AlloyMC: Alloy Meets Model Counting

Jiayi Yang
University of Texas at Austin, USA

Wenxi Wang
University of Texas at Austin, USA

Darko Marinov
University of Illinois at Urbana-Champaign, USA

Sarfraz Khurshid
University of Texas at Austin, USA

ABSTRACT

Specifying and analyzing desired properties of software systems can play an important role in the development of more dependable systems. Alloy is a mature tool-set that provides a first-order, relational logic with transitive closure for writing specifications, and a fully automatic backend based on propositional satisfiability (SAT) solvers for analyzing them. Alloy's intuitive notation and support for modern solvers make it an effective specification and analysis tool, which has led it to be applied in several domains, including verification, security, and synthesis.

This paper introduces a new backend for Alloy, which complements SAT solvers, and provides a new method to assist Alloy users to more effectively use the tool-set, specifically in scenarios where *multiple solutions* to the same formula are desired. We add to the Alloy backend support for model counting, i.e., computing the number of solutions to the given formula. We extend the Alloy grammar to add a new command for model counting, and extend the Alloy GUI to customize it. Our implementation, called AlloyMC, supports two state-of-the-art model counters: the approximate model counter ApproxMC and the exact model counter ProjMC. AlloyMC runs on Linux, Mac, and Windows. To use AlloyMC, users just download and run its integrated JAR file with no need to install dependencies (e.g., model counters and their dependent libraries). The AlloyMC source code, the JAR file, and the data set are available publicly.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**.

KEYWORDS

Alloy, model counting, solution enumeration

ACM Reference Format:
Jiayi Yang, Wenxi Wang, Darko Marinov, and Sarfraz Khurshid. 2020. AlloyMC: Alloy Meets Model Counting. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3368089.3417938

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3417938

1 INTRODUCTION

Propositional model counting is the problem of computing the number of solutions for a given propositional formula, which is the number of distinct truth assignments to variables for which the formula evaluates to true [10]. Effective model counting is a key that enables a variety of new application areas such as probabilistic inference, planning, combinatorial design [5, 8, 14, 22, 24, 26, 27], security [4, 20, 25], and system and software analysis [11, 15, 16]. A commonly used form of model counting is *projected model counting* where the goal is to compute the number of distinct solutions with respect to only a specific subset of variables; this subset is called *independent* or *primary* variables. Model counting techniques broadly fall into two categories: approximate counting and exact counting [17]. Two tools that represent the state-of-the-art in these categories are ApproxMC [13] for approximate model counting and ProjMC [21] for exact model counting. Both ApproxMC and ProjMC support projected model counting.

Alloy is a lightweight toolset for writing and analyzing specifications of software systems [18]. It is designed to naturally facilitate writing correct specifications where the user starts with a partial specification and builds it incrementally, aided at each step by rapid analysis performed by Alloy's backend. The Alloy analyzer employs the heavily optimized constraint solver Kodkod [31], which uses a bound on the universe of discourse, termed *scope*, to translate Alloy specifications into propositional satisfiability (SAT) formulas. The translated SAT formula is then solved by off-the-shelf SAT solvers, which have seen noteworthy advances over the last two decades [9]. Alloy's intuitive language design, its efficient translation, and powerful backend solving have enabled a wide range of Alloy applications, including design and modeling [3, 6, 7], hardware and software security [1, 32], system analysis and verification [12, 19, 23], and software testing [28].

Some of these applications require enumerating *multiple solutions* to the given specification, e.g., when it characterizes properties of desired object graphs for bounded exhaustive testing [34], or of desired hardware litmus tests for finding hardware vulnerabilities [33]. The effectiveness of Alloy in such applications depends critically on the quality and number of solutions created. A key issue with Alloy's original backend that is based solely on SAT solving is that users can't use it to estimate the number of solutions without actually enumerating them all using an enumerating SAT solver. Thus, if the user wants to modify their specification, say by adding a new constraint, to reduce the number of solutions [32], they have to wait for full enumeration of the initial specification and the modified specification to determine if a desired reduction is achieved. Furthermore, it is definitely not applicable for users to click the button for every solution enumeration to count even for a moderate-sized problem which usually has millions of solutions.

