, we mean an operation that converts an existing test into a new one).

Heuristic #2: SNAP builds new tests by apply Eq. [1] to old tests. To minimize redundancy, SNAP uses old tests that are quite distant. More specifically, SNAP uses the centroids found after applying a \( k = 5 \)-means clustering algorithm to the \( N = 100 \) initial samples.

Heuristic #3: We have an intuition that the more frequently we see a delta, the more likely it might represent a valid change to a test. Hence, when SNAP mutates our centroids, it uses deltas that are seen most frequently.

Heuristic #4: We have another intuition that test cases that pass verification and somehow less interesting than those that fail. Hence, when SNAP finds a new failing test, it repairs it (using the process described below) and focuses the rest of the test generation on that harder case.

Heuristic #5: Z3 is much slower for generating new tests than repairing invalid tests than for verifying that a test is valid. As discussed in the introduction, the reason for this is that the search space of options is much larger for generation that for repairing than for verification. Hence, SNAP needs to verify more than it repairs (and also do repairs more than generating new tests).

Algorithm [1] shows how SNAP uses all these heuristics. In this algorithm, each test is a zero or one (false, true) for all the variables in the CNF of our case studies.

SNAP uses the Z3 theorem prover for steps 1a,3biii, and 3biv. As required by Heuristic #5, SNAP performs verification

---

**Algorithm 1: SNAP**

0) **Set up**
   a) let \( N = 100 \); i.e. initial sample size;
   b) let \( k = 5 \); i.e. number of clusters;
   c) let \( suite = \emptyset \); i.e. the output test suite;
   d) let \( samples = \emptyset \); i.e. a temporary work space.

1) **Initial samples generation:**
   a) Add \( N \) solutions (from Z3) to \( samples \)
   b) Put all \( samples \) into \( suite \) (since they are valid)

2) **Delta preparation** (heuristic#1):
   a) Find delta \( \delta = (a \oplus b) \) for all \( a, b \in samples \).
   b) Weight each delta by how often it repeats

3) **Sampling**
   a) Find \( k \) centroids in \( samples \) using \( k \)-means (heuristic#2):
   b) For each centroid \( c \), repeat \( N \) times:
      i) Select stochastically two deltas \( \delta_i, \delta_j \) at probability equal to their weight (heuristic#3).
      ii) Get new candidate via \( c \oplus (\delta_i \lor \delta_j) \)
      iii) Verify new candidate using Z3:
      iv) If invalid, then repair using Z3 (see III-A) and add to sample (heuristic#4).
      v) Add new to \( suite \);

4) **Loop or terminate:**
   a) If diversity improving (see §III-B), go to step 2.
   b) Else terminate, returning \( suite \).
more often than repair, which in turn is performed far more often than generation:

- The call to Z3 in step 1a can be the slowest (since this a generate call that must navigate all the constraints of our CNF). Hence, we only do this $N = 100$ times;
- The call to Z3 in step 3biii verification call is much faster since all the variables are set;
- The call of Z3 in the step 3biv repair call, is a little slower than step 3biii since (as discussed below), our repair operator introduces some open choices into the test. But note that we only need to repair the minority of new tests that fail verification. How small is that minority? Later in this paper, we can use Fig. 3 to show that repairs are only needed on 30% (median) of all tests.

Algorithm 1 requires a repair function for step 3biv, and a termination function for step 4a. Those two functions are discussed below.

A. Implementing “Repair”

When the new test (found in step3ii) is invalid, SNAP uses Z3 to repair that test. As mentioned in the introduction, SNAP’s repair function deletes potentially “dubious” parts of a test case, then calls Z3 to fill in the missing details. In this way, when we repair a test, most of the bits are set and Z3 only has to search a small space.

