GenTree: Inferring Configuration Interactions using Decision Trees

KimHao Nguyen
University of Nebraska-Lincoln, USA
kdnguyen@cse.unl.edu

ThanhVu Nguyen
George Mason University, USA
tvn@gmu.edu

***Abstract*—In this paper, we demonstrate the implementation details and usage of GenTree, a dynamic analysis tool for learning a program's interactions. Configurable software systems, while providing more flexibility to the users, are harder to develop, test, and analyze. GenTree can efficiently analyze the *interactions* among configuration options in configurable software. These interactions compactly represent large sets of configurations and thus allow us to efficiently analyze and discover interesting properties (e.g., bugs) in configurable software. Our experiments on 17 configurable systems spanning 4 languages show that GenTree efficiently finds precise interactions using a tiny fraction of the configuration space. GenTree and its dataset are open source and available at https://github.com/unsat/gentree and a video demo is at https://youtu.be/x3eqUflvlN8.**

I. Introduction

Modern software systems are increasingly designed to be configurable. This has many benefits, but also significantly complicates tasks such as testing, debugging, and analysis due to the number of configurations that can be exponentially large—in the worst case, every combination of option settings can lead to a distinct behavior [1].

Existing works on highly-configurable systems [1]–[4] showed that we can automatically find *interactions* to concisely describe the configuration space of the system. These works define an interaction for a location as a logically weakest formula over configuration options such that any configuration satisfying that formula would cover that location. Such interactions can help understand the configurations of the system, e.g., determine what configuration settings cover a given location; determine what locations a given interaction covers; find important options, and compute a minimal set of configurations to achieve certain coverage; etc.

While existing interaction techniques are useful, they have several limitations. The symbolic analysis in [2] does not scale to large systems, even when being restricted to boolean configuration options, and is language-specific (C programs). iTree [3], [4] uses decision trees to generate configurations to maximize coverage, but achieves few and imprecise interactions. Both of these works only focus on interactions that can be represented as purely conjunctive formulae. The dynamic approach iGen [1] can infer interactions that are purely conjunctive, purely disjunctive, and specific mixtures of the two. However, these interactions are still too limited to capture complex interactions in real-world systems.

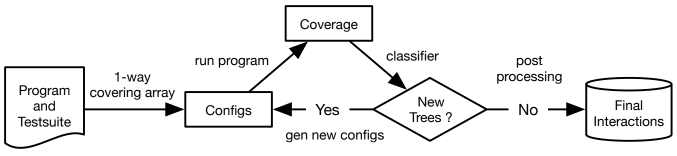

Fig. 1: GenTree overview

We present GenTree, an iterative "guess-and-check" approach to automatically learn, check, and refine program's interactions. GenTree runs the program under a small sample of configurations to obtain coverage data; uses a custom C5.0 classifying algorithm on these data to build decision trees representing interaction candidates; and then analyzes the trees to generate new configurations to further refine the trees and interactions in the next iteration.

The GenTree approach has several benefits. By using dynamic analysis, GenTree is language agnostic and supports complex programs (e.g., those using third party libraries) that might be difficult for static analyses. By considering only small configuration samples, GenTree is efficient and scales well to large programs. By integrating with iterative refinement, GenTree generates small sets of useful configurations to gradually improve its results. By using decision trees, GenTree supports expressive interactions representing arbitrary boolean formulae and allows for generating effective counterexample configurations. Finally, by using a classification algorithm customized for interactions, GenTree can build trees from small data samples to represent accurate interactions.

The source code and dataset of GenTree are publicly available at https://github.com/unsat/gentree. The full details of the GenTree approach are available in the research paper [5]. In this tool paper, we present more in-depth the technical details and usage of GenTree, not only as an implementation of GenTree algorithm but also as a generic platform to develop and test data-driven approaches for interaction learning.

