Delta debugging fault-triggering propositional model counting instances

To facilitate debugging of unweighted model counters using SharpVelvet

David N. Coroian[1]

Supervisor: Dr. Anna L.D. Latour[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
January 26, 2025

Name of the student: David N. Coroian
Final project course: CSE3000 Research Project
Thesis committee: Dr. Anna L.D. Latour, Dr. Martin Skrodzki

Abstract

Propositional model counting (#SAT) is the counting variant of the Boolean Satisfiability (SAT) problem. Development of #SAT solvers has seen a boom in recent years. These tools are complex and hard to debug. To address this, we propose a delta debugger that reduces fault-triggering unweighted model counting instances. Our delta debugger shows an improvement compared to state of the art in the related field of SAT solvers.

1 Introduction

The first three paragraphs of this section are based on Latour's project description [2024].

The aim of *Propositional Model Counting* [Gomes *et al.*, 2021] is to find the *number* of unique solutions ("*models*") that satisfy a certain Boolean formula. This is the counting version of the Boolean Satisfiability (SAT) [Biere *et al.*, 2021] NP-complete problem, which aims to answer whether there is *at least one solution*. Therefore, the problem of model counting (#SAT) is conjectured to be computationally harder than NP.

There are numerous applications of model counting and its variants, three notable examples include: reliability estimation of power grid networks [Duenas-Osorio *et al.*, 2017], optimisation problems in bioinformatics and social sciences [Latour *et al.*, 2022] and verification of neural networks [Baluta *et al.*, 2019].

SharpVelvet [Latour and Soos, 2024] is an ongoing project that aims to provide model counter developers with a *fuzzer* and a *delta debugger*. The purpose of the fuzzer is to generate and run problem instances, in order to *trigger bugs* in a model counter. Once a bug is found, the delta debugger minimises the fault-triggering instance as much as possible, whilst still triggering the bug. As such, the model counter developers are more likely to locate the incorrect piece of code.

Since model counters have seen a boom only in the last few years, there are not many, if any, other fuzzing tools freely available to their developers. Modern model counters are complex and hard to debug, also witnessed by a recently proposed delta debugger for unweighted model counters, TestMC [Usman *et al.*, 2020]. We believe we can improve upon Usman et al. by implementing state of the art in general delta debugging [Wang *et al.*, 2021].

This leads to the research question we answer in this paper: **"Given an unweighted model counting instance that triggers a bug in a solver, and means of interacting with that solver, how much can we minimise the instance in such a way that it still triggers a bug, using heuristics based on the CNF structure and properties of individual clauses?"**.

In Section 2 we provide relevant definitions. Section 3 presents state of the art in delta debugging and related work. Then, in section 4 we introduce the DeltaMC framework. We describe our experimental setup in section 5 and analyse the results in section 6. We describe how we conducted responsible research in section 7 and conclude in section 8.

2 Preliminaries

This section introduces relevant concepts.

Input format

In boolean logic, a proposition in *Conjunctive Normal Form* (CNF) is defined as a conjunction ($\wedge$) of one or more clauses. A clause is a disjunction ($\vee$) of literals. Lastly, a literal consists of a variable or its negation. Two examples of propositions in CNF are $(p) \wedge (q)$ and $(p \vee q \vee r) \wedge (o \vee (\neg z))$.

Within general delta debugging, the logical proposition in CNF corresponds to the input ("instance") and a clause corresponds to an input element. For the scope of this work the literals within a clause are kept unchanged by our delta debugger.

Delta debugging

Delta debugging is defined as minimising a fault-triggering input, such that the resulting reduced instance still triggers a fault in the software that is being debugged. We measure the *performance* of a delta debugger in terms of achieved reduction (in $\%$) of the input size (n) and the ratio between the number of *delta-debugging tests* performed and n. A delta-debugging test consists of running the software with an intermediate reduced instance to check whether the bug is reproduced, and a pass is achieved if so.

Model counting flavours

There exist four variations of the model counting problem, *i.e.* unweighted, weighted, projected and projected weighted model counting [Gomes *et al.*, 2021]. In the scope of this paper we focus solely on unweighted model counting. These type of counters accept a proposition in CNF as input, similar to SAT solvers.

3 Background and Related Work

This section presents general delta-debugging techniques, related work in the field of SAT solvers and one proposed delta debugger for model counters.