To find the “dubious” section, we reflect on how step 3bii operates. Recall that the new test is $c \oplus \delta$ where $\delta = a \oplus b$ and $a, b$ are valid tests taken from samples. Since $a, b$ were valid tests, then the “dubious” parts of the test are anything that was not seen in both $a$ and $b$. Hence, we preserve the bits in $c \oplus \delta$ bits (where the corresponding $\delta$ bit was 1), while removing all other bits (where $\delta$ bit was 0). For example:

- Assuming we are mutating $c = (1, 0, 0, 1, 0, 0, 0)$ using $\delta = (1, 0, 1, 0, 1, 0, 1)$. If $c \oplus \delta = (0, 0, 1, 1, 0, 0, 1, 0)$ is invalid, then SNAP deletes the “dubious” sections as follows.
- SNAP preserves any “1” bits that were seen in $\delta$.
- SNAP deletes the other bits; i.e. the 2, 4, 6, 8th bits ($0, 0, 1, 1, 0, 0, 1, 0$).
- Z3 is then called to fill out the missing bits of (0?1?0?1?).

Heuristic #5 (shown above) is based on the assumption that these last step (where Z3 repairs the vector) is usually faster than generating a completely new solution from scratch.

B. Implementing “Termination”

To implement SNAP’s termination criteria (step 4a), we need a working measure of diversity. Recall from the introduction that one test suite is more diverse than another if it uses more of the variable settings with disjunctions inside the CNF. Diverse test suites are better since they cover more parts of the code.

To measure diversity, we used the normalized compression distance (NCD) proposed by Feldt et al. [24]. Feldt et al. showed that a test suite with high NCD implies higher code coverage during the testing.

NCD is based on information theory – the Kolmogorov complexity [25] of a binary string $x$ is the length of the shortest program that outputs $x$. NCD is based on the observation that the degree to which a string can be compressed by real-world compression programs (such as gzip or bzip2). SNAP uses gzip to estimate Kolmogorov complexity. Let $C(x)$ be the length for the compression of $x$ and $C(X)$ be the compression length of binary string set $X$’s concatenation. The NCD of $X$ is defined as

$$NCD(X) = \frac{C(X) - \min_{x \in X} \{C(x)\}}{\max_{x \in X} \{C(X \setminus \{x\})\}} \quad (2)$$

SNAP uses NCD as follows. This algorithm terminates when improvement of NCD was less than $X = 5\%$ within $T = 10$ minutes.

C. Engineering Choices

Our implementation used the following control parameters that were set via engineering judgment:

- $X = 5\%$;
- $T = 10$ minutes;
- $N = 100$ samples;
- $k = 5$ clusters.

In future work, it could be insightful to vary these values.

Another area that might bear further investigation is the clustering method used in step 3a. For this paper, we tried different clustering methods. Clustering ran so fast that we were not motivated to explore alternate algorithms. Also, we found that the details of the clustering were less important than pruning away most of the items within each cluster (so that we only mutate the centroid).

IV. EXPERIMENTAL SET-UP

A. Code

To explore the research questions shown in the introduction, the SNAP system shown in Algorithm 1 was implemented in C++ using Z3 v4.8.4 (the latest release when the experiment was conducted). A $k$-means cluster was added using the free edition of ALGLIB [26], a numerical analysis and data processing library delivered for free under GPL or Personal/Academic license. QuickSampler does not integrate the samples verification into the workflow. Hence, in the experiment, we adjusted the workflow of QuickSampler so that all samples are verified before termination. Also, the outputs of QuickSampler were the assignments of independent support. The independent support is a subset of variables which completely determines all the assignments to a formula [1]. In practice, engineers need the complete test case input; consequently, for valid samples, we extended the QuickSampler to get full assignments of all variables from independent support’s assignment via propagation.

3Just as an aside, we note that we did not adopt the diversity metric (distribution of samples displayed as histogram) from [1], [16] since computing that metric is very time-consuming. For the case studies of this paper, that calculation required days of CPU.
B. Experimental Rig

We compared SNAP to the state-of-the-art QuickSampler, technique purposed by Dutra et al. at ICSE’18. To ensure a repeatable result, we updated the Z3 solver in QuickSampler into the latest version.

