TestMC: Testing Model Counters using Differential and Metamorphic Testing

Muhammad Usman
University of Texas at Austin, USA
muhammadusman@utexas.edu

Wenxi Wang
University of Texas at Austin, USA
wenxiw@utexas.edu

Sarfraz Khurshid
University of Texas at Austin, USA
khurshid@ece.utexas.edu

ABSTRACT

Model counting is the problem for finding the number of solutions to a formula over a bounded universe. This is a classic problem in computer science that has seen many recent advances in techniques and tools that tackle it. These advances have led to applications of model counting in many domains, e.g., quantitative program analysis, reliability, and security. Given the sheer complexity of the underlying problem, today's model counters employ sophisticated algorithms and heuristics, which result in complex tools that must be heavily optimized. Therefore, establishing the correctness of implementations of model counters necessitates rigorous testing. This experience paper presents an empirical study on testing industrial strength model counters by applying the principles of differential and metamorphic testing together with bounded exhaustive input generation and input minimization. We embody these principles in the TestMC framework, and apply it to test four model counters, including three state-of-the-art model counters from three different classes. Specifically, we test the exact model counters projMC and dSharp, the probabilistic exact model counter Ganak, and the probabilistic approximate model counter ApproxMC. As subjects, we use three complementary test suites of input formulas. One suite consists of larger formulas that are derived from a wide range of real-world software design problems. The second suite consists of a bounded exhaustive set of small formulas that TestMC generated. The third suite consists of formulas generated using an off-the-shelf CNF fuzzer. TestMC found bugs in three of the four subject model counters. The bugs led to crashes, segmentation faults, incorrect model counts, and resource exhaustion by the solvers. Two of the tools were corrected subsequent to the bug reports we submitted based on our study, whereas the bugs we reported in the third tool were deemed by the tool authors to not require a fix.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Empirical software validation**.

KEYWORDS

Model counting, metamorphic testing, differential testing, delta debugging

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416563

ACM Reference Format:
Muhammad Usman, Wenxi Wang, and Sarfraz Khurshid. 2020. TestMC: Testing Model Counters using Differential and Metamorphic Testing. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3324884.3416563

1 INTRODUCTION

Model counting is the problem for finding the number of solutions to a formula over a bounded universe. This is a classic problem in computer science, which generalizes the satisfiability problem, i.e., whether a formula has a solution or not. Recent years have seen many advances in techniques and tools for model counting [20]. These advances have led to many applications of model counting in various domains, e.g., quantitative program analysis [44], reliability [26], and security [7]. To handle the huge complexity of the underlying problem, today's model counters employ sophisticated algorithms and heuristics, which result in complex tools that must be heavily optimized for high efficiency. Therefore, establishing the correctness of implementations of model counters – even if they are proven correct on paper – necessitates rigorous testing.

This experience paper presents an empirical study on testing industrial strength model counters by applying the principles of four well-studied testing approaches: 1) bounded exhaustive input generation where the system under test is tested against all non-equivalent inputs within a bound on the input size [50]; 2) differential testing where the outputs of multiple systems under test are compared to detect faulty behaviors when some outputs do not match [36]; 3) metamorphic testing where metamorphic relations among results for different inputs are used as surrogate for test oracles [47]; and 4) input minimization using delta debugging where inputs that cause failures are minimized to create smaller fault-revealing inputs [58]. We embody these principles in the TestMC framework, and apply it to test a variety of model counters.

Our specific focus is the common class of model counters for propositional logic. Propositional model counting is a *#P-complete* problem, which generalizes the propositional satisfiability (SAT) problem [12], and hence is both highly useful and extremely expensive to solve in practice. It has many applications such as Bayesian belief networks, knowledge compilation, planning, and combinatorial designs [22, 24, 45]. Propositional model counters take as input formulas in *conjunctive normal form* (CNF), just as SAT solvers [23]. A CNF formula is a conjunction (logical and) of *clauses*, where each clause is a disjunction (logical or) of *literals*, where each literal is a boolean variable or its negation. The standard CNF file format that commonly used model counters and SAT solvers support is the DIMACS format [9], which is a text file where each clause is essentially a list of literals on a new line.

TestMC's test generation module implements a dedicated generator for CNF files in the DIMACS format. Due to the simplicity of the DIMACS format, generating CNF files is relatively straightforward, e.g., a non-deterministic generator that chooses the number of clauses, number of variables, and the literals in each clause suffice. To optimize generation, our generator first creates a set of unique clauses based on the desired number of variables, then creates CNF formulas by selecting subsets of the clauses with respect to the desired maximum number of clauses, and finally writes each formula to a file. Due to the nature of bounded exhaustive testing, even for small bounds, there can be a fairly large number of CNF formulas. For example, for a bound of 4 variables (i.e., 8 literals) and 4 clauses, there are 6.65 million unique CNF formulas (modulo re-ordering of clauses within a formula).

We apply TestMC to test four model counters from three general classes of model counters. The model counters are: projMC [34] and dSharp [37], which are *exact* model counters; Ganak [48], which is a *probabilistic exact* model counter; and ApproxMC [18], which is a *probabilistic approximate* model counter. Three of the select model counters, namely projMC, Ganak, and ApproxMC, are the current state-of-the-art in their respective classes.

Each of the four model counters we choose supports *projected model counting*, where the solver allows computing the counts with respect to a given subset of variables, rather than the default of all variables. Support for projected model counting is particularly important for computing solutions that are relevant to the problem domain because the original problem rarely exists in CNF, and translation to CNF typically introduces auxiliary variables and results in a formula that is *equisatisfiable* but not equivalent (and can have a different number of solutions from the original formula with the differences being on the values of the auxiliary variables that exist only in CNF).

Since a model counter's output is a non-negative integer, differential testing mostly requires just a simple integer comparison. Since our test subjects include a probabilistic exact and a probabilistic approximate counter, the comparator we define allows defining a tolerance threshold for the equality check. To complement differential testing, especially in cases when differential testing detects a discrepancy between the counts of model counters or only one model counter produces a result and all others timeout, we define a sanity check and four metamorphic relations that are based on propositional equivalences, primary variables, and formula simplification.

As subjects, we use three test suites of input formulas. One suite consists of significantly larger formulas that are derived from a wide range of software design problems (SDP) that are part of the standard distribution of the well-known Alloy tool-set [31]. The second suite includes a bounded exhaustive set of small formulas that TestMC generated. The third suite contains the formulas generated by an off-the-shelf CNF fuzzer [17].

TestMC found bugs in three model counters – projMC, dSharp, and Ganak. The bugs caused four different kinds of failures: crashes, segmentation faults, incorrect results, and resource exhaustion. The fault revealing input formulas were minimized using delta debugging, and form a part of the bug reports that were submitted to the tools' authors. We submitted bug reports for all the three tools that TestMC reported as faulty. Two of the tools (projMC and Ganak) were corrected subsequent to our bug reports. The bugs in the third tool (dSharp) were deemed by the authors to not necessitate a fix.

This paper makes the following contributions:

- **Framework**. We introduce TestMC, the first framework for testing the functional correctness of model counters. Our framework performs bounded-exhaustive test input generation using a dedicated CNF formula generator, differential testing using a special purpose comparator, metamorphic testing using a family of metamorphic relations for model counters, and input minimization using delta debugging. TestMC defines one sanity check and four metamorphic relations as test oracles. The sanity check and all four metamorphic relations utilize domain knowledge, specifically the fact that we are testing model counters for formulas in propositional logic. Therefore, these metamorphic relations can also be utilized for SAT problems. We employ TestMC to generate a corpus of 6.65 million CNF formulas, which serve as bounded exhaustive test suites that can be used by other model counters and SAT solvers.
- **Study**. We use TestMC to test four model counters, including two exact model counters, one probabilistic exact model counter, and one probabilistic approximate model counter. We test the model counters against three suites of subjects: a) 203 significantly larger CNF subjects derived from a wide class of software models; b) the bounded exhaustive suite of 6.65 million small CNF subjects generated by TestMC; and c) large number of CNF subjects (10 million) generated using an off-the-shelf CNF fuzzer.
- **Lessons Learned**. We present a set of lessons learned during our experience of testing model counters.