II. GenTree Approach

GenTree takes as input a program and returns interactions that are *arbitrary* boolean formula over the program options mapping to program locations. Figure 1 gives an overview of the main components in GenTree:

```
//9 configuration options:
//s, t, u, v (bool);
      a, b, c, d, e ∈ {0, 1, 2}

printf ("L0\n"); // True

if  (a ≡ 1 ∨ b ≡ 2) {
  //a ≡ 1 ∨ b ≡ 2
  printf ("L1\n");
}
else if  (c ≡ 0 ∧ d ≡ 1) {
  //a ∈ {0, 2} ∧ b ∈ {0, 1}
  //∧c ≡ 0 ∧ d ≡ 1
  printf ("L2\n");
}

if (u ∧ v) {
  printf ("L3\n"); //u ∧ v
  return;
}
else {
  printf ("L4\n"); //ū ∨ v̄
  if  (s ∧ e ≡ 2){
    //s ∧ e ≡ 2 ∧ (ū ∨ v̄)
    printf ("L5\n");
    return;
  }
}

//(s̄ ∨ e ∈ {0, 1}) ∧ (ū ∨ v̄)
printf ("L6\n");

if (e ≡ 2) {
  //s̄ ∧ e ≡ 2 ∧ (ū ∨ v̄)
  printf ("L7\n");
  if (u ∨ v) {
    //s̄ ∧ e ≡ 2 ∧ ((u ∧ v̄) ∨ (ū ∧ v))
    printf ("L8\n");
  }
}
```

Fig. 2: A program with 9 locations L0–L8 annotated with interactions

Initial Configurations: `GenTree` first uses a random 1-way covering array [6] to obtain a set of initial configurations, which contains all possible settings of each individual option.

Decision Trees: For each covered location l, `GenTree` uses a classification algorithm called $\mathrm{C5_i}$, developed specifically for this work, to build a decision tree representing the interaction for l. To build the tree for l, $\mathrm{C5_i}$ uses two sets of data: the *hit* sets consisting of configurations covering l and the *miss* set consisting of configurations not covering l. $\mathrm{C5_i}$ only uses a small sample of configurations to build a decision tree and thus could be inaccurate when classifying unseen configurations.

New Configurations: `GenTree` next attempts to create new configurations to refine the tree representing the interaction for location l. A *hit* path on a decision tree for location l is a path from the root to a leaf with hit label (i.e., covers l), and similarly, a *miss* path is a path lead to a leaf with miss label (i.e., does not cover l). Observe that if a hit path is precise, then any configuration satisfying its condition would cover l (similarly, any configuration satisfying the condition of a miss path would not cover l). Thus, we can validate a path by generating configurations satisfying its condition and checking the coverage. Configurations generated from a hit (or miss) path that do not (or do) cover l are *counterexample* configurations, which show the imprecision of the path condition and help build a more precise tree in the next iteration.

Next Iterations: `GenTree` repeats the process of building trees and generating new configurations until it cannot generate new coverage or refine existing trees for several consecutive iterations. In a postprocessing step, `GenTree` combines the hit path conditions of the decision tree for each location l into a logical formula representing the interaction for l.

Example: We illustrate the expressive power of `GenTree` using the C program in Figure 2. This program has nine configuration options listed on the first line of the figure.

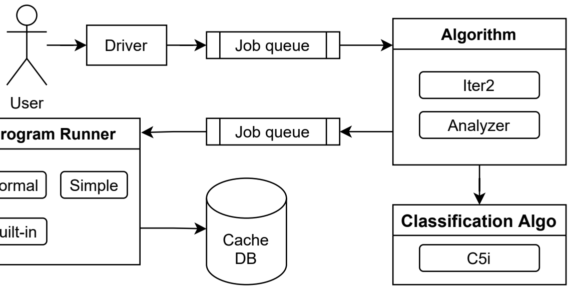

Fig. 3: `GenTree` components

The print statements mark nine interesting locations $L0$–$L8$. At each location, we list the associated desired interaction. For example, $L5$ is covered by any configurations in which s is `true`, e is 2, and either u or v is false.