3.1 General delta-debugging techniques

This section presents the four delta-debugging algorithms we found in the literature.

Leave-one-out

The *naïve* delta-debugging algorithm [Vu *et al.*, 2023] iteratively tries to remove an element from the instance. The average-case asymptotic number of tests performed is bound by $O(n^2)$.

Delta Debugging

Zeller and Hildebrandt introduced the concept of delta debugging and proposed dd-min [2002], based on the classic algorithm of *binary search*. Initially, tests are performed for instances containing half of the input. If successful, the algorithm discards the other half. Otherwise, it continues with the original instance. Afterwards, quarters of the instance are removed and the resulting subsequences are tested. In the same way, the reduction continues until individual elements are removed, similar to the leave-one-out strategy. The authors bound the worst-case asymptotic number of tests performed by dd-min by $O(n^2)$.

Algorithm 1: The prob-dd algorithm.

Input: Set S containing the elements of the initial instance.
Output: Set Q containing the elements of the reduced instance.

$n \leftarrow |S|$;
$p[1 \ldots n] \leftarrow \sigma$; `// Initial probabilities.`
repeat
  `// Next delta-debugging test.`
  $D \leftarrow$ `SelectTest` (p);
  `// True if fault still present.`
  $R \leftarrow$ `RunTest` (D);
  `// Update probabilistic model.`
  $p \leftarrow$ `UpdateModel` (R, p);
until `ReductionComplete` (p);
$Q \leftarrow$ `ReducedInput` (p);
return Q

Probabilistic Delta Debugging

Wang et al. recently introduced the prob-dd algorithm that employs a *probabilistic model* to guide the reduction of the input. The description of the algorithm is based on the original paper [2021].

A minimal version of the original algorithm can be seen in Algorithm 1. The input S is assumed to be pre-processed, possibly using domain-specific knowledge, such that all *statistical dependencies* between elements are eliminated. Subsequently, the model assigns each input element a Bernoulli random variable, representing whether the element is included in the final, reduced instance Q. Together with the previous assumption, it immediately follows that these variables are mutually independent. All the variables are assigned an initially identical value of σ.

Moving on to the delta-debugging part, prob-dd forms a subset D of the last fault-triggering instance, initially S, according to the probabilities p. As explained earlier, a higher probability implies a greater chance of an element belonging to the set Q. Afterwards, the delta-debugging test is performed and the model is updated to reflect whether D produced a bug. Finally, the reduction concludes when all probabilities are either 0 *or* 1. Wang et al. bound the worst-case asymptotic number of tests conducted by prob-dd by $O(n)$. For an in-depth explanation of the prob-dd algorithm, we point the reader to the original article.

Similarity-Based Isolation

Vu et al. introduced similarity-iso [2023], improving dd-min and dd [Zeller and Hildebrandt, 2002] on localizing the fault-triggering elements by using a domain-specific distance metric to group such elements. The algorithm is bound by the underlying performance of dd-min. We refer the reader to the full article for more details.

3.2 Delta Debuggers for SAT solvers

Based on the literature study we conducted, state of the art in SAT delta debuggers is considered to be the adaptation of dd-min [Zeller and Hildebrandt, 2002] introduced by Brummayer et al. [2010]. The authors improve the performance of the original dd-min algorithm by incorporating domain-specific knowledge. We refer the reader to the original paper for technical details.

3.3 Delta Debuggers for Model Counters

The TestMC framework [Usman *et al.*, 2020] is the only implementation of a delta debugger, developed specifically for model counters, that we were able to find during our literature study. The proposed delta debugger is based on the dd-min algorithm and achieved a 30% reduction of the input. At the time of submission of this paper the TestMC source code is not publicly available.

4 Methodology

This section describes the scope of this research, our proposed delta-debugging framework, and the two delta debuggers we applied to unweighted model counting.

4.1 Scope

Given the duration of this project, we implemented a delta debugger solely for unweighted model counters. The projected, weighted and projected weighted model counting problems are more complex and therefore a non-trivial delta debugger implementation for these types of model counters would require considerably more effort. Out of the four delta-debugging techniques, we chose to implement prob-dd [Wang *et al.*, 2021]. In addition, we apply cnfdd [Brummayer *et al.*, 2010] to unweighted model counting. The naïve method of leave-one-out [Vu *et al.*, 2023] cannot achieve better performance than dd-min [Zeller and Hildebrandt, 2002], which is implemented by cnfdd. In addition, considering that similarity-iso [Vu *et al.*, 2023] is built on top of dd-min, we decided to apply prob-dd in order to experiment with a different delta-debugging approach.