With model counters and other backend constraint solvers becoming more and more complex, there is an increasing need for thoroughly testing their functionality, especially as their use increases in critical domains, such as security. We believe our work provides a practical framework that is based on well-understood testing techniques and can handle real-world tools. We hope our experience with TestMC proves valuable in the development of correct model counters and solvers, and for more effective deployment of model counters.

2 EXAMPLES

This section presents small illustrative examples to describe the basics of CNF formulas, and the model counting and projected model counting problems. We also present some small CNF formulas as illustrative examples that were generated by TestMC and uncovered bugs in some of the model counters we tested.

2.1 CNF and model counting

Consider the boolean formula "$f = (a \vee \neg b) \wedge (\neg a \vee b)$", where a and b are boolean variables. Since f has 2 variables, there are a total of 4 possible assignments to a and b: 1) $a = false, b = false$; 2) $a = false, b = true$; 3) $a = true, b = false$; and 4) $a = true, b = true$. Of these, the following 2 assignments are solutions, i.e., assignments such that f is *true*: 1) $a = true, b = true$; and 2) $a = false, b = false$.

Since f is a conjunction of 2 clauses, f is already in CNF. The following file shows an encoding of f in the DIMACS CNF format:

```
p cnf 2 2
1 -2 0
-1 2 0
```

The first line, which begins with the character "p" defines the type of formula (i.e., cnf), the number of variables (i.e., 2), and the number of clauses (i.e., 2) respectively. Each subsequent line of text shown includes a clause. The variables are identified by positive integer ids. A positive literal is just a variable, and represented by the corresponding variable's id; a negative literal is the negation of a variable, and represented by the negation of the corresponding integer id. Each clause ends with the number 0. For example, representing variable "a" as number "1" and "b" as number 2, the clause "$(a \vee \neg b)$" is represented in CNF as "1 -2 0". Comments may be included in a CNF in lines that start with the character "c". Given this input CNF file for formula f, all four model counters that we use as test subjects, namely ApproxMC, dSharp, Ganak, and projMC report 2 as the model count, which is the correct result.

2.2 Projected model counting

We next illustrate the *projected* model counting problem [8], where the model count for the input formula is with respect to a subset, say W, of the set containing all variables, say V, in the formula, so only solutions that differ on at least one value of some variable in W are considered unique. The variables in W are termed as *primary* variables.

Consider the formula $g = (a \wedge b) \vee (c \wedge d)$, which is not in CNF. We can use logical equivalences to translate this formula into CNF, e.g., as $(a \vee c) \wedge (a \vee d) \wedge (b \vee c) \wedge (b \vee d)$. While using logical equivalences suffices to translate any propositional formula into CNF, such translation can cause an exponential increase in the formula size; to avoid such increase, practical tools use alternative translations, which introduce new variables and preserve the formula's satisfiability but may not preserve its solution count [8]. To illustrate, g can alternatively be translated by introducing auxiliary variables u and v as follows: $h = (u \vee v) \wedge (\neg u \vee a) \wedge (\neg u \vee b) \wedge (\neg v \vee c) \wedge (\neg v \vee d)$. Any solution to h contains a solution to g, and any solution to g can be extended to form a solution for h. However, the total number of solutions for g is 7, and the total number of solutions for h is 9, i.e., g and h have different model counts. Indeed, the space of candidate solutions for g and h have different sizes, which are $2^4 = 16$ and $2^6 = 64$ respectively.

The support for projected model counting in modern model counters makes utilizing them much more feasible since they can be employed simply by translating to CNF without worrying about the translation creating an exponentially longer formula, or the model counter providing an inaccurate count due to auxiliary variables. As an illustration of projected model counting, consider using projMC for computing the projected model count for formula h with respect to the set of variables in g. The following CNF file represents h:

```
c ind 1 2 3 4 0
p cnf 6 5
5 6 0
-5 1 0
-5 2 0
-6 3 0
-6 4 0
```

where variables a, b, c, d, u, and v are represented using integers 1, 2, 3, 4, 5, and 6 respectively. The comment in the first line (that starts with a "c") uses the format used by ApproxMC and Ganak to specify the list of primary variables, which are also termed *independent* variables; this line states that variables with ids 1, 2, 3, and 4 are primary variables; the line terminates with "0". In contrast to ApproxMC and Ganak, projMC and dSharp require the list of primary variables to be input separately from the CNF formula. For convenience, we show examples where the primary variables are listed as a comment as required by ApproxMC and Ganak; when invoking projMC and dSharp, we provide the lists separately as required by them. projMC reports 7 as the model count, which is correct and the same as the model count for the formula g.

2.3 Example failures in Ganak, dSharp and projMC

We next show four CNF formulas that lead to failures in the Ganak, dSharp and projMC model counters. Specifically, we show the *smallest* CNF formulas that exhibit these failures.

Example failure (incorrect count) in Ganak on a formula generated by the TestMC bounded-exhaustive input generator

```
c ind 1 0
p cnf 2 1
1 2 0
```

This formula is a disjunction of two variables, and there is only one primary variable (that has id "1"). The formula has three solutions but only two of them are unique with respect to just the primary variable. For this input formula, Ganak incorrectly reports the model count as 3. In contrast, the three other model counters we tested report the correct count of 2.

Example failure (incorrect count) in dSharp on a formula generated by the TestMC bounded-exhaustive input generator

```
c ind 1 0
p cnf 1 2
1 0
-1 0
```

This formula has one variable and two clauses and represents a contradiction since it is a conjunction of the variable and its negation. The only variable in the formula is the primary variable. dSharp outputs a model count of 1, which is wrong since the formula is unsatisfiable.

Example failure (Assertion Error) in Ganak on a formula generated by the TestMC bounded-exhaustive input generator

It turns out that the above formula also reveals another bug in Ganak. In fact, given this formula, Ganak gives an assertion error. In contrast, the other two model counters, projMC and ApproxMC, produce the correct model count of 0 for this formula.

```
linux:~/Desktop/ganak/build$ python3 ganak.py -p exampleb.cnf
rm: cannot remove 'mis.out': No such file or directory
rm: cannot remove 'mis.timeout': No such file or directory
c Outputting solution to console
c GANAK version 1.0.0
The value of delta is 0.05
The value of hashrange is 64x1
```

```
ganak: /.../Desktop/ganak/src/instance.h:215: ClauseIndex
Instance::addClause(std::vector<LiteralID>&):
Assertion `!isUnitClause(literals[0].neg())' failed.
The total user time taken by ganak is:  0.0
```

Example failure (seg fault) in projMC on a formula derived from a software design problem

```
linux:~/Desktop/author_projmc$ ./projMC_linux example.cnf
-fpv=example.var
c Benchmark Information
c Number of variables: 1794
c Number of clauses: 3020
c Number of literals: 6630
c Integer mode
c
c Option list
c Caching: 1
c Variable heuristic: VSADS
c Phase heuristic: TRUE
c Partitioning heuristic: YES + graph reduction +
equivalence simplication
c
Segmentation fault (core dumped)
```

projMC gives a segmentation fault on this formula. Due to space limitations, this formula together with other example formulas that cause failures are provided at the GitHub repository[1].

3 MODEL COUNTERS UNDER TEST

This section discusses the essential background and gives a brief description of the model counters that form our test subjects. We also state the specific versions of the tools we evaluated.

The state-of-the-art model counters can be classified into three categories: (1) exact model counters that output the exact number of solutions; (2) probabilistic exact model counters that output the counts within a given confidence level; and (3) approximate model counters that give counts within a given confidence level *and* tolerance score. As test subjects, we choose a state-of-the-art model counter in each of the three categories, as well as an earlier model counter that was among the first to support projected model counting. Specifically, we choose the following four model counters: probabilistic approximate model counter ApproxMC [18] (Section 3.1), the probabilistic exact model counter Ganak [48] (Section 3.2) and the exact model counters dSharp [37] (Section 3.3) and projMC [34] (Section 3.4). All four of these model counters support projected model counting.