Our key insight is that we can enhance Alloy by adding direct support for model counting so the user does not have to repeatedly pay the cost of full enumeration using SAT solvers, and can instead benefit from the state-of-the-art model counters that use advanced methods to efficiently compute the approximate or exact number of solutions. We present AlloyMC, which adds model counting support to Alloy. The specific model counting problem for Alloy specifications directly maps to projected model counting where the goal is to count the number of solutions with respect to the CNF-level variables that directly correspond to the relations declared in the Alloy specification.

AlloyMC extends the Alloy grammar to support a new command for invoking backend model counters, and extends the Alloy GUI to support customizing the invocation. AlloyMC allows users to (optionally) specify an expected model count, and reports the model counting results with respect to the expectation, thereby providing another way to validate Alloy specifications in the spirit of unit testing. We extended Alloy parsing and translation accordingly to convert Alloy specifications with the count command into the corresponding projected model counting problems. At present, AlloyMC supports two state-of-the-art model counters – the approximate model counter ApproxMC and the exact model counter ProjMC. The design of AlloyMC allows other model counters to be added in future. AlloyMC runs on Linux, Mac, and Windows. To use AlloyMC, users just download and run its integrated JAR file with no need to install dependencies (e.g., model counters and their dependent libraries). The AlloyMC source code, the JAR file and the data set are available publicly (https://github.com/jiayiyang1997/org.alloytools.alloy/tree/model_count). A screencast of the tool demonstration is also available at https://youtu.be/iogIrJ8kHI4.

2 ALLOY BACKGROUND

This section gives a quick overview of Alloy using an illustrative example. An Alloy model declares sets and relations, defines constraints on them, and includes commands for invoking the backend engine. The keyword `sig` declares a set of atoms, and may optionally have fields that declare (binary, ternary, or higher arity) relations. The keyword `pred` introduces a *predicate* that defines constraints on the sets and relations, and can be invoked elsewhere. Constraints can also defined using *facts* that must always be satisfied. Alloy's `run` command allows users to invoke a predicate so the Alloy analyzer returns an instance if the predicate is satisfiable, or reports that the predicate is unsatisfiable. Users can also use Alloy's `check` command to search for a counterexample that refutes the given constraint. Each command has a user-defined *scope* that bounds the number of atoms in the signatures; the default scope is 3, i.e., at most 3 atoms in each set. The Alloy analyzer performs scope-bounded analysis where the analysis results are valid for the given scope.

Figure 1 illustrates a small Alloy specification of an acyclic linked list, and an invocation of the analyzer to search for a list with up to 5 nodes (Lines 1–9). The signature `List` (Lines 1–2) requires that there be exactly one list atom (restricted by keyword `one`). The signature `Node` (Lines 3-4) declares a set of node atoms. The `header` field (Line 2) is a partial function that maps the list atom to at most one node atom. The `link` field (Line 4) is similarly a partial

```
1. one sig List {
2.         header: lone Node }
3. sig Node {
4.         link: lone Node }
5. fact { List.header.*link = Node }
6. pred Acyclic(l: List) {
7.         all n: l.header.*link | n !in n.^link }
8. run Acyclic for 5
9. run Acyclic for 5 expect 1
10. cmd1: count Acyclic for 5
    -- simple count command
11. cmd2: count Acyclic for 5 expect 8
    -- expected count command
12. cmd3: count Acyclic for 5 expect >= 8
    -- operators can be used: <=, >=, =, >, <, !=
13. cmd4: count Acyclic for 5 expect 1*2^4
    -- expected count in x*y^z format
14. cmd5: count Acyclic for 5 expect -1
    -- ignore negative expected count
```

Figure 1: Acyclic linked list Alloy model

function from nodes to nodes. The `fact` ensures that each node is reachable from the list header. The predicate `Acyclic` (Lines 6–7) states that for every node `n` which is reachable from the parameter list `l`'s header following zero or more traversals (defined by `*`) along the `link`, `n` is not reachable from itself following one or more traversals (defined by `^`) along the `link`. The first `run` command (Line 8) searches for an `Acyclic` list `l` with up to 5 nodes. Alloy allows users to specify an expectation on whether *a* solution is expected or not using they `expect` clause, which helps in validating the specification (akin to unit testing). The second `run` command (Line 9) searches for an `Acyclic` list with the expectation that such a list is expected (indicated by "`expect 1`"); "`expect 0`" indicates that the predicate invoked is expected to be unsatisfiable; if the user provides a negative number for the `expect` clause, the clause is ignored by the analyzer.