4.2 DeltaMC framework

We present DeltaMC, an optional extension of SharpVelvet [Latour and Soos, 2024] that enables support for delta debugging model counters.

DeltaMC can be used to debug any type of model counter, *i.e.* unweighted, weighted, projected or projected weighted, if provided with a corresponding delta debugger implementation.

The high-level design can be seen in Algorithm 2. Initially, the delta debugger is instantiated for an instance that produced a bug in the debugged model counter. Afterwards, preprocessing of the CNF formula takes place, during which free and fixed variables are removed and equivalent adjacent clauses are merged. Subsequently, the main loop will run until the delta debugger is unable to minimise the instance further. Within one iteration, a delta-debugging test is selected, then SharpVelvet is employed to fuzz the model counter and lastly the result of the test, *i.e.* pass if fault produced, is passed onto the delta debugger. Finally, the resulting reduced instance is printed, and as such it can be further manually reviewed and reported to the model counter developers.

Algorithm 2: The DeltaMC framework.

Input: Delta debugger DD, fault-triggering input $instance$, debugged solver $buggy\_mc$.
Output: Reduced instance in CNF.

```
Function DeltaDebug (instance, buggy_mc)
  begin
    /* Initialise delta debugger with
       the preprocessed instance.   */
    preprocesssed_ins ← Preprocess
      (instance);
    DD.Initialise (preprocessed_ins);
    while !DD.Finished () do
      /* Select the next
         delta-debugging test.     */
      test_ins ← DD.SelectTest ();
      /* Check whether fault is
         triggered.                */
      bug_present ← Fuzz (buggy_mc, test_ins);
      // Process result of test.
      DD.Update (bug_present);
    end
    /* Print the reduced CNF
       instance.                   */
    DD.PrintReducedInstance ();
end
```

4.3 Prob-dd application

Wang et al. assume statistical independence between the input elements when defining the probabilistic model [2021]. In the context of model counting, some degree of dependence is still present in the preprocessed input. This is caused by propositional variables that occur in more than one clause. Therefore this assumption does not hold, since an instance where each variable belongs to at most one clause can easily be solved by modern model counters. In spite of that, we believe that in practice prob-dd will have a good performance, since clauses in the preprocessed input are fairly distinct, resulting in a low degree of statistical dependence.

The authors assume no prior knowledge of the input elements. In the original paper, Wang et al. state that the choice of σ does not considerably influence the number of tests performed, neither the input size reduction. In the scope of this work, we integrate prior knowledge of the input elements in order to improve the performance of prob-dd. Our hypothesis is that setting higher initial probabilities for the more complex clauses and lower values for the simpler clauses, could reduce the amount of delta-debugging tests and possibly improve the input reduction.

4.4 Prob-dd Heuristics

We now present the three heuristics we used for setting the initial probabilities of the model:

Heuristic 1 (H1)
We assign each variable an initial probability of 0.1. Wang et al. set the initial probability based on the expected reduction ratio [2021].

Heuristic 2 (H2)
We use the ratio between the number of literals present in the clause and the total number of literals in the proposition as initial probability.

Heuristic 3 (H3)
We compute a score for each clause based on the rarity of its literals, *i.e.* how often they appear in the entire proposition. We believe that a literal that appears more frequently results in stronger clauses, *i.e.* harder to solve by the model counters and more likely to trigger bugs.

We define the frequency F of a literal p as the number of its occurrences in the proposition L. As can be seen in equation 1, the score S of a clause c is calculated as the sum of frequencies of its literals, divided by the total sum of frequencies.

$$S_c = \frac{\sum\limits_{p \in c} F_p}{\sum\limits_{p \in L} F_p} \quad (1)$$

Probabilities scaling
Running prob-dd with the raw scores of H2 and H3 resulted in unexpected behaviour, since the values are often below 0.01. Such values rendered the model unable to select the next delta-debugging test. To address this, we further scale the initial probabilities to $[0.1, 0.4]$.