3.1 ApproxMC

ApproxMC [18] is a state-of-the-art approximate model counter that is now in its third generation. The key idea behind ApproxMC is to employ *universal hashing* to iteratively partition the solution space into smaller regions that contain approximately the same number of solutions by adding XOR constraints, and finally count the number of solutions in one region, and compute the estimate for the full space. A special SAT solver called CryptoMiniSAT [49], which supports XOR constraints, is iteratively invoked by ApproxMC. The use of universal hash functions provides theoretical guarantees for the counting approximation. Moreover, the number of SAT calls was reduced from the initial $O(n)$ to $O(log(n))$ using the dependency information among different SAT calls. Also, a new architecture for efficiently solving XOR constraints which are the representation of the hash function was proposed more recently. Past experiments showed the model counting by ApproxMC was efficient as well as remarkably close to the exact counts. For our experiments, we use the latest version of ApproxMC[2] (ApproxMCv3, Git commit ID 23d4439) with the default seed (i.e., 1).

3.2 Ganak

Ganak [48] outputs the projected model counts within the given confidence level. Ganak is built on top of sharpSAT [52]. Ganak relies on probabilistic component caching and universal hashing which significantly improve the counting performance. Moreover, it employs a number of heuristics on top of sharpSAT including probabilistic component caching, new variable branching heuristic, new phase selection heuristic, independent support heuristic, exponentially decaying randomness heuristic, and learn and start-over heuristic. In our evaluation, we used Ganak[3] (Git commit ID 3620813).

3.3 dSharp

dSharp [37] is among the first tools for projected model counting. Similar to Ganak, it builds upon earlier work on sharpSAT [52], and introduces four more components namely dynamic decomposition, implicit binary constraint propagation (IBCP), conflict analysis, and component caching. Dynamic decomposition breaks down the problem into two components during the search and then adds each component as a child to an "AND" node. IBCP is proposed as a look-ahead strategy to select the unassigned variable and evaluate the impact of the assignment in advance. Conflict analysis is applied for non-chronological backtracking and learning mechanism. Finally, dSharp utilizes component caching where the broken components are stored for fast retrieval. In addition to these four components, it also added support for solving deterministic decomposable negation normal form (d-DNNF) formulas which are translated into directed acyclic graphs. We used dSharp-ASP[4] (BitBucket commit ID 791668a) for experiments.

3.4 projMC

projMC [34] is a state-of-the-art exact model counter that uses a recursive algorithm defined for deterministic disjunctive form. projMC first creates partitions of the original CNF formula such that they are pairwise variable independent using disjunctive decomposition. It then calculates the number of partitions and solves each partition recursively by checking if there is a satisfiable solution. If so, the counts of each partition are combined as a total model count of the problem; otherwise, it reports a contradiction. projMC[5] is available as an executable file[6].

4 TESTMC FRAMEWORK

This section introduces our TestMC framework for testing model counters and describes its key components.

[1] https://github.com/muhammadusman93/TestMC-ASE2020

[2] https://github.com/meelgroup/ApproxMC
[3] https://github.com/meelgroup/ganak
[4] https://bitbucket.org/haz/dsharp-asp/src/default/
[5] projMC is proprietary software. Hence, there is no Git commit ID.
[6] http://www.cril.univ-artois.fr/kc/projmc.html

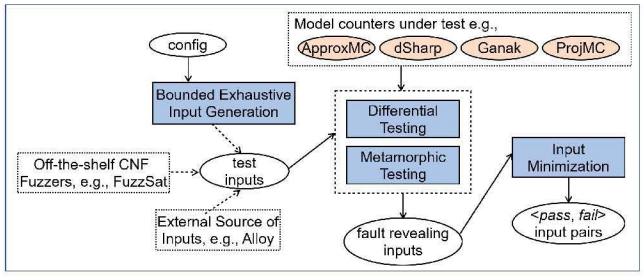

Figure 1: TestMC framework. config is the maximum number of boolean variables and clauses to define the size bound for bounded exhaustive input generator.

```
1  CNF generate(int maxVars, int maxClauses) {
2      Set<Integer> literals = createAllLiterals(maxVars);
3      Set<Clause> allClauses = createAllClauses(literals);
4      allClauses = reduce(allClauses);
5      int numClauses = choose(1, maxClauses);
6      CNFBody body = chooseSubset(numClauses, allClauses);
7      int projVars = choose(1, maxVarId(body));
8      CNF formula = addProjVarInfo(projVars, body);
9      return formula;
10 }
```

Figure 2: TestMC bounded exhaustive generation algorithm.

4.1 Overview

Figure 1 shows an overall architecture of the TestMC framework. There are four basic components: (1) bounded exhaustive test generation (Section 4.2); (2) differential testing (Section 4.3); (3) metamorphic testing (Section 4.4); and (4) input minimization with delta debugging (Section 4.5). To test the input model counters, TestMC uses a pool of CNF formulas that serve as test inputs to the model counters. TestMC uses differential testing and metamorphic testing to determine test failures. Specifically, for each CNF input in the pool, TestMC invokes each model counter under test and compares their results (with respect to a threshold on the allowed difference). In cases where differential testing detects a discrepancy between the counts of model counters or only one counter returns a result and the others timeout, metamorphic testing helps to validate the returned count. Once failures are found, TestMC uses input minimization based on delta debugging [57] to reduce the fault-revealing inputs.

The current pool of CNF input files contains formulas from three sources. One source is CNF formulas translated from software design problems (SDP) that are included in the standard distribution of the well-known Alloy tool-set [1, 32], the second source is our bounded-exhaustive generator, and the third source is CNF formulas generated using an off-the-shelf CNF fuzzer [17]. Alloy allows building and analyzing designs of software systems, and has been used in many applications, including analyzing distributed algorithms [56] and finding security bugs [39]. Alloy's backend analyzer translates Alloy designs to CNF formulas, and employs off-the-shelf SAT solvers, which allows Alloy users to check desired properties of their designs. The Alloy distribution includes a variety of real-world problems and hence provides a valuable source of CNFs as test inputs. The CNF fuzzer allows to randomly create a large number of CNF formulas and TestMC bounded exhaustive generator creates formulas that are helpful in checking corner cases and giving developers small fault revealing CNF formulas which makes it easier for them to debug their tools.

4.2 Bounded Exhaustive Generation

We designed and implemented a dedicated generator for creating bounded-exhaustive test suites consisting of files in CNF format for testing model counters that support computing projected model counts. Figure 2 shows our input generation algorithm in Java-like pseudo-code. The input `maxVars` and `maxClauses` define the maximum number of boolean variables and the maximum number of clauses respectively to define the size bound for the inputs generated. The helper method `createAllLiterals` returns a set that contains all literals with respect to the input number of variables. In general, for n variables, there are $2n$ literals. The helper method `createAllClauses` returns a set of all clauses that can be formed using a subset of the input set of literals where each clause is viewed as *set* of literals. For example, the CNF clauses "1 -2 0" and "-2 1 0" are considered to be the same and only one of them is generated. Breaking such symmetries is essential for applying bounded-exhaustive testing – even for small bounds. The helper method `reduce` allows removing some clauses from the set, which are considered not interesting; at present, we remove clauses that contain a tautology, i.e., a variable and its negation since that clause is always satisfied regardless of the assignment to that variable.

The helper method `choose` represents *non-deterministic* choice (implemented similar to `Verify.getInt(,)` function of the Java Path Finder [30]) that selects a number between 1 and `maxClauses` (inclusive); thus, `numClauses` is the number of clauses that the generated formula will contain. The helper method `chooseSubset` also represents non-deterministic choice, which chooses a subset of size `numClauses` from the set `allClauses`. In general, for x literals, there are $2^x - 1$ clauses that contain at least one literal. Selecting a *set* of clauses allows the algorithm to break more symmetries as two CNF formulas that differ only by the *order* in which the clauses appear in the formula are considered the same and only one of them will be generated. Next, the algorithm non-deterministically chooses a number of variables that are primary variables. Finally, the CNF formula is initialized and returned.