To reduce the observation error and test the performance robustness, we repeated all experiment 30 times with 30 different random seeds. To simulate real practice, such random seeds were used in Z3 solver (for initial solution generation), ALGLIB (for the k-means) and other components. Due to the space limitation, we cannot report results for all 30 repeats. Correspondingly we report the medium or the IQR (75-25th variations) results.

All experiments were conducted in the machines with Xeon-E5@2GHz and 4GB memory, running CentOS. These were multi-core machines but for systems reasons, we only used one core per machine.

C. Case Studies

Table II lists the attributes of all the case studies used in this work. We can see that number of variables ranges from hundreds to more than 486K. The large examples have more than 50K clauses, which is very huge. For exposition purposes, we divided the case studies into three groups: the small case studies with vars < 6K; the medium case studies with 6K < vars < 12K and the large case studies with vars > 12K.

For the following reasons, our case studies are the same as those used in the QuickSampler paper:

• We wanted to compare our method to QuickSampler over same case studies;
• Their case studies were online available;
• Their case studies are used in multiple papers [1], [16] etc.

These case studies are representative of scenarios engineers met in software testing or circuit testing in embedded system design. They include bit-blasted versions of SMTLib case studies, ISCAS89 circuits augmented with parity conditions on randomly chosen subsets of outputs and next-state variables, problems arising from automated program synthesis and constraints arising in bounded theorem proving. For more introduction of the case studies, please see [1], [17] etc.

For pragmatic reasons, certain case studies were omitted from our study. For example, we do not report on diagStencilClean.sk or diagStencilClean.sk_41_36 in the experiment, since the purpose of this paper is to sample a set of valid solutions to meet the diversity requirement; while there are only 13 valid solutions from this model. The QuickSampler spent 20 minutes (on average) to search for one solution.

Also, we do report on the case studies marked with a star(∗) in Table II. Based on the experiment, we found that even though the QuickSampler generates tens of millions of samples for these examples, all samples were the assignment to the independent support (defined in §IV-A). The omission of these case studies is not a critical issue. Solving or sampling these examples is not difficult; since they are all very small, as compared to other larger case studies.

V. RESULTS

The rest of this paper use the machinery defined above to answer the four research questions posed in the introduction.

A. RQ1: How Reliable is the Eq. 1 Heuristic?

QuickSampler ran so quickly since it assumed that tests generated using Eq. 1 did not need verification. This research question checks that assumption, as follows.

For each case study, we randomly generated 100 valid solutions, \( S = \{s_1, s_2, \ldots, s_{100}\} \) using Z3. Next, we selected three \( \{a, b, c\} \subseteq S \) and built a new test case using Eq. 1 i.e. \( new = c \oplus (a \oplus b) \).

Fig. 2 reveals the number of identical deltas seen within these 100² of deltas. Among all case studies, we rarely found large sets of unique deltas. This means that among the 100 valid solutions given by Z3, many \( \delta s \) were shared within various pairwise solutions. This is important since, if otherwise, the Eq. 1 heuristic would be dubious.

The percentage of these deltas that proved to be valid in step3biii of Algorithm 1 are shown in Fig. 3 Dutra et al.’s estimate were that the percentage of valid tests generated by Eq. 1 was usually 70% or more. As shown by the median