3 ALLOYMC

3.1 Overview

Figure 2 illustrates the overall architecture of AlloyMC. AlloyMC adds support for model counting to Alloy by extending the original Alloy language with a new command named `count` (Section 3.2). We extended the Alloy lexer and parser accordingly to convert the Alloy specification with the count command into an intermediate AST. We also modified the Kodkod engine to convert the intermediate representation into an appropriate SAT formula (a CNF instance in the standard DIMACS format) that is augmented with the information on primary variables, to create a standard input for model counters that support projected model counting (Section3.3). Once the backend counter finishes, the GUI reports detailed results, including the SAT formula generation time, counting time, the model count, and whether the count is as expected (if the `expect` clause is used). In addition, the original output of the individual model counter is also provided as a link to a text file.

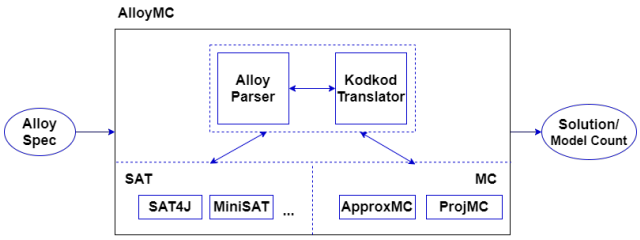

Figure 2: AlloyMC architecture

```
cmdDecl ::= [name ":"] ("run"|"check"|"count")
            (name|block) scope
scope ::= "for" number                    [expect]
scope ::= "for" number "but" typescope,+ [expect]
scope ::= "for"              typescope,+ [expect]
scope ::=                                 [expect]
expect ::= "expect" [op] num
typescope ::= ["exactly"] number [name|"int"|"seq"]
op ::= ">=" | "<=" | "<" | ">" | "=" | "!="
num ::= number | number "*" number "^" number
```

Figure 3: Extended Alloy grammar for AlloyMC

3.2 Alloy Language Extension for Counting

We support model counting for Alloy by introducing to the Alloy language[1] the `count` command, which we design following the spirit of Alloy's `run` command. Figure 3 shows the production rules of the Alloy grammar that are modified by AlloyMC. The key modifications are: 1) `count` is a new keyword (production `cmdDecl`); 2) relational operator can be used in the `expect` clause (for the `count` command), so the user can specify the expected model count, e.g., is "`<= 8`"; and 3) expressions of the form $a \times b^c$, as used by model counters, are supported to make it easier for the user to provide an expected model count (or bound) when the count is a large number, which is often the case for complex problems that arise in practice. Thus, the `count` command allows the user to specify expected lower/upper bound for the count (using >=, <=, < or >), expected exact count (using =), or expected impossible count (using `!=`). Following the default behavior of Alloy's `run` command, we ignore the `expect` clause when the given expected count is negative.

To illustrate, recall the example in Figure 1. It lists 5 count commands for the acyclic linked list specification (Lines 10–14). The command `cmd1` counts the number of acyclic lists with up to 5 nodes. Commands `cmd2`, `cmd3`, `cmd4`, and `cmd5` illustrate some of the different ways of writing count command. Since the expected value for `cmd5` is negative, this command is equivalent to `cmd1`.