4.5 Cnfdd application

We obtained the latest version [Artho *et al.*, 2013] of cnfdd[1] [Brummayer *et al.*, 2010]. Since SAT CNF instances are similar to unweighted model counting CNF instances, the only change needed in the source code was printing $c\ t\ mc$ at the top of the delta debugger output file. This comment line signalled to model counters to treat the instance as unweighted.

5 Experimental Setup

This section presents the experimental setup used for the DeltaMC framework, our implementation of prob-dd, the cnfdd setup, our assumptions about faults, the hardware setup, the solvers and generators used.

5.1 Research Questions

Through the experiments we aim to answer the following two research questions:
Q1: How do the prob-dd heuristics perform when applied to unweighted model counting?
Q2: How do prob-dd and cnfdd compare in terms of performance?

5.2 DeltaMC setup

The framework is implemented in Python 3.11, using Cython 3.0.11[2] to integrate C++ implementations of delta debuggers. The choice of programming languages followed naturally, given that SharpVelvet [Latour and Soos, 2024] is

[1]Source code available at https://fmv.jku.at/cnfuzzdd/cnfuzzdd2013.zip
[2]https://cython.org/

implemented in Python while our implementation of prob-dd is in C++. In addition, we built the framework decoupled from the actual delta debugger implementation to facilitate future research in this field.

5.3 Prob-dd implementation

Our prob-dd implementation is exclusively based on the algorithm described in the original paper [Wang *et al.*, 2021]. The implementation is in C++ 20, and we reused parts of the GPMC solver [Suzuki *et al.*, 2017] source code[3].

We chose to build our delta debugger on top of an existing model counter since it enabled us to implement solely the prob-dd logic. As such, our implementation makes use of the existing functionality for parsing, printing, preprocessing and storing in-memory CNF instances.

5.4 Cnfdd setup

Since adapting the implementation to follow the proposed API was outside the scope of this work, we wrapped cnfdd with a short Python script containing the DeltaMC fuzzing logic.

5.5 Solvers

We used the binaries [Fichte *et al.*, 2024] submitted to the 2024 edition of the Model Counting Competition [Fichte *et al.*, 2021]. The unweighted track, with exact precision, had ten participants, out of which we selected Ganak [Sharma *et al.*, 2019], D4 [Lagniez and Marquis, 2017] and GPMC [Suzuki *et al.*, 2017], corresponding to the first, fourth and eighth positions of the leader-board respectively. By applying our delta debugger to state-of-the-art solvers with different underlying implementations, we demonstrate its relevance and future potential to assist development of model counters.

5.6 Fault definition

We consider three types of faults, namely wrong counts, crashes and timeouts.

Wrong count

The model count of a solver is considered wrong if the majority of the other surveyed solvers agree on a different result.

Crash

A crash is defined as an abnormal early exit of a solver, without providing any result.

Timeout

We expect solvers to finish counting within a limit of 10 seconds for an easy instance. An instance is considered easy if any of the three model counters is able successfully solve it within the time limit. This limit was chosen to facilitate experimenting with the delta debugger, considering the time frame of this project.

[3]Source code available at `https://git.trs.css.i.nagoya-u.ac.jp/k-hasimt/GPMC`

5.7 CNF generators

SharpVelvet provides adaptations of the CNFuzz and FuzzSAT generators [Brummayer *et al.*, 2010]. We generated 100.000 CNF instances, 70.000 using CNFuzz, and 30.000 using FuzzSAT. We used the default configuration of CNFuzz, since the resulting instances were of varying hardness and size. For FuzzSAT we used three configurations, in order to vary the size and structure of the generated instances.

5.8 Hardware

We acknowledge the use of computational resources of the DelftBlue supercomputer [Centre (DHPC), 2025]. We ran experiments on nodes equipped with Intel Xeon E5-6248R CPUs, running at 3.0 GHz. The operating system of the cluster is Red Hat Enterprise Linux 8.1. Each solver was allocated 64 GB of RAM and one core, since the underlying implementations do not make use of parallelisation.

6 Results

This section presents the fuzzing results and delta-debugging performance achieved.

6.1 Triggered faults

Model counting software has improved considerably in recent years. In addition, it is highly likely these tools were fuzzed and debugged using the CNFuzz and FuzzSAT generators [Brummayer *et al.*, 2010]. As a consequence, we were not able to trigger any wrong count or crash bugs. We fuzzed solvers with 100.000 CNF instances, which exhibited a considerable amount of variety in structure and hardness. While we consider our sample size relevant, finding bugs is an uncertain process.