| Size      | Case studies | Vars | Clauses |
|-----------|--------------|------|---------|
| Small     | blasted_case47 | 118  | 328     |
|           | blasted_case110 | 287  | 1263    |
|           | s820a_7_4     | 616  | 1703    |
|           | s820a_15_7    | 685  | 1987    |
|           | s1238a_3_2    | 685  | 1850    |
|           | s1196a_3_2    | 689  | 1805    |
|           | s832a_15_7    | 693  | 2017    |
|           | blasted_case1_b12_2* | 827  | 2725    |
|           | blasted_squaring16* | 1627 | 5835    |
|           | blasted_squaring7* | 1628 | 5837    |
|           | 70.sk_3_40    | 4669 | 15864   |
|           | ProcessBean.sk_8_64 | 4767 | 14458   |
|           | 56.sk_6_38    | 4836 | 17828   |
|           | 35.sk_3_52    | 4894 | 10547   |
|           | 80.sk_2_48    | 4963 | 17060   |
| Medium    | 7.sk_4_50     | 6674 | 24816   |
|           | doublyLinkedList.sk_8_37 | 6889 | 26918   |
|           | 19.sk_3_48    | 6984 | 23867   |
|           | 29.sk_3_45    | 8857 | 31557   |
|           | isolateRightmost.sk_7_481 | 10024 | 35275   |
|           | 17.sk_3_45    | 10081 | 27056   |
|           | 81.sk_5_51    | 10764 | 38006   |
|           | LoginService2.sk_23_36 | 11510 | 41411   |
| Large     | sort.sk_8_32  | 12224 | 49611   |
|           | parity.sk_11_11 | 13151 | 47506   |
|           | 77.sk_3_44    | 14524 | 27573   |
|           | 20.sk_1_51    | 15465 | 60994   |
|           | enqueueSeqSK.sk_10_42 | 16465 | 58515   |
|           | karatsuba.sk_7_41 | 19593 | 82417   |
|           | tutorial3.sk_4_31 | 486193 | 2598178  |
values of Fig. 3, this was indeed the case. However, we also see that in lower third of those results, the percent of valid tests generated by Eq. 1 is very low: 25% to 50% (median to max). This result alone would be enough to make us cautious about using QuickSampler since, when the Eq. 1 heuristics fails, it seems to fail very badly. We recommend:

**Conclusion #1:** Eq. 1 should not be used without a verification of the resulting test.

By way of comparisons, it is useful to add here that SNAP verifies every test case it generates. This is practical for SNAP, but impractical for QuickSampler since these two systems typically process $10^3$ to $10^8$ test cases, respectively. In any case, another reason to recommend SNAP over QuickSampler is that the former delivers tests suites where 100% of all tests are valid.

**B. RQ2: How Diverse are the SNAP Test Cases?**

As stated in our introduction, diverse test suites are better since they cover more parts of the code. A concern with SNAP is that, since it explores fewer tests than QuickSampler its tests suites could be far less diverse.

Fig. 4 compares the diversity of the test suites generated by our two systems. These results are expressed as ratios of the observed NCD values. Results less than one indicate that SNAP’s test suites are less diverse than QuickSampler.

In that figure, we see that occasionally, SNAP’s faster test suite generation means that the resulting test suites are much less diverse (see s1238_3_2 and parity.sk_11_11). That said, while QuickSampler’s tests are more diverse, the overall difference is usually very small.

Also, RQ1 showed us that many of the QuickSampler tests are invalid. This means that the diversity numbers reported for


**Fig. 5:** RQ3 results: Time to terminated (seconds). The y-axis is in log scale. The SNAP sampling time for $s1238a_3_2$ and $parity.sk_{11_11}$ is not reported since their achieved NCD were much worse than QuickSampler's (see Fig. 4). Fig. 6 illustrates the corresponding speedups.

QuickSampler are somewhat inflated since invalid tests would not enter the branches they are meant to cover. Hence, overall, we say:

**Conclusion #2:** The diversity of SNAP’s test suites are not markedly worse than those of QuickSampler.

### C. RQ3: How Fast is SNAP?

Fig. 5 shows the execution time required for SNAP and QuickSampler. The y-axis of this plot is a log-scale and shows time in seconds. These results are shown in the same order as Table II. That is, from left to right, these case studies grow from around 300 to around 3,000,000 clauses.

For the smaller case studies, shown on the left, SNAP is sometimes slower than QuickSampler. Moving left to right, from smaller to larger case studies, it can be seen that SNAP often terminates much faster than QuickSampler.