3.3 Projected Model Counters in AlloyMC

As stated before, the model counting problems that arise from Alloy specifications are projected model counting problems. The reason is that the translation of Alloy to CNF introduces auxiliary variables that exist in the CNF formula but are present purely to enable the conjunctive normal form of the formula. For the example specification shown in Figure 1, there are 36 primary variables in the translated SAT formula that directly encodes the Alloy specification with respect to the scope of 5 (1 variable for `List`, 5 for `Node`, 5 for `header`, and 25 for `link`). In contrast, once converted to CNF, there

[1]The original grammar for Alloy (version 4) is briefly summarized here: http://alloytools.org/download/alloy4-grammar.txt

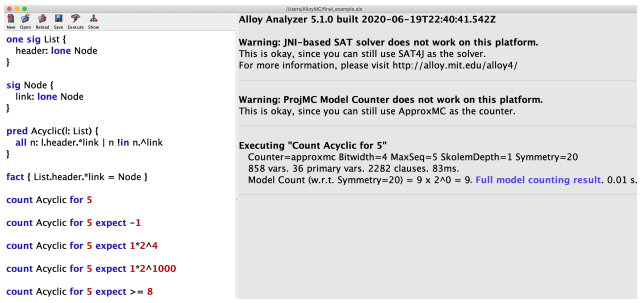

Figure 4: Alloy GUI

are 664 variables. The conversion to CNF creates a formula that is *equi-satisfiable* but *not* (necessarily) equivalent, and hence the model count of the formula before conversion can be different from the model count after conversion. However, the projected model count of the CNF formula with respect to the (primary) variables in the original SAT formula is the same as the model count of the original formula.

AlloyMC employs two state-of-the-art robust projected model counters, namely ApproxMC [13] and ProjMC [21] for approximate and exact model counting, respectively. Note that we design AlloyMC to be extensible so other projected model counters can be added in the future. ApproxMC, initially proposed in 2013, employs universal hash functions to efficiently partition the solution space into roughly equal small cells. The model count of each randomly chosen cell, which is used for approximating the overall count, is obtained by selective solution enumeration using CryptoMinisat [30]. ApproxMC is now in its third generation, called ApproxMC3 [29] (the version we use for AlloyMC) with its source code publicly available[2]. ProjMC, one of the most recently published model counters, uses a recursive algorithm and employs a difsjunctive decomposition method with search for disjoint components. Only the binary code of ProjMC under Linux OS is now available[3].

4 IMPLEMENTATION AND USAGE

The AlloyMC tool is currently available under Linux (Ubuntu), Mac, and Windows 64-bit operating systems. In this section, we describe key parts of the implementation, the overall workflow, and how to use AlloyMC as a convenient wayeof specifying and solving model counting problems in Alloy.

At the front-end level, AlloyMC builds on the original Alloy GUI and makes a few modifications to support model counting (Figure 4). Alloy GUI has three parts: menu bar (with a shortcut toolbar), coding area, and log panel. The menu bar allows choosing key settings of the tool. In the coding area, users write their Alloy specifications. The log panel shows the analysis results.

We modified the menu bar so users can select the model counter to use, and also adjust the *symmetry breaking* setting [31] to be used in model counting. Originally in Alloy, when `run` or `check` commands are executed, Alloy adds *symmetry breaking predicates* to the formula with the goal of assisting the backend SAT solver to more efficiently solve the satisfiability problem [2]. While the addition of symmetry breaking predicates in Alloy generally makes backend SAT solvers more efficient, an issue with these predicates

[2]ApproxMC3 source code is available at https://github.com/meelgroup/ApproxMC
[3]ProjMC binary code is available at http://www.cril.univ-artois.fr/kc/projmc.html

Table 1: AlloyMC evaluation results on 10 Alloy models (time in seconds)

subject	#primary vars	#vars	#clauses	CNF time [s]	ProjMC		ApproxMC	
					time [s]	count	time [s]	count
addressBook1g/delUndoesAdd	36	1,567	2,214	0.26	1.34	5,429	0.79	5,632
farmer/solvePuzzle	80	1,137	3,011	0.41	0.08	2	0.01	2
farmer/NoQuantumObjects	80	1,177	3,079	0.81	1.39	512	0.41	544
hotel3/NoBadEntry	407	8,640	21,479	0.15	TO	-	18.4	18,176
life/Square	95	4,666	13,803	0.27	5.04	962	8.48	976
life/Show	324	27,572	93,759	0.29	TO	-	TO	-
life/interesting	90	3,664	11,372	0.49	TO	-	30.39	12544
messaging/SomeState	76	609	731	0.5	0.27	88,505,010	1.63	88,080,384
messaging/OutOfOrder	496	4,513	7,007	0.35	TO	-	TO	-
p300-hotel/NoIntruder	135	3,798	6,638	0.9	34.66	4,662	3.77	4,608
queens/NQueensProblem	8,192	141,969	378,236	7.62	28.35	92	24.72	92
bst/RepOk	425	10,210	27,547	0.24	31.41	1,430	23.73	1,408
dllist/RepOk	141	4,341	12,016	0.06	19.58	5,080	2.75	4,736
rbt/RepOk	297	10,945	26,406	1.74	0	0	0.12	0