To see an illustration of the test generation algorithm, assume `maxVars` = 2 and `maxClauses` = 3. Then, `literals` = {1, 2, -1, -2}, and there are 15 ($2^4 - 1$) clauses that can be formed as subsets of these 4 literals:

```
Clause  1 = {1}          Clause  2 = {2}
Clause  3 = {-1}         Clause  4 = {-2}
Clause  5 = {1,2}        Clause  6 = {1,-1}
Clause  7 = {1,-2}       Clause  8 = {2,-1}
Clause  9 = {2,-2}       Clause 10 = {-1,-2}
Clause 11 = {1,2,-1}     Clause 12 = {1,2,-2}
Clause 13 = {1,-1,-2}    Clause 14 = {2,-1,-2}
Clause 15 = {1,2,-1,-2}
```

The algorithm prunes clause 6, 9, 11, 12, 13, 14 and 15 since they contain a tautology. This leaves 8 clauses that are used for generating CNF formulas. The algorithm creates all subsets of up to 3 clauses. There are $\binom{8}{1} = 8$ subsets of size 1; $\binom{8}{2} = 28$ subsets of size 2; and $\binom{8}{3} = 56$ subsets of size 3. Thus, there are 92 unique subsets with 1, 2, or 3 clauses. Since there are 2 variables in total we can project on one of them or on both of them; if the CNF formula only has one variable, we only use that as the primary variable. The total number of CNF formulas for the size bound `maxVars` = 2 and `maxClauses` = 3 is 181. For testing the subject model counters in this paper, we set the size bound to `maxVars` = 4 and `maxClauses` = 4. There are 255 ($2^8 - 1$) clauses that can be formed as subsets of 8 literals. The algorithm prunes 175 clauses which contain a tautology, which leaves 80 clauses that are used for generating CNF formulas. The algorithm creates all subsets of up to 4 clauses. There are $\binom{80}{1} = 80$ subsets of size 1; $\binom{80}{2} = 3160$ subsets of size 2; $\binom{80}{3} = 82160$ subsets of size 3; and $\binom{80}{4} = 1581580$ subsets of size 4. Thus, there are 1,666,980 unique subsets with 1, 2, 3 or 4 clauses. Since there are 4 variables in total, we can have up to 4 projection variables for a formula. Note that the number of projection variables is always ≤ the number of variables in the formula. It gives 6.65 million CNF formulas.

4.3 Differential Testing

For differential testing, TestMC checks for differences in the model counters' outputs; if they differ, it checks if a majority agrees and if it does, the minority is considered as likely faulty. Since our test subjects include a probabilistic exact and a probabilistic approximate counter, the comparator we define allows the user to define a tolerance for the equality check (only for ApproxMC and Ganak). For our experiments, we set the tolerance to 10% (two outputs mismatch if they are not within 10% of each other) based on a recent study [55].

4.4 Metamorphic Testing

To complement differential testing, especially in cases when differential testing detects a discrepancy between the counts of model counters or only one model counter produces a result and all others timeout, we define one sanity check and four metamorphic relations as test oracles. The sanity check and all four relations utilize domain knowledge, specifically the fact that we are testing model counters for formulas in propositional logic.

Let f be a formula in propositional logic with n variables, so the space of all possible boolean assignments to the variables has size 2^n. Assume there are k primary variables where $k \leq n$; assume (without loss of generality) the primary variable ids are $1, \ldots, k$. Let $mc(f)$ represent the model count for f. Let $pmc(f, k)$ represent the projected model count for f with respect to the k primary variables. Let $f_{x=v}$ be f where the variable x is assigned the boolean value v.

Sanity Check (SC1): $pmc(f, k) \leq 2^k$, i.e., the projected model count over k variables must be no more than 2^k. It is simple to evaluate this relation: take log_2 of the tool's output and compare it with k; if k is smaller, the tool's output is faulty. SC1 does not require any additional invocation of the model counter.

Metamorphic relation 1 (MR1): $pmc(f, k) \geq pmc(f_{v=true}, k)$ where v is a variable in f, i.e., the model count for formula f is greater or equal to the model count for *reduced* formula $f_{v=true}$, i.e., f where v is set to true. Given variable v, we can create the CNF formula for $f_{v=true}$ simply by adding the clause "`v 0`" to the original CNF for f. In our experiments we set v to 1. MR1 requires one additional invocation of the model counter.

Metamorphic relation 2 (MR2): $pmc(f, k) = pmc(f_{v=true}, k) + pmc(f_{v=false}, k)$ where v is a variable in f, i.e., the model count for formula f is the sum of the model counts for *reduced* formulas $f_{v=true}$, i.e., f where v is set to true, and $f_{v=false}$, i.e., f where v is set to false. Given variable v, we can create the CNF formulas for $f_{v=true}$ and $f_{v=false}$ simply by adding the clauses "`v 0`" and "`-v 0`" respectively to the original CNF for f. In our experiments we set v to 1. MR2 requires two additional invocations of the model counter.

Metamorphic relation 3 (MR3): Let y be a clause in f and z be a literal in y. Let y' be a clause containing all literals originally in y and literal z added again. Thus, replacing y with y' should have no effect on the original formula f (y' is equivalent to y). In this case, $pmc(f, k)$=$pmc(f - y + y', k)$, i.e., the count of original formula should be equal to the count of the original formula with y being removed and y' added. MR3 requires one additional invocation of the model counter.

Metamorphic relation 4 (MR4): Let y' be a clause containing a tautology. Let f' be another formula which includes all the clauses in formula f and one more clause y' (clause containing tautology). Thus, both the original formula f and the new formula f' should have the same count since adding a tautology to the original formula should have no effect on the model count. In this case, $pmc(f, k)$=$pmc(f + y', k)$, i.e., the count of the original formula should be equal to the count of the new formula with y' added. MR4 requires one additional invocation of the model counter.

4.5 Input Minimization with Delta Debugging

To facilitate debugging, TestMC supports input minimization using delta debugging for fault-revealing inputs [2]. The simplicity of the CNF file format helps with effective minimization since any subset of the CNF clauses is itself a CNF formula; to create a new CNF file using a subset, we populate the new file with the selected clauses and update the number of clauses listed in the declaration line (that starts with "`p`"). We follow a standard method for input minimization that uses binary search to find a sequence of clauses in the original formula such that the failure recurs [59]. In addition, we create a pair of two CNF files $\langle C, D \rangle$ where C and D include subsets of clauses in the original fault revealing input such that C does not cause a failure, D does cause a failure and D includes all the clauses in C and one additional clause. We expect the pair $\langle C, D \rangle$ to assist in fault localization [43] where the user can compare the two traces on inputs that are almost identical and differ by only one clause.

Table 1: Basic information about the test suite.

		SDPGen	BEGen	FuzzGen
CNF Formulas	Total	203	6,649,854	10,000,000
Total Vars	Min	28	1	2
	Max	93,764	4	34,755
	Avg	4,764.12	3.99	11.39
	Std	9,028.88	0.09	27.56
Primary Vars	Min	3	1	2
	Max	2,048	4	34,755
	Avg	201.72	2.50	11.39
	Std	269.52	1.12	27.56
Total Clauses	Min	31	1	3
	Max	291,349	4	123,891
	Avg	11,066.47	3.95	29.40
	Std	25,640.81	0.23	95.61

5 EXPERIMENTAL EVALUATION

This section describes the test suites employed for testing model counters, evaluates TestMC using three research questions and presents examples of bugs and fixes of the tested model counters.

5.1 Test Suite

This section describes the CNF formulas used by TestMC for testing model counters. The test suites are classified into three categories: a) test suite (denoted as SDPGen) containing 203 large CNF formulas derived from a wide class of software design problems; b) test suite (denoted as BEGen) containing small CNF formulas created by bounded exhaustive generator; and c) test suite (denoted as FuzzGen) containing a large number of formulas with varied scales generated by an off-the-shelf CNF fuzzer called FuzzSAT [17]. Table 1 summarizes detailed information about each test suite, including the total number of test cases, minimum, maximum, average, and standard deviation of the number of total variables, primary variables, and total clauses of the input formula, respectively.

Formulas generated using software design problems (SDPGen) Alloy is able to translate modeled real-world problems to CNF formulas, that are solved by backend SAT solvers for checking desired properties. The Alloy distribution includes a variety of real-world problems and hence provides a valuable source of CNF formulas as a potential test suite. There are 203 large CNF formulas generated from software design problems in Alloy [1, 32]. The number of total variables ranged from a minimum of 28 to a maximum of 93,764, with an average of 4,764.12. The number of primary variables ranged from a minimum of 3 to a maximum of 2,048, with an average of 201.72. The number of clauses ranged from a minimum of 31 to a maximum of 291,349, with an average of 11,066.47.