Fig. 6 is a summary of Fig. 5 that divides the execution time for both systems. From this figure it can be seen:

**Conclusion #3:** SNAP was 10 to 3000 times faster than QuickSampler (median to max).

There are some exceptions to this conclusion, where QuickSampler was faster than SNAP (see the right-hand-side of Fig. 5). We note that in most of those cases, those models are small (17,000 clauses or less). For medium to larger models, with 20,000 to 2.5 million clauses, SNAP is nearly always orders of magnitude faster than QuickSampler.

**D. RQ4: How Easy is it to Apply SNAP’s Test Cases?**

Finally, we end on a pragmatic note. The smaller a test suite, the easier it is for programmers to run those tests. Therefore it is important to ask which method produces fewer tests: QuickSampler or SNAP?

Table III compares the number of tests (different suggested inputs) generated by QuickSampler and SNAP. As shown by the last column in that table,

**Conclusion #4:** SNAP’s test cases were 10 to 750 times smaller than those of QuickSampler (median to max).

Hence, we argue that it would be easiest for an industrial practitioner to execute and maintain the test suites generated by SNAP.

### VI. DISCUSSION

**A. Why does SNAP Work?**

This section reflects on the success of SNAP. Specifically, we ask why can SNAP generate comparable diversity to QuickSampler? We conjecture that, for SE problems:
TABLE III: RQ4: results. Number of unique valid cases in the test suite. Case studies are sorted by number of variables. Same color scheme as Table II.

| Case studies          | $S_N$ | $S_Q$ | $S_Q/S_N$ |
|-----------------------|-------|-------|-----------|
| blasted_case100       | 3005  | 4270  | 1.40      |
| tutorial3.sk_4_31     | 336   | 2953  | 8.79      |
| 81.sk_5_51            | 227   | 2814  | 12.40     |
| sort.sk_8_52          | 812   | 10184 | 12.54     |
| karatsuba.sk_7_41     | 139   | 4210  | 30.29     |
| 20.sk_1_51            | 239   | 10039 | 42.00     |
| doublyLinkedList      | 278   | 12042 | 43.32     |
| 17.sk_3_45            | 228   | 12780 | 56.05     |
| ProcessBean           | 1193  | 75392 | 63.20     |
| 7.sk_4_50             | 258   | 18090 | 70.12     |
| 56.sk_6_38            | 1827  | 149031| 81.57     |
| 80.sk_2_48            | 653   | 54440 | 83.37     |
| 77.sk_3_44            | 245   | 33858 | 138.20    |
| 35.sk_3_52            | 258   | 193920| 751.63    |

Combining solutions from local samples leads to diverse global solutions.

$S_N$ is an example of such a “local sampling”. Consider how it executes: each round of the sampling (step 3 in Algorithm 1) focuses on a local sample (generated by the clustering method). All these local samples then combined into a global test suite. The success of $S_N$’s local sampling strategy may be a comment on the nature of software systems. Large SE systems are the result of much work by many teams. In such systems, small parts of software combine into some greater whole. For such systems, local sampling (as done in $S_N$) would be useful.

There is much other evidence of the benefits of local sampling in SE. Chen et al. proposed a local sampling technique called SWAY [27], [28] that finds optimized configuration for agile requirements engineering [29]. Subsequently, it was seen that the same local sampling approach leads to a new high-watermark in optimizing cloud container deployments [30]. All of that researches used the same strategy (local sampling) and had similar results (orders of magnitude improvement on the prior state-of-the-art).

Based on this, we make two observations. Firstly, local sampling is a strategy that may be very useful in many future SE applications. Secondly, if SE problems have such a structure (of larger problems composed of numerous smaller ones), then it would not be insightful to assess new methods using randomly generated solutions (e.g. to assess $S_N$ using randomly generated CNF formula). We say this since such randomly generated models may not contain the structures that have been found so useful in local sampling methods like $S_N$ and elsewhere [27], [28], [30].

B. Threats to Validity

One threat to validity of this work is the baseline bias. Indeed, there are many other sampling techniques, or solvers, that $S_N$ might be compared to. However, our goal here was to compare $S_N$ to a recent state-of-the-art result from ICSE’18. In further work, we will compare $S_N$ to other methods.