is that they modify the model count since they remove some symmetric solutions. Moreover, Alloy's symmetry breaking is heuristic, and there is no direct way to compute the precise impact that the additional predicates have on the model count. Thus, in the context of model counting, it is necessary to allow the user to choose whether or not they want symmetry breaking predicates. AlloyMC adds an option to support symmetry breaking setting (off, default, or twice the default) in the *Options* menu in the menu bar.

AlloyMC enhances the coding area to support the extended grammar. Moreover, AlloyMC summarizes the result of model counting in the log panel. The running information is shown once the counter starts, and the count results are shown once the counter finishes. Moreover, warning messages are shown if the counter is not supported under the current environment. In addition, a link to a text file that contains verbatim output of the used model counter is added, so the user can view the detailed output if desired. AlloyMC allows the user to stop the counter at any time during the execution, using the existing "Stop" button in the shortcut toolbar.

Our implementation also made AlloyMC easy to install so the user does not have to separately install its various dependencies (e.g., the counters and the dependent libraries). We adjusted Bnd [4] files to distribute platform-related executable files and dynamic libraries in the final JAR file. Bnd is a widely used build tool with the goal of generating JAR files for complicated Java projects; it is also utilized by the original Alloy analyzer. The dynamic libraries are what model counters need to run with. Once the user executes the tool, the platform-related executable files and dynamic libraries are copied from the JAR file to the tool's temporary directory. The environment variable ("`LD_LIBRARY_PATH`" for Linux; and "`DYLD_FALLBACK_LIBRARY_PATH`" for Mac) for dynamic library search is set for the counters automatically. To run AlloyMC, users only need to download and run the JAR file.

5 EXPERIMENTS

For evaluation subjects, we take 10 Alloy specifications as our subjects including 6 specifications randomly taken from Alloy standard distribution (addressBook1g, farmer, hotel3, life, messaging, and p300-hotel), and 4 specifications randomly taken from a previous model counting study [35] (queens, bst, dllist, and rbt). We change each run and check command in original Alloy specification into a count command, keeping the original scope. We made "rbt" specification an exception by changing its scope to explicitly make "RepOk"predicate unsatisfiable, to see how AlloyMC performs for an unsatisfiable formula. Table 1 shows subjects with specification names and corresponding predicate names in the command.

In the experiments, we take each Alloy specification and manually run it in AlloyMC GUI. We set up time-out as 60 seconds to approximate users' tolerable waiting time. In Table 1, we report translated CNF formula info consisting of primary variable number (#primary vars), total variable number (#vars), total clause number (#clause), and CNF generation time; we also report model counter (ApproxMC and ProjMC) outputs including the running time and model count. "TO" indicates the subject timed out, and "-" indicates unknown count. All the experiments were performed on a machine with 4-core, 2.7 GHz Intel Core i7 CPU with 16 GB memory.

Overall, AlloyMC correctly parsed the input Alloy specification with the extended grammar. The CNF formula is generated quickly and solved correctly by both counters. AlloyMC also performs correctly under the unsatisfiable case ("rbt" specification) by returning 0 model count. We also observe that users can get the results for most commands within our time-out limit, even for a case when the model count is large ("SomeSate" predicate in "messaging" specification); note that it would be infeasible to get such a large model count using the solution enumeration in the original Alloy tool.