Formulas generated using a bounded exhaustive generator (BEGen) A total of 6,649,854 (6.65 million approximately) CNF formulas were generated by the bounded-exhaustive generator using the algorithm described in Section 4.2. These formulas are in small scale. The number of total variables ranged from a minimum of 1 to a maximum of 4, with an average of 3.99; the number of primary variables ranged from a minimum of 1 to a maximum of the total number of variables in the formula, with an average of 2.50. The number of clauses ranged from a minimum of 1 to a maximum of 4, with an average of 3.95. These formulas are helpful in checking corner cases and more importantly give developers small fault revealing CNF formulas which makes it easier to debug the tool.

Formulas generated using off-the-shelf CNF fuzzer (FuzzGen) A total of 10,000,000 (10 million) formulas were randomly generated by CNF fuzzer called FuzzSAT [17] for rigorous testing of model counters. The number of total variables ranged from a minimum of 2 to a maximum of 34,755, with an average of 11.39. The number of clauses ranged from a minimum of 3 to a maximum of 123,891, with an average of 29.40. FuzzSAT considers all variables to be primary variables (number of primary variables is always equal to the number of total variables) which may cause the testing on projected model counting ineffective. As part of future work, we plan to add support to FuzzSAT for generating formulas with different sets of primary variables, by selecting them randomly.

The Bounded-Exhaustive Generator as well as all 3 test suites (16.65 million CNF formulas in total) are available at: https://github.com/muhammadusman93/TestMC-ASE2020. All experiments were performed on Ubuntu 16.04 with an Intel Core-i7 8750H CPU (2.20 GHz) and 16GB RAM.

5.2 Research Questions

This section answers following three research questions.

- **RQ1: How Effective is TestMC Differential Testing Module in Finding and Categorizing Bugs/Failures in Model Counters?**
- **RQ2: How Effective are the TestMC Metamorphic Testing Relations in Finding Bugs/Failures in Model Counters?**
- **RQ3: How Effective is Test Input Minimization with Delta Debugging?**

RQ1: How Effective is TestMC Differential Testing Module in Finding and Categorizing Bugs/Failures in Model Counters?

TestMC was able to find 4 different types of bugs/failures in 3 model counters. The first bug type called *WSat* describes that a model counter returns the wrong satisfiability of the input formula, specifically, returns SAT for UNSAT problem. The second bug type called *WCnt* indicates that a model counter gives a wrong count of the formula. If the model counter crashes due to segmentation fault, it is classified as *SegFault* bug. Lastly, if the model counter terminates unexpectedly, it is classified as *Crash* bug. Table 2 summarizes the results of each bug type that happened in each model counter, for each test suite. Since our test subjects include a probabilistic exact and a probabilistic approximate counter, the comparator we define allows the user to define a tolerance for the equality check (only for ApproxMC and Ganak). For our experiments, we set the tolerance to 10% (two outputs mismatch if they are not within 10% of each other) based on a recent study [55].

For Ganak, TestMC found two types of bugs: *WCnt* in 3,996,331 cases and *Crash* in 56,617 cases, which strongly indicates the existence of bugs inside the tool. We found that all of the *Crash* cases in Ganak were assertion failures. For dSharp, TestMC found two types

Table 2: Results of applying differential testing on *Ganak*, *dSharp* and *projMC* for each test suite are shown. ApproxMC is not shown because it did not give faulty results.

Ganak					
Input	**WSat**	**WCnt**	**SegFault**	**Crash**	**Total**
SDPGen	0	77	0	4	81
BEGen	0	3996254	0	48344	4044598
FuzzGen	0	0	0	8269	8269
Total	0	3996331	0	56617	4052948
dSharp					
Input	**WSat**	**WCnt**	**SegFault**	**Crash**	**Total**
SDPGen	0	0	0	18	18
BEGen	48224	0	0	0	48224
FuzzGen	7490	0	0	0	7490
Total	55714	0	0	18	55732
projMC					
Input	**WSat**	**WCnt**	**SegFault**	**Crash**	**Total**
SDPGen	0	0	2	0	2
BEGen	0	0	0	0	0
FuzzGen	0	0	0	0	0
Total	0	0	2	0	2

Table 3: Results of applying Sanity check and Metamorphic relations for formulas on which differential testing detected discrepancy. Total number of these formulas is given in brackets in *Input* column of table. "-" indicates that there was no formula on which differential testing detected discrepancy.

Ganak					
Input	**SC1**	**MR1**	**MR2**	**MR3**	**MR4**
SDPGen (77)	48	15	50	0	0
BEGen(3996254)	2902390	16386	1972934	0	0
FuzzGen(0)	-	-	-	-	-
Total	2902438	16401	1972984	0	0
Percentage	72.63%	0.41%	49.37%	0%	0%
dSharp					
Input	**SC1**	**MR1**	**MR2**	**MR3**	**MR4**
SDPGen(0)	-	-	-	-	-
BEGen(48224)	0	0	13705	0	0
FuzzGen(7490)	0	0	2	0	0
Total	0	0	13707	0	0
Percentage	0%	0%	24.60%	0%	0%

of bugs: *WSat* in 55,714 cases, and *Crash* in 18 cases on which dSharp even made the operating system crash, unfortunately. We investigated the reasons behind the OS crash and found that dSharp does not place any limit on the amount of memory it uses. It exhausts all of the memory resulting in a fatal crash. Lastly, for projMC, TestMC found *SegFault* bug in 2 cases.

Specifically, when tested on *BEGen*, 3,996,254 cases of *WCnt* bug and 48,344 cases of *Crash* bug were found in Ganak; 48,224 cases of *WSat* bug were found in dSharp; and no bugs were found for ApproxMC and projMC. When tested on FuzzGen; 8,269 cases of *Crash* bug were found in Ganak; 7,490 cases of *WSat* bug were found in dSharp; and similar to BEGen no bugs were found in ApproxMC and projMC. When tested on SDPGen, 77 cases of *WCnt* bug were found and 4 cases of *Crash* bug were found in Ganak; 18 cases of *Crash* bug were found in dSharp; and 2 cases of *SegFault* bug were found in projMC; and no bugs were found for ApproxMC. Note that, FuzzGen cannot detect *WCnt* bugs in Ganak while BEGen can. This may be due to the fact that random fuzzers generate CNF formulas by considering all variables as primary variables and hence may not be effective for testing bugs in projected model counting.

```
...
if (literals.size() == 1) {
assert(!isUnitClause(literals[0].neg()));
unit_clauses_.push_back(literals[0]);
return 0; }
...
```

(a) Location of *Crash* bug in Ganak

```
...
if (literals.size() == 1) {
addBinaryClause(literals[0], literals[0]);
return 0; }
...
```

(b) Fix for *Crash* bug for Ganak

```
...
else if(strcmp(argv[i], "-p")==0)
theSolver.config().perform_projectedmodelcounting
     = false;
...
```

(c) Location of *WCnt* bug in Ganak

```
...
else if(strcmp(argv[i], "-p")==0)
theSolver.config().perform_projectedmodelcounting
     = true;
...
```

(d) Fix for *WCnt* bug for Ganak

Figure 3: Fault localization and bug fixes for Ganak

Whereas our bounded exhaustive generator is able to generate all possible combinations of primary variables and thus could be highly effective in finding bugs in projected model counting, regardless of the small scales of the generated formulas. To summarize, we can conclude that *TestMC differential testing module is highly effective in finding and categorizing bugs/failures in model counters.*

RQ2: How Effective are the TestMC Metamorphic Testing Relations in Finding Bugs/Failures in Model Counters?

Unsatisfying metamorphic relations indicate the existence of bugs whereas satisfying metamorphic relations help to narrow down the list of potential bugs. Table 3 shows the results of metamorphic testing on Ganak and dSharp. TestMC only found *SegFault* bugs in projMC so metamorphic testing is not applicable for projMC. For Ganak, SC1 and MR2 proved to be most useful followed by MR1. SC1 was able to find 72.63% of formulas on which TestMC found (during differential testing) *WCnt* bug for Ganak. MR1 was able to find 0.41% and MR2 was able to find 49.37% of such formulas. However, MR3 and MR4 were not able to find any bug. SC1 points out problems w.r.t projected model counting. We can observe that Ganak violated SC1 2902438 times. Similarly, MR2 helps to point out that Ganak is giving larger than expected counts. Code inspection revealed that Ganak is ignoring the primary variables, thereby increasing the state space of the solutions. MR3 is helpful in finding bugs related to repeated literals in a clause and MR4 is good at detecting bugs related to tautological clauses. The reason why MR3 and MR4 failed to find any bugs is because the model counters under test do not have such bug types. However, the usefulness of MR3 and MR4 should not be overlooked.