A second threat to validity is internal bias that raises from the stochastic nature of sampling techniques. $S_N$ requires many random operations. To mitigate the threats, we repeated the experiments for 30 times and reported the medium or IQR of those results.

A third threat is the measurement bias. To determine the diversity of a test suite, in the experiment, we use normalized compression distance (NCD). Prior research has argued for the value of that measure [24]. However, there exist many other diversity measurements for the theorem proving problem and changing the diversity measurement might lead to change of the results. That said, in one research report, it is impossible to explore all options. For the convenience of further exploration, we have released the source code of $S_N$ in the hope that other researchers will assist us by evaluating $S_N$ on a broader range of measures.

Another threat is hyperparameter bias. The hyperparameter is the set of configurations for the algorithm. For $S_N$, we need to use a set of control parameters shown in [31]–[33] which might be useful for finding better settings for $S_N$. This is a clear direction for future work.

VII. Conclusion

The experiments of this paper suggest that $S_N$ is a better test suite generation system than QuickSampler. Our system avoids much of the redundant reasoning that:

- Slows up QuickSampler by a factor of 10 to 3000 (compared to $S_N$);
- And which also results in test suites that are 10 to 750 times larger than they need to be (again, compared to $S_N$).

Another reason to prefer $S_N$ is that since its test suites are so small, we can run verification on 100% of all of $S_N$’s test. That is, unlike QuickSampler, all our tests are known to be valid.

$S_N$ has its drawbacks. Specifically, the diversity of its test suites are not always as good as QuickSampler. That said, this difference in diversity is so small (see Fig. 4) that overall, given all the other advantages of $S_N$, we can still recommend it.

As shown by the Algorithm 1 pseudocode, $S_N$ is relatively simple to implement. Hence, if nothing else, we can
recommend SNAP as a baseline method against which more elaborate methods might be benchmarked.

REFERENCES

[1] R. Dutra, K. Laeufer, J. Bachrach, and K. Sen, “Efficient sampling of sat solutions for testing,” in 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). IEEE, 2018, pp. 549–559.

[2] R. Baldoni, E. Coppa, D. C. Delia, C. Demetrescu, and I. Finocchi, “A survey of symbolic execution techniques,” ACM Computing Surveys (CSUR), vol. 51, no. 3, p. 50, 2018.

[3] M. Christakis, P. Müller, and W. Wüthrich, “Guiding dynamic symbolic execution toward unverified program executions,” in Proceedings of the 38th International Conference on Software Engineering. ACM, 2016, pp. 144–155.

[4] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2008, pp. 337–340.

[5] A. Biere, “Picosat essentials,” Journal on Satisfiability, Boolean Modeling and Computation, vol. 4, pp. 75–97, 2008.

[6] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, “The mathsat 4 smt solver,” in International Conference on Computer Aided Verification. Springer, 2008, pp. 299–303.

[7] N. Bjørner, A.-D. Phan, and L. Fleckenstein, “rz-an optimizing smt solver,” in International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2015, pp. 194–199.

[8] M. Finke, “Equisatisfiable sat encodings of arithmetical operations,” Online] http://www.martin-finke.de/documents/Masterarbeit_Bitblast_Finke.pdf, 2015.

[9] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz, “Modeling design constraints and biasing in simulation using bdds,” in Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design. IEEE Press, 1999, pp. 584–590.

[10] M. A. Iyer, “Race: A word-level atpg-based constraints solver system for smart random simulation,” in null. IEEE, 2003, p. 299.

[11] J. Yuan, A. Aziz, C. Pixley, and K. Albin, “Simplifying boolean constraint solving for random simulation-vector generation,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 23, no. 3, pp. 412–420, 2004.

[12] W. Wei, J. Erenrich, and B. Selman, “Towards efficient sampling: Exploiting random walk strategies,” in AAAI, vol. 4, 2004, pp. 670–676.