For this example, `GenTree` found the correct interactions for all locations within eight iterations and under a second. The table below shows the number of iterations and configurations used to find the interaction for each location. For example, the interaction $\bar{s} \wedge e \equiv 2 \wedge ((u \wedge \bar{v}) \vee (\bar{u} \wedge v))$ of $L8$ took 58 configurations and is discovered at iteration 4, and the interaction `true` of $L0$ was quickly discovered from the initial configurations. Overall, `GenTree` found all of these interactions by analyzing approximately 360 configurations (median over 11 runs) out of 3888 possible ones.

	L0	L1	L2	L3	L4	L5	L6	L7	L8
Iter. Found	1	2	6	1	2	5	3	3	4
# Configs	3	27	144	15	30	123	50	47	58

Existing interaction analyses fail to capture these interactions. The works on symbolic configuration analysis [2] and iTree [3], [4] only support conjunctions and therefore cannot generate the correct interactions for any line except $L2$ and $L3$. The iGen dynamic analysis [1] also cannot generate the complex disjunctive interactions for $L6$ and $L8$.

III. Implementation

`GenTree` is implemented in C++ as a modular framework for developing and testing data-driven interaction learning approaches. The implementation uses standard C++ libraries and relies on Z3 SMT solver [7] to encode and simplify interactions.

Figure 3 provides the overview of the design of `GenTree`. `GenTree` heavily utilizes multicore processing to speed up the execution time. `GenTree` divides jobs into threads using a *job queue*, which is a shared FIFO queue. We push jobs that can be run in parallel into a shared job queue, and a set of worker threads pop jobs from the queue and process them one by one. `GenTree` consists of four main components:

a) Driver: The driver parses user input (as CLI flags), performs common initializations and task distributions such as pushing tasks to job queue to run in parallel, analyzes, and outputs results.

b) Algorithm: An *algorithm* is a self-contained unit of work in GenTree. It could be an interaction learning algorithm or an analyzer that read the interaction results and do various analyses on them. An algorithm is explicitly designed to be able to run in parallel, hence all of its global states are stored in a *context* that can be accessed by objects belonging to that algorithm, but not other ones. Currenly, there are two algorithms in the tool:

Iter2: This implements the main GenTree algorithm. It supports dynamic configuration via a Javascript (JS) script, which takes a subject program information (e.g., configuration space) and tunes the GenTree algorithm parameters accordingly. Using a scripting language provides flexibility to quickly adjust parameters without recompiling GenTree. Aside from the core algorithm, *Iter2* can also (i) find ground truth interactions by brute forcing all possible configurations and (ii) build decision trees from all random configurations (instead of from the counterexample generation heuristic by GenTree).

Analyzer: This analyzes the results of GenTree, compares with other results, and computes various statistics. It also produces tables and graphs for the evaluation section in [5]. It can (i) compare two outputs (e.g., GenTree outputs versus the ground truth) for discrepancies, (ii) do random testing: continuously run the subject programs with random configurations and check if the interactions produced by GenTree match the actual execution traces, (iii) compute interactions statistic and report number interaction types (pure conjunctive, pure disjunctive, mixed) and various metrics (number of configuration used, number of locations, running time, etc.), and (iv) find a small set of configurations that cover all reachable locations using GenTree interactions result.

c) Program Runner: Run the program and collect the coverage information. The *algorithm* treats program runner as a black-box function, with input is a configuration, and output is the set of covered locations. The program runner is pluggable and supports different program types: normal programs (currently supports C, Python, Perl, and OCaml programs), simple programs (programs output their covered locations to stdout), Otter [8] programs, and built-in programs. Under the hood, the program runner is parallelized using a job queue. It also has an embedded key/value database (RocksDB) to cache the coverage results: querying the database is much faster than running the program and the cached coverage results are deterministic, which makes debugging much easier.