For dSharp, MR2 proved to be useful since it was able to identify 24.60% of formulas on which TestMC found (during differential testing) *WSat* bug for dSharp. Although no other metamorphic relations were violated by dSharp, these relations were still helpful. For example, having 0 as a count for SC1 shows that TestMC did not find any bugs related to projected model counting in dsharp. This matches our observation in RQ1. There were a total of 15 formulas on which one or more model counters timed-out and MR relations were able to confirm faults in 3 (20 %) of these formulas. Note that most metamorphic relations requires 1 additional invocation of the model counter (for each formula) which is costly. Since our subject model counters include approximate (ApproxMC) and probabilistic exact (Ganak) counters, there can be false positives using metamorphic and differential testing. Therefore, metamorphic testing is applied only in cases where a discrepancy is detected using differential testing. Overall, *the metamorphic relations served useful in finding bugs in model counters.*

RQ3: How Effective is Test Input Minimization with Delta Debugging?

On average, TestMC was able to remove 30% of the clauses. For each of the files on which model counter crashes, TestMC gave a pair of two CNF files $\langle C, D\rangle$ where C and D include subsets of clauses in the original file such that C does not cause a failure, D does cause a failure, and D includes all the clauses in C and one more clause. We executed Ganak using C and D. We then observed which lines were executed differently between the two files. This helped us to pinpoint the exact location of the *Crash* bug in Ganak. This shows that test input minimization helps the developers to localize the bugs in model counters and makes it easy to fix them. *We concluded that input minimization with delta debugging is a very helpful technique in debugging model counters.*

5.3 Discussion

Fault localization and bug fixes for model counters. As explained earlier, the source code of projMC was not available and developers of dSharp considered bugs as lack of functionality. Therefore, we could not perform fault localization on these two model counters. However, we were successfully able to locate and fix bugs in Ganak. Figure 3a and 3b show location and fix for *Crash* bug in Ganak respectively. Ganak checks for UNSAT formulas by asserting that two unit clauses should not contain negation of the same literal. However, the assertion terminates Ganak before returning the expected 0 value. Figure 3c and 3d show location and fix for one of the reasons behind *WCnt* bug in Ganak. Ganak checks if a user specified *-p* in command for projected model counting. It uses *strcmp* function (returns 0 for matched strings and non-zero

otherwise) to compare the command with a predefined set of options. Ganak mistakenly turns projected model counting off by setting it to *false* instead of *true* due to which Ganak had problems in projected model counting.

Number of unique bugs found in model counters. Table 2 and Table 3 showed the number of formulas for which the model counters produced incorrect results. It is also important to know the number of unique bugs found in the model counters. For Ganak, TestMC found *WCnt* bug and *Crash* bug. All instances of *Crash* bug in Ganak were assertion errors (at the same location of the code) and all instances of *WCnt* bug were related to incorrect parsing of projected model counting formulas. In sum, we can say that TestMC found 2 unique bugs in Ganak. For dSharp, TestMC found two types of bugs i.e., *Crash* and *WSat* bug. For all 18 formulas for which TestMC reported *Crash* bug in dSharp, we experienced fatal OS crashes because dSharp consumed too much memory. The developers of dSharp recommended to use the tool inside containers with limited maximal memory usage. For the case of *WSat* bug, the authors confirmed that they expect the model counter to be deployed for only satisfiable formulas; and a SAT solver is responsible for checking unsatisfiable formulas. Since, the developers considered *WSat* and *Crash* bugs as lack of functionalities, we count them as two unique bugs. TestMC reported 2 instances of *SegFault* bug in projMC. Unfortunately, we were unable to get the access to source code which failed us to study the number of unique bugs in projMC. For ApproxMC, TestMC did not report any types of bugs.

Bug reports. Since Ganak is publicly available at GitHub, we used GitHub to submit bug report (two types of bugs i.e, *Crash* and *WCnt*) to Ganak developers. The bug report is available at https://github.com/meelgroup/ganak/issues/1. The developers have accepted the presence of bugs and made fixes to their tool. The fixed version of their tool is now available at Ganak's GitHub repository. For dSharp, we reported two bugs (*WSat* and *Crash*) to the authors via email. They considered the bugs as lack of functionalities and no fixes have been made so far. Since there is no public repository of projMC for us to report bugs (and the source-code is not available), we emailed the authors of projMC to report *SegFault* bug. We submitted the bug report together with two CNF formulas on which projMC crashed. The authors of projMC accepted the presence of a bug and promptly provided us with the fixed binary executable version of their tool.

Threats to Validity. Our focus in this paper was on testing the model counters in their standard configuration where they are most commonly deployed. More comprehensive testing can consider adjusting their configurations, e.g., initialization seed and confidence level, as well as the probabilistic nature of Ganak and ApproxMC, which may reveal more bugs. Our test generator controlled the number of formulas generated by breaking symmetries, e.g., by ignoring differences in the order clauses appear in a formula. While such symmetry breaking is necessary for making bounded exhaustive testing feasible, they may prevent the generation of an input formula that would have exposed a bug. We set a timeout of 5000 seconds which is common in the field of model counting. For the automatically generated small CNF formulas, the timeout did not matter since those inputs represented problems with low complexity, which were easily solved by all four model counters. For the larger CNF formulas that were derived from software designs, there were only 15 cases in which one or more model counters timed out. It is possible that more bugs are found if the model counters are kept running for a longer time, or if more diverse formulas are used.

6 LESSONS LEARNED

We learned several valuable lessons including the types of bugs found in model counters, reasons causing these bugs, what bugs are considered worth fixing by the tool authors, and factors that users of model counters may want to consider.

Lesson 1: The definition of a fault can be surprisingly ambiguous. We learned to our surprise that for some model counters, such as dSharp, it is acceptable for the tool developers if the tool incorrectly returns a positive model count, e.g., 1, when the input formula is *unsatisfiable*, i.e., has 0 solutions. The reasoning behind this contradictory situation is that the developers (in this case) expect the model counter to be deployed for only satisfiable formulas, and a propositional satisfiability (SAT) solver to be deployed for unsatisfiable formulas. While this expectation can be upheld in tool competitions where two different categories of formulas (sat and unsat) can be defined a prioiri, unfortunately, this expectation can make the deployment of such model counters costly in software analysis because to use such a model counter one must also run a SAT solver to ensure that the model counter's precondition of satisfiability of the input formula is met, thereby paying the cost of SAT solving *and* model counting.

Lesson 2: Small input formulas are extremely useful in testing and fault localization. While the benefits of bounded-exhaustive testing are well-documented, unfortunately its application remains quite limited. Our study shows how automatically generated small inputs are effective at revealing faults in state-of-the-art model counters. Despite the simplicity of generating such inputs, tool developers overlook their usefulness. Moreover, small inputs are extremely valuable in fault localization. For example, we were able to locate the *WCnt* bug in Ganak using small formulas generated using our TestMC bounded exhaustive generator, and we located the *Crash* bug in Ganak using test input minimization with delta debugging.

Lesson 3: Developers can overlook simple bugs in their code. Even though modern model counters employ sophisticated algorithms that have been rigorously validated on paper, their tool embodiments can fail for fairly simple reasons. Perhaps it is natural for developers to focus on the more complex parts of the system and meticulously engineer those parts while putting less focus on the other parts, which then become a source of the system failure. For example, for Ganak, we found that the model counting algorithm was not broken but in fact the code to parse command line arguments was buggy. Moreover, the tool did not handle assertion errors properly.

Lesson 4: It is difficult to help debug proprietary (closed - source) software. While our reported bug reports were promptly addressed by the authors of projMC, unfortunately we were not

able to get access to the source code to directly help with debugging the faults. This also prevented us to study the unique bugs in projMC. In such a situation we believe developments have a particularly valuable role to play since they exhibit minimal execution paths that show passing and failing executions.