[13] V. Gogate and R. Dechter, “Samplesearch: Importance sampling in exploiting random walk strategies,” Artificial Intelligence, vol. 175, no. 2, pp. 694–729, 2011.

[14] S. Erron, C. P. Gomes, and B. Selman, “Uniform solution sampling using a constraint solver as an oracle,” arXiv preprint arXiv:1210.4861, 2012.

[15] S. Chakraborty, K. S. Meel, and M. Y. Vardi, “Balancing scalability and uniformity in sat witness generator,” in Proceedings of the 51st Annual Design Automation Conference. ACM, 2014, pp. 1–6.

[16] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, “On parallel scalable uniform sat witness generation,” in International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2015, pp. 304–319.

[17] K. S. Meel, M. Y. Vardi, S. Chakraborty, D. J. Fremont, S. A. Seshia, D. Fried, A. Ivrii, and S. Malik, “Constrained sampling and counting: Universal hashing meets sat solving,” in Workshops at the thirteenth AAAI conference on artificial intelligence, 2016.

[18] C. A. Tovey, “A simplified np-complete satisfiability problem,” Discrete applied mathematics, vol. 8, no. 1, pp. 85–89, 1984.

[19] X. Jia, C. Ghezzi, and S. Ying, “Enhancing reuse of constraint solutions to improve symbolic execution,” in Proceedings of the 2015 International Symposium on Software Testing and Analysis. ACM, 2015, pp. 177–187.

[20] W. Visser, C. S. Pseanu, and R. Pelánek, “Test input generation for java containers using state matching,” in Proceedings of the 2006 international symposium on Software testing and analysis. ACM, 2006, pp. 37–48.

[21] S. B. Akers, “Binary decision diagrams,” IEEE Transactions on computers, no. 6, pp. 509–516, 1978.

[22] B. Selman, H. A. Kautz, B. Cohen et al., “Local search strategies for satisfiability testing,” Cliques, coloring, and satisfiability, vol. 26, pp. 521–532, 1993.

[23] Y. Mansour, N. Nisan, and P. Tiwari, “The computational complexity of universal hashing,” Theoretical Computer Science, vol. 107, no. 1, pp. 121–133, 1993.

[24] R. Feldt, S. Pouliot, D. Clark, and S. Yoo, “Test set diameter: Quantifying the diversity of sets of test cases,” in 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST). IEEE, 2016, pp. 223–233.

[25] M. Li and P. Vitányi, An introduction to Kolmogorov complexity and its applications. Springer Science & Business Media, 2013.

[26] S. Bochkanov and V. Byistritsky, “Aglib,” Available from: www.aglib.net, vol. 59, 2013.

[27] J. Chen, V. Nair, R. Krishna, and T. Menzies, “‘Sampling’ as a baseline optimizer for search-based software engineering,” IEEE Transactions on Software Engineering, 2018.

[28] J. Chen, V. Nair, and T. Menzies, “Beyond evolutionary algorithms for search-based software engineering.” Information and Software Technology, vol. 95, pp. 281–294, 2018.

[29] T. Menzies and J. Richardson, “Xomo: Understanding development options for autonomy,” in COCOMO forum, vol. 2005, 2005.

[30] J. Chen and T. Menzies, “Ri: A stochastic-based method for workflow scheduling in the cloud,” in 2018 IEEE 11th International Conference on Cloud Computing (CLOUD). IEEE, 2018, pp. 318–325.

[31] W. Fu and T. Menzies, “Revisiting unsupervised learning for defect prediction,” in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, 2017, pp. 72–83.

[32] T. Xia, R. Krishna, J. Chen, G. Mathew, X. Shen, and T. Menzies, “Hyperparameter optimization for effort estimation,” arXiv preprint arXiv:1805.00336, 2018.

[33] V. Nair, A. Agrawal, J. Chen, W. Fu, G. Mathew, T. Menzies, L. Minku, M. Wagner, and Z. Yu, “Data-driven search-based software engineering,” in 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR). IEEE, 2018, pp. 341–352.

10