Normal programs: running and collecting coverage information requires many external dependencies: the test suite, the subject program, the interpreter and virtual environment (e.g., for Python), and the location coverage collection tool. We developed a simple DSL to specify the dependencies and runtime arguments for the testing programs. Figure 4 is a run script for pylint program. Each line is a command with structure <cmd> <arguments>. For example, the command var name pylint sets variable name with value pylint. Variable names inside curly bracket, e.g., {name}, will be substituted by their value. The DSL is designed to be simple

```
include conf.pycov
var name pylint

cov_arg --include={site-packages}/pylint/*
loc_trim_prefix {site-packages}/pylint/

bin {bindir}/{name}
# ================================
clean_wd
run {} {testdir}/py_pylint/alg_igen.py
```

Fig. 4: Config script for pylint

```
FN(testprog,
    VARS(a, b, c, d, e),
    DOMS(2, 2, 2, 3, 3), {
        if ((a || b) && (c || d)) LOC("L1");
        else LOC("L2");
    })
```

Fig. 5: A built-in program named testprog

but expressive enough to reduce the configuration boilerplate and flexible enough to support different run scenarios. More examples on how to configure the normal program runner could be found in the benchmarks folder in the GitHub repository at [9].

Simple programs: The program outputs its covered locations to the standard output, separated by the newline character. If a program is not directly supported by the normal programs mode (C, Python, Ocaml, Perl), we can use the simple program runner, with the entry point is a Python or Shell script that invokes the actual program and collects the coverage information, and then print to the standard output.

Built-in programs: Built directly into GenTree. We use C++ macros to create a domain-specific language (DSL) to specify simple test programs. Figure 5 is a simple builtin program testprog with 5 options a, b, c, d, e, with domains $0 \leq a, b, c < 2$ and $0 \leq d, e < 3$. The built-in programs greatly reduce the algorithms development cycle because it is faster than running an external program, and everything is self-contained in GenTree executable.

d) Classification Algorithm: Build decision trees from a set of *hit* and *miss* configurations using C5$_\mathbf{i}$, a customized classification algorithm described in [5]. The implementation is deterministic and highly optimized. Aside from the tree-building algorithm, it also contains some useful primitives to work with the tree: find leaves with less supporting configurations, generate counterexamples from a leaf (non-duplicated with all previously generated configurations), serialize and deserialize a tree from text, generate an SMT formula from a tree, and compile a tree into a simple bytecode format to quickly evaluate the tree on any configurations.

IV. Usage

The detailed instructions for obtaining the artifact package and running experiments can be found in [10]. Figure 6 illustrates how to install (step 1), run GenTree (steps 2 and 3), and examines its results (step 4).

```
# Step 1: Pull Docker image
docker pull unsatx/gentree_docker:icse21

# Step 2: Run container
docker run −it −−rm −−tmpfs /mnt/ramdisk \
    unsatx/gentree_docker:icse21 bash

# Step 3: Run GenTree (inside container)
cd ~/gentree/wd
./gt −J2 −cx −BF @ex_paper # example
./gt −J2 −cx −BF @ex_paper −−full # example (ground truth)
./gt −J2 −cx −GF 2/id      # coreutils id (C)
./gt −J2 −cx −YF 2/vsftpd # vsftpd (Otter)

# Step 4: Using the Analyzer
./gt −J2 −cxw −j4 −GF 2/ln −−full −O full.txt
./gt −J2 −cr −j4 −GF 2/ln −O out.txt
./gt −A0 −T0 −GF 2/ln −v10 −I full.txt,out.txt # compare
```

Fig. 6: Commands to install GenTree

Steps 3 in Figure 6 shows how to run GenTree on several example programs. @ex_paper is the example C program listed in Figure 2. id and vsftpd are subject programs listed in Table 1 in [5]. By default, GenTree outputs result interactions to the standard output. To output to a file instead, use the flag -O <filename>. Step 3 demonstrates how to run GenTree on built-in programs (@ex_paper), normal programs (id), and Otter programs (vsftpd).

Step 4 in Figure 6 shows how use GenTree to analyze the results. First, we run an exhaustive search to find the ground truth in full.txt. Then, we run GenTree and write results to out.txt. Finally, we use the analyze to compare GenTree results with the ground truth. The analyzer output could be ln.c:495 diff (cex = 1), which means GenTree interaction for location ln.c:495 is different than the ground truth (with one counterexample). In fact, ln.c:495 is a long disjunction over all configurations and GenTree infers the interaction true.