In an ideal scenario, the performance of a delta debugger is measured by reducing fault-triggering instances. Considering we were not able to produce any bugs, we tested our delta debugger implementation by reducing inputs that triggered timeouts. This experiment showcases the ability of the delta debugger to narrow the *hardness* of the instance, *i.e.* the subset of clauses which take the longest to solve. We believe this performance would further translate well to reducing instances that trigger wrong count or crash bugs. Out of the 100.000 CNF instances we fuzzed solvers with, a small proportion triggered timeouts, from which we randomly selected 30 CNF instances and delta debugged.

6.2 Delta-debugging performance

Table 1 presents the average input reduction performance and Table 2 presents the average ratio between the number of delta-debugging tests performed and input size.

We now answer our research questions:

Q1: Heuristics performance. While all of the three heuristics achieve a high reduction of the input, on average H2 performs better by a small margin, followed by H3. Similarly, H2 conducts the least amount of delta-debugging tests, while H1 requires the most tests. Thus, we show that, in the context of unweighted model counting, prior knowledge of the input successfully improves the performance of the probabilistic model by a small yet non-insignificant margin.

Solver	H1	H2	H3	cnfdd
Ganak	84.69	91.46	90.13	89.43
D4	84.06	89.44	87.28	90.41
GPMC	82.89	86.10	84.18	86.52

Table 1: Average input reduction (%) achieved by the three prob-dd heuristics and cnfdd.

Solver	H1	H2	H3	cnfdd
Ganak	0.44	0.33	0.37	3.26
D4	0.49	0.36	0.42	3.59
GPMC	0.50	0.40	0.47	3.18

Table 2: Average ratio between number of tests and input size, *i.e.* efficiency of the three prob-dd heuristics and cnfdd (lower is better).

Q2: Comparison with cnfdd [Brummayer *et al.*, 2010]. H2 reduces the input approximatively as much as cnfdd, while H1 and H3 achieve a slightly weaker reduction. All the heuristics perform considerably less tests compared to cnfdd, with H2 performing $\sim 10\times$ less tests. In practice, this translates to a significantly shorter runtime of the delta debugger. This improvement is explained by the efficiency of prob-dd.

7 Responsible Research

This section presents ethical considerations of this work, how scientific integrity was observed while conducting research, and finally the transparency and reproducibility aspects.

7.1 Ethical considerations

The resulting tool, DeltaMC, could at most be used for debugging model counting software. This software is already freely and publicly available. To our knowledge there has been no malicious use of such solvers at the time of submitting this article. While we consider malignant use unlikely to happen in the future, model counting does have practical applications. Considering the example of verifying neural networks [Baluta *et al.*, 2019], one could ask about the ethical implications or intended uses of the verified neural network itself. Therefore, we consider this project does not introduce new ethical concerns.

7.2 Scientific integrity

All the source code and binaries were obtained and used according to their respective open-source licenses. No use of AI tools was made at any point of this research.

7.3 Transparency and Reproducibility

All our research artifacts are available open source and we provide instructions on how to use our delta debugger in the repository.

In order to abide by the terms of the various licenses governing the source code and binaries we made use of, our artifacts are published in two separate repositories.

The DeltaMC code, prob-dd implementation and the CNF instances used are available under MIT license at: `https://github.com/davidcoroian/cse3000-deltamc`, commit hash 8d9377f.

The adapted cnfdd source code and wrapper script are published under GPLv3 license at: `https://github.com/davidcoroian/cse3000-cnfdd`, commit hash 93dd8d7.

8 Conclusions and Future Work

This paper focused on delta debugging unweighted model counters. First, we proposed the DeltaMC framework, an optional extension of SharpVelvet [Latour and Soos, 2024] that can be coupled with a delta debugger implementation to debug a model counter. Then, we implemented a delta debugger based on the state-of-the-art technique of probabilistic delta debugging [Wang *et al.*, 2021]. Finally, our empirical evaluation of the performance of our delta debugger demonstrates a $\sim 10\times$ improvement in terms of number of delta-debugging tests performed compared to state of the art in the related field of SAT solvers.