Lesson 5: Model counters can exhibit silent failures. Like other systems, there can be multiple types of bugs in model counters, including silent failures where the tool reports a result that is invalid. We found two bug types in Ganak, namely the *WCnt* bug and the *Crash* bug, and two bug types in dSharp, namely the *WSat* bug and the *Crash* bug.

Lesson 6: Some model counters can consume up all system memory. On some executions of dSharp, we experienced fatal OS crashes because it consumed all system memory. For users of model counters, it is best to run them inside containers with limited maximal memory usage. This is particularly important when the model counter is one of the backend tools employed in a software analysis.

Lesson 7: Public version of the tool may remain faulty well after the bugs have been fixed internally. While the authors of projMC promptly provided us fixed versions of the tool based on the bug reports we submitted, the public version of projMC remains faulty (at the time of submission of this paper, which is over 6 months after we first reported the faults). There can be various reasons for the delay in pushing the updates to the public version. It is therefore important for the tool users to explicitly check with the tool authors if they have an internal version that has improvements that have not yet been made public.

7 RELATED WORK

This paper reports, to the best of our knowledge, the first work on applying automated testing techniques to find bugs in propositional model counters. The related work spawns many software testing areas, including differential testing, metamorphic testing, and test input generation for different kinds of systems [25, 28, 29, 33, 35, 46, 53, 60]. This section focuses on the most closely related work on bug finding for constraint solvers.

In the context of SAT solvers, the most closely related previous work is by Brummayer et al. [17] who introduced novel fuzzing techniques for SAT and quantified boolean formula (QBF) solvers. They implemented three fuzzers namely CNFFuzz for CNF formula generation, FuzzSAT for 3-SAT formula generation, and QBFuzz for quantified boolean formula generation, and used delta debugging for minimizing failure-inducing CNF inputs. The key differences between their work and this paper are: TestMC automates the testing of model counters, which are a generalization of SAT solvers, and require different test oracles, which TestMC's differential and metamorphic testing modules introduce; and moreover, TestMC introduces a bounded-exhaustive generator for CNF formulas with primary variables, which to our knowledge, is the first such tool.

Several projects have focused on fuzzing and metamorphic testing of SMT solvers [13] and constraint propagation solvers. Brummayer et al. [15] proposed grammar-based black-box fuzz testing for randomly generating bit-vector SMT formulas, combined with hierarchical delta-debugging using the knowledge of formula structures and types. Similar approaches are available for answer set solver
tool which performs SMT problem instance transformation and generation for string constraint solvers. Akgun et. al [3] proposed metamorphic testing for a constraint propagation solver called Minion [27] by checking the correctness and propagation level of a new propagation algorithm for a constraint by comparing it with a previously existing algorithm. Testing of the Gecode solver [51] has evolved similarly to Minion's. Several other works deal with the generation of random formulas [21, 38, 42], but focusing on theoretical properties of formulas and not on their suitability for supporting the solver debugging or testing.

Model-based testing [54], a common method in many domains [19] was also used for testing solvers by Artho et. al [4]. They used it to test sequences of application programming interface (API) calls and different system configurations for the SAT solver Lingeling [10]. Subsequently, a model-based API testing framework for the SMT solver Boolector [40] was proposed [41]. Furthermore, Modbat [5, 6], a model-based API testing tool that provides an embedded domain-specific language (DSL) for specifying the model, was used for testing the SAT solver PicoSAT[11].

We believe our bounded-exhaustive CNF generator and the corpus of CNF test inputs provide a useful resource for testing SAT solvers and other model counters. Moreover, our metamorphic relations admit a straightforward specialization for SAT solvers. We plan to leverage TestMC to test a broader class of CNF-based solvers and analysis tools.

8 CONCLUSION

This experience paper presented an empirical study on testing industrial strength model counters by applying the principles of differential and metamorphic testing together with bounded exhaustive input generation and input minimization. These principles were embodied in the TestMC framework, and applied to test four model counters, including three state-of-the-art model counters from three different categories: exact model counting, probabilistic exact model counting, and probabilistic approximate model counting. As test inputs, three complementary suites of CNF formulas were used. One suite consisted of significantly larger formulas that are derived from a wide range of real-world software design problems. The second suite consisted of a bounded exhaustive set of small formulas that TestMC generated. The third suite consisted of CNF formulas generated using an off-the-shelf CNF fuzzer. TestMC found bugs in three of the four subject model counters. The bugs led to crashes, segmentation faults, incorrect model counts, and resource exhaustion by the solvers. Faults in two of the three model counters were fixed by their authors based on the bugs found by TestMC.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for very helpful comments and feedback. This work was partially supported by the National Science Foundation grant CCF-1718903.

REFERENCES

[1] 2019. Alloy 4 Download Webpage. http://alloy.lcs.mit.edu/alloy/download.html.

[2] Bestoun Ahmed. 2016. Test Case Minimization Approach Using Fault Detection and Combinatorial Optimization Techniques for Configuration-Aware Structural Testing. *Journal of Engineering Science and Technology* 12 (05 2016), 737–753.

[3] Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. 2018. Metamorphic Testing of Constraint Solvers. In *Principles and Practice of Constraint Programming*. 727–736.

[4] Cyrille Artho, Armin Biere, and Martina Seidl. 2013. Model-Based Testing for Verification Back-Ends. In *Tests and Proofs*. 39–55.

[5] Cyrille Artho, Martina Seidl, Quentin Gros, Eun-Hye Choi, Takashi Kitamura, Akira Mori, Rudolf Ramler, and Yoriyuki Yamagata. 2015. Model-Based Testing of Stateful APIs with Modbat. *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2015), 858–863.

[6] Cyrille Valentin Artho, Armin Biere, Masami Hagiya, Eric Platon, Martina Seidl, Yoshinori Tanabe, and Mitsuharu Yamamoto. 2013. ModBat: A Model-Based API Tester for Event-Driven Systems. In *Hardware and Software: Verification and Testing*, Valeria Bertacco and Axel Legay (Eds.). 112–128.

[7] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. 2015. Automata-Based Model Counting for String Constraints. In *Computer Aided Verification*. 255–272.

[8] Rehan Abdul Aziz, Geoffrey Chu, Christian J. Muise, and Peter J. Stuckey. 2015. #∃SAT: Projected Model Counting. In *SAT*.

[9] Fahiem Bacchus and Toby Walsh. 2005. A non-CNF DIMACS style.

[10] Armin Biere. [n.d.]. *Lingeling and Friends at the SAT Competition 2011*. Technical Report.

[11] Armin Biere. [n.d.]. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT* ([n. d.]), 2008.

[12] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. 2009. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*.

[13] Nikolaj Bjørner. 2016. SMT Solvers: Foundations and Applications. *Dependable Software Systems Engineering* 45 (2016), 24.

[14] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. 2018. StringFuzz: A Fuzzer for String Solvers. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 45–51.

[15] Robert Brummayer and Armin Biere. 2009. Fuzzing and Delta-debugging SMT Solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories* (Montreal, Canada) *(SMT '09)*. ACM, New York, NY, USA, 1–5. https://doi.org/10.1145/1670412.1670413

[16] Robert Brummayer and Matti Järvisalo. 2010. Testing and Debugging Techniques for Answer Set Solver Development. *CoRR* abs/1007.3223 (2010). arXiv:1007.3223 http://arxiv.org/abs/1007.3223

[17] Robert Brummayer, Florian Lonsing, and Armin Biere. 2010. Automated Testing and Debugging of SAT and QBF Solvers. In *Theory and Applications of Satisfiability Testing – SAT 2010*, Ofer Strichman and Stefan Szeider (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 44–57.

[18] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. 2016. Algorithmic Improvements in Approximate Counting for Probabilistic Inference: From Linear to Logarithmic SAT Calls. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*.

[19] Harald Cichos, Sebastian Oster, Malte Lochau, and Andy Schürr. 2011. Model-Based Coverage-Driven Test Suite Generation for Software Product Lines. In *Model Driven Engineering Languages and Systems*, Jon Whittle, Tony Clark, and Thomas Kühne (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 425–439.

[20] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. 1999. *Model Checking*. MIT Press.