GenTree's performance can be controlled by various parameters. The -j4 flag makes the program runner run at most 4 subject programs in parallel. The -c flag controls the program runner database (-cxw runs the program and caches the coverage data and -cr make GenTree only uses the cached data). GenTree runs almost deterministically[1] when running in -cr mode.

For better interoperability, the GenTree output is designed to be both human and machine-readable. For each discovered interaction, GenTree outputs a block similar to Figure 7. Lines started with the character '#' are comments and should be ignored while parsing. In Figure 7, the comments tell us how many *hit* and *miss* configurations classified by the decision tree, which iteration the tree was last rebuilt, and how many configurations were used to build the tree. The first component in the block is a list of locations covered by the interaction (e.g., $L4$). The second component is an SMT-LIB 2.0 [11] formula generated by Z3 [7]. The last component is a serialized decision tree (pre-order traversal) for internal usage.

```
# M/H: 51 / 137
# Last rebuild:    iter 2  num_configs 30
L4,
−
(or (= u |0|) (= v |0|))
−
3 H 4 HM
```

Fig. 7: GenTree's output for the interaction $\bar{u} \vee \bar{v}$ in the example program in Figure 2 in [5]

V. Evaluation

We evaluated GenTree on 17 programs in C, Python, Perl, and OCaml [5]. Our results show that GenTree generates precise interaction results (similar to what GenTree would produce if it inferred interactions from all possible configurations) and scales extremely well (works with real-world programs with *hundreds of trillion* configurations) because it only explores a small fraction of the large configuration spaces.

Results from GenTree also confirmed several observations made by prior work (e.g., conjunctive interactions are common but disjunctive and mixed interactions are still important for coverage; and enabling options, which must be set a certain way to cover most locations, are common). We also found that complex interactions supported by GenTree, but not from prior works, cover a non-trivial number of locations and are critical to understanding the program behaviors at these locations.

Acknowledgment

We thank the anonymous reviewers for their helpful comments. This work was supported in part by awards W911NF19-1-0054 from the Army Research Office; CCF-1948536 from the National Science Foundation; and a Faculty Seed award from the University of Nebraska-Lincoln (UNL). KimHao Nguyen is also supported by a UCARE Award from UNL.

References

[1] T. Nguyen, U. Koc, J. Cheng, J. S. Foster, and A. A. Porter, "iGen: Dynamic interaction inference for configurable software," in *Foundations of Software Engineering*, 2016, pp. 655–665.

[2] E. Reisner, C. Song, K. Ma, J. S. Foster, and A. Porter, "Using symbolic evaluation to understand behavior in configurable software systems," in *International Conference on Software Engineering*. ACM, 2010, pp. 445–454.

[3] C. Song, A. Porter, and J. S. Foster, "iTree: Efficiently discovering high-coverage configurations using interaction trees," in *International Conference on Software Engineering*, 2012, pp. 903–913.

[4] ——, "iTree: Efficiently discovering high-coverage configurations using interaction trees," *Transactions on Software Engineering*, vol. 40, no. 3, pp. 251–265, 2014.

[5] K. Nguyen and T. Nguyen, "Gentree: Using decision trees to learn interactions for configurable software," in *International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1598–1609.

[1]All GenTree code are determistic, but the Z3 solver could give non-deterministic results. However, we have not encountered any non-determinism while developing GenTree.

[6] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn, "Constructing test suites for interaction testing," in *International Conference on Software Engineering*. IEEE, 2003, pp. 38–48.

[7] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[8] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *Static Analysis Symposium*. Springer, 2011, pp. 95–111.

[9] K. Nguyen and T. Nguyen, "GenTree," 2021, accessed on 2021-02-01. [Online]. Available: https://github.com/unsat/gentree

[10] ——, "Artifact for GenTree: Using Decision Trees to Learn Interactions for Configurable Software," in *International Conference on Software Engineering*. IEEE, 2021, pp. 177–178.

[11] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," Department of Computer Science, The University of Iowa, Tech. Rep., 2010, available at `www.SMT-LIB.org`.