While our findings are promising, during this research we were unable to produce wrong count or crash bugs and we benchmarked our delta debugger by reducing timeouts. We believe the delta debugger would achieve a good performance when minimising wrong count or crash bugs, but this remains to be tested in future experiments.

Future work in this field includes implementing delta debuggers for weighted, projected and projected weighted model counters. In addition, it could be worth experimenting with other recently proposed delta-debugging techniques such as similarity-iso [Vu *et al.*, 2023].

References

[Artho *et al.*, 2013] Cyrille Artho, Armin Biere, and Martina Seidl. Model-Based Testing for Verification Back-Ends. In Margus Veanes and Luca Viganò, editors, *Tests and Proofs*, pages 39–55, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Baluta *et al.*, 2019] Teodora Baluta, Shiqi Shen, Shweta Shinde, Kuldeep S. Meel, and Prateek Saxena. Quantitative Verification of Neural Networks and Its Security Applications. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 1249–1264, New York, NY, USA, November 2019. Association for Computing Machinery.

[Biere *et al.*, 2021] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, second edition edition, 2021.

[Brummayer *et al.*, 2010] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated Testing and Debugging of SAT and QBF Solvers. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing – SAT 2010*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, Berlin, Heidelberg, 2010.

[Centre (DHPC), 2025] Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 2). `https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2`, 2025.

[Duenas-Osorio *et al.*, 2017] Leonardo Duenas-Osorio, Kuldeep Meel, Roger Paredes, and Moshe Vardi. Counting-Based Reliability Estimation for Power-Transmission Grids. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31(1), February 2017. Art. no. 1.

[Fichte *et al.*, 2021] Johannes K. Fichte, Markus Hecher, and Florim Hamiti. The model counting competition 2020. *Association for Computing Machinery Journal of Experimental Algorithmics (JEA)*, 26:1–26, October 2021.

[Fichte *et al.*, 2024] Johannes Fichte, Markus Hecher, and Arijit Shaw. Model counting competition 2024: Submitted solvers. `https://doi.org/10.5281/zenodo.14249109`, November 2024.

[Gomes *et al.*, 2021] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model Counting. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 993–1014. IOS Press, second edition edition, 2021.

[Lagniez and Marquis, 2017] Jean-Marie Lagniez and Pierre Marquis. An Improved Decision-DNNF Compiler. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence, IJCAI-17*, volume 17, pages 667–673, 2017.

[Latour and Soos, 2024] Anna L.D. Latour and Mate Soos. SharpVelvet. `https://github.com/meelgroup/SharpVelvet`, August 2024.

[Latour *et al.*, 2022] Anna L.D. Latour, Behrouz Babaki, Daniël Fokkinga, Marie Anastacio, Holger H. Hoos, and Siegfried Nijssen. Exact stochastic constraint optimisation with applications in network analysis. *Artificial Intelligence*, 304:103650, March 2022.

[Latour, 2024] Anna L.D. Latour. Finding different ways to break a solver. `https://projectforum.tudelft.nl/course_editions/106/generic_projects/5772`, November 2024.

[Sharma *et al.*, 2019] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S Meel. GANAK: A Scalable Probabilistic Exact Model Counter. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI-19*, volume 19, pages 1169–1176, 2019.

[Suzuki *et al.*, 2017] Ryosuke Suzuki, Kenji Hashimoto, and Masahiko Sakai. Improvement of projected model-counting solver with component decomposition using SAT solving in components. Technical report, JSAI Technical Report, SIG-FPAI-506-07, 2017.

[Usman *et al.*, 2020] Muhammad Usman, Wenxi Wang, and Sarfraz Khurshid. TestMC: Testing Model Counters using Differential and Metamorphic Testing. In *Proceedings of the 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 709–721, Melbourne, VIC, Australia, 2020. ISSN: 2643-1572.

[Vu *et al.*, 2023] Anh D. Vu, Christos Tsigkanos, Jorge-Arnulfo Quiané-Ruiz, Volker Markl, and Timo Kehrer. On Irregularity Localization for Scientific Data Analysis Workflows. In *Computational Science – ICCS 2023*, pages 336–351. Springer, 2023.

[Wang *et al.*, 2021] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. Probabilistic Delta debugging. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, pages 881–892, New York, NY, USA, August 2021. Association for Computing Machinery.

[Zeller and Hildebrandt, 2002] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002. Conference Name: IEEE Transactions on Software Engineering.