[21] Nadia Creignou, Uwe Egly, and Martina Seidl. 2012. A Framework for the Specification of Random SAT and QSAT Formulas. In *Tests and Proofs*, Achim D. Brucker and Jacques Julliand (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 163–168.

[22] Adnan Darwiche. 2000. On the tractable counting of theory models and its application to belief revision and truth maintenance. *CoRR* cs.AI/0003044 (2000). http://arxiv.org/abs/cs.AI/0003044

[23] Martin Davis, George Logemann, and Donald Loveland. 1962. A Machine Program for Theorem-proving. *Commun. ACM* 5, 7 (July 1962), 394–397. https://doi.org/10.1145/368273.368557

[24] Carmel Domshlak and Jörg Hoffmann. 2011. Probabilistic Planning via Heuristic Forward Search and Weighted Model Counting. *CoRR* abs/1111.0044 (2011). arXiv:1111.0044 http://arxiv.org/abs/1111.0044

[25] Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. 2018. Testing Probabilistic Programming Systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 574–586. https://doi.org/10.1145/3236024.3236057

[26] Antonio Filieri, Corina S. Păsăreanu, and Willem Visser. 2013. Reliability Analysis in Symbolic Pathfinder. In *ICSE*. 622–631.

[27] Ian P. Gent, Chris Jefferson, and Ian Miguel. 2006. MINION: A Fast, Scalable, Constraint Solver. In *Proceedings of the 2006 Conference on ECAI 2006: 17th European Conference on Artificial Intelligence August 29 – September 1, 2006, Riva Del Garda, Italy*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 98–102. http://dl.acm.org/citation.cfm?id=1567016.1567043

[28] Muhammad Ali Gulzar, Yongkang Zhu, and Xiaofeng Han. 2019. Perception and Practices of Differential Testing. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice* (Montreal, Quebec, Canada) *(ICSE-SEIP '19)*. IEEE Press, Piscataway, NJ, USA, 71–80. https://doi.org/10.1109/ICSE-SEIP.2019.00016

[29] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. 2018. DLFuzz: Differential Fuzzing Testing of Deep Learning Systems. *CoRR* abs/1808.09413 (2018). arXiv:1808.09413 http://arxiv.org/abs/1808.09413

[30] Klaus Havelund and Thomas Pressburger. 2000. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer* 2, 4 (2000), 366–381.

[31] Daniel Jackson. 2002. Alloy: A Lightweight Object Modeling Notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11, 2 (April 2002).

[32] Daniel Jackson. 2012. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.

[33] Tomasz Kuchta, Thibaud Lutellier, Edmund Wong, Lin Tan, and Cristian Cadar. 2018. On the correctness of electronic documents: studying, finding, and localizing inconsistency bugs in PDF readers and files. *Empirical Software Engineering* 23, 6 (01 Dec 2018), 3187–3220. https://doi.org/10.1007/s10664-018-9600-2

[34] Jean-Marie Lagniez and Pierre Marquis. 2019. A Recursive Algorithm for Projected Model Counting. *Proceedings of the AAAI Conference on Artificial Intelligence* 33 (07 2019), 1536–1543. https://doi.org/10.1609/aaai.v33i01.33011536

[35] Daniel Lehmann and Michael Pradel. 2018. Feedback-directed Differential Testing of Interactive Debuggers. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 610–620. https://doi.org/10.1145/3236024.3236037

[36] William M. McKeeman. 1998. Differential Testing for Software. *DIGITAL TECHNICAL JOURNAL* 10, 1 (1998), 100–107.

[37] Christian Muise, Sheila Mcilraith, J. Beck, and Eric Hsu. 2012. Dsharp: Fast d-DNNF Compilation with sharpSAT. https://doi.org/10.1007/978-3-642-30353-1_36

[38] Juan Antonio Navarro Pérez and Andrei Voronkov. 2005. Generation of Hard Non-Clausal Random Satisfiability Problems. *Proceedings of the National Conference on Artificial Intelligence* 1, 436–442.

[39] Joseph P. Near and Daniel Jackson. 2016. Finding Security Bugs in Web Applications Using a Catalog of Access Control Patterns. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 947–958. https://doi.org/10.1145/2884781.2884836

[40] Aina Niemetz, Mathias Preiner, and Armin Biere. 2014. Boolector 2.0. *JSAT* 9 (2014), 53–58.

[41] Aina Niemetz, Mathias Preiner, and Armin Biere. 2017. MODEL-BASED API TESTING FOR SMT SOLVERS.

[42] Eugene Nudelman, Kevin Leyton-Brown, Holger H. Hoos, Alex Devkar, and Yoav Shoham. 2004. Understanding Random SAT: Beyond the Clauses-to-Variables Ratio. In *Principles and Practice of Constraint Programming – CP 2004*, Mark Wallace (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 438–452.

[43] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 609–620. https://doi.org/10.1109/ICSE.2017.62

[44] Quoc-Sang Phan and Pasquale Malacaria. 2014. Abstract Model Counting: A Novel Approach for Quantification of Information Leaks. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security* (Kyoto, Japan) *(ASIA CCS '14)*. ACM, New York, NY, USA, 283–292. https://doi.org/10.1145/2590296.2590328

[45] Dan Roth. 1996. On the Hardness of Approximate Reasoning. *Artif. Intell.* 82, 1-2 (April 1996), 273–302. https://doi.org/10.1016/0004-3702(94)00092-1

[46] Sergio Segura, Amador Durán, Ana B. Sánchez, Daniel Le Berre, Emmanuel Lonca, and Antonio Ruiz-Cortés. 2015. Automated Metamorphic Testing of Variability Analysis Tools. *Softw. Test. Verif. Reliab.* 25, 2 (March 2015), 138–163. https://doi.org/10.1002/stvr.1566

[47] Sergio Segura, Gordon Fraser, Ana B. Sánchez, and Antonio Ruiz Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Trans. Software Eng.* 42, 9 (2016), 805–824.

[48] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. 2019. GANAK: A Scalable Probabilistic Exact Model Counter. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, 1169–1176. https://doi.org/10.24963/ijcai.2019/163

[49] Mate Soos. 2014. CryptoMiniSat v4. *SAT Competition* (2014), 23.

[50] Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson. 2004. Software Assurance by Bounded Exhaustive Testing. *SIGSOFT Softw. Eng.*

Notes 29, 4 (July 2004), 133–142. https://doi.org/10.1145/1013886.1007531

[51] Gecode Team. 2006. *Gecode: Generic constraint development environment.* http://www.gecode.org

[52] Marc Thurley. 2006. sharpSAT – Counting Models with Advanced Component Caching and Implicit BCP. In *Theory and Applications of Satisfiability Testing - SAT 2006*, Armin Biere and Carla P. Gomes (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 424–429.

[53] Muhammad Usman, Wenxi Wang, Marko Vasic, Kaiyuan Wang, Haris Vikalo, and Sarfraz Khurshid. 2020. A Study of the Learnability of Relational Properties: Model Counting Meets Machine Learning (MCML). In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1098–1111. https://doi.org/10.1145/3385412.3386015

[54] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A Taxonomy of Model-based Testing Approaches. *Softw. Test. Verif. Reliab.* 22, 5 (Aug. 2012), 297–312. https://doi.org/10.1002/stvr.456

[55] Wenxi Wang, Muhammad Usman, Alyas Almaawi, Kaiyuan Wang, Kuldeep S. Meel, and Sarfraz Khurshid. 2020. A Study of Symmetry Breaking Predicates and Model Counting. In *TACAS*.

[56] Pamela Zave. 2017. Reasoning About Identifier Spaces: How to Make Chord Correct. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1144–1156.

[57] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Toulouse, France) *(ESEC/FSE-7)*. Springer-Verlag, Berlin, Heidelberg, 253–267. http://dl.acm.org/citation.cfm?id=318773.318946

[58] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.* 28, 2 (Feb. 2002), 183–200. https://doi.org/10.1109/32.988498

[59] A. Zeller and R. Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (Feb 2002), 183–200. https://doi.org/10.1109/32.988498

[60] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. 2019. Finding and Understanding Bugs in Software Model Checkers. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, New York, NY, USA, 763–773. https://doi.org/10.1145/3338906.3338932