6 CONCLUSION

This paper introduced the AlloyMC tool that, to our knowledge, adds the first model counting backend to Alloy. AlloyMC supports two state-of-the-art model counters: ApproxMC for approximate model counting, and ProjMC for exact model counting. We believe AlloyMC promises more effective applications of Alloy in domains where multiple solutions are required, e.g., for bounded-exhaustive testing. The knowledge of model counts can also guide the users to refine their specifications, e.g., by adding constraints to reduce the number of solutions and create a high quality test suite that can both be enumerated as well as executed feasibly.

ACKNOWLEDGMENTS

This work was funded in part by NSF grants CNS-1646305, CCF-1718903, and CCF-1956374.

[4] Bnd/BndTools source code and usage are available at https://github.com/bndtools/bnd

REFERENCES

[1] Devdatta Akhawe, Adam Barth, Peifung E Lam, John Mitchell, and Dawn Song. 2010. Towards a formal foundation of web security. In *2010 23rd IEEE Computer Security Foundations Symposium*. IEEE, 290–304.

[2] Fadi A Aloul, Karem A Sakallah, and Igor L Markov. 2006. Efficient symmetry breaking for boolean satisfiability. *IEEE Trans. Comput.* 55, 5 (2006), 549–558.

[3] Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kastner. 2010. Detecting dependences and interactions in feature-oriented design. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*. IEEE, 161–170.

[4] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. 2015. Automata-based model counting for string constraints. In *International Conference on Computer Aided Verification*. Springer, 255–272.

[5] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. 2003. Algorithms and complexity results for# SAT and Bayesian inference. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.* IEEE, 340–351.

[6] Biljana Bajić-Bizumić, Claude Petitpierre, Hieu Chi Huynh, and Alain Wegmann. 2013. A model-driven environment for service design, simulation and prototyping. In *International Conference on Exploring Services Science*. Springer, 200–214.

[7] Kacper Bąk, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Feature and meta-models in Clafer: mixed, specialized, and coupled. In *International Conference on Software Language Engineering*. Springer, 102–122.

[8] Vaishak Belle, Andrea Passerini, and Guy Van Den Broeck. 2015. Probabilistic Inference in Hybrid Domains by Weighted Model Integration. In *Proceedings of the 24th International Conference on Artificial Intelligence* (Buenos Aires, Argentina) *(IJCAI'15)*. AAAI Press, 2770–2776.

[9] Armin Biere, Marijn Heule, and Hans van Maaren. 2009. *Handbook of satisfiability*. Vol. 185. IOS press.

[10] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsch. [n.d.]. Model Counting. *11.12. 1. Autarkies and qualitative matrix analysis* ([n. d.]), 633.

[11] Mateus Borges, Quoc-Sang Phan, Antonio Filieri, and Corina S Păsăreanu. 2017. Model-counting approaches for nonlinear numerical constraints. In *NASA Formal Methods Symposium*. Springer, 131–138.

[12] Fabian Büttner, Marina Egea, Jordi Cabot, and Martin Gogolla. 2012. Verification of ATL transformations using transformation models and model finders. In *International Conference on Formal Engineering Methods*. Springer, 198–213.

[13] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. 2013. A Scalable Approximate Model Counter. In *Proc. of CP*. 200–216.

[14] Carmel Domshlak and Jörg Hoffmann. 2007. Probabilistic planning via heuristic forward search and weighted model counting. *Journal of Artificial Intelligence Research* 30 (2007), 565–620.

[15] William Eiers, Seemanta Saha, Tegan Brennan, and Tevfik Bultan. 2019. Subformula caching for model counting and quantitative program analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 453–464.

[16] Antonio Filieri, Corina S Păsăreanu, and Willem Visser. 2013. Reliability analysis in symbolic pathfinder. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 622–631.

[17] Carla P Gomes, Ashish Sabharwal, and Bart Selman. 2008. Model counting. (2008).

[18] Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *TOSEM* (2002).

[19] Eunsuk Kang and Daniel Jackson. 2008. Formal modeling and analysis of a flash filesystem in Alloy. In *International Conference on Abstract State Machines, B and Z*. Springer, 294–308.

[20] Vladimir Klebanov. 2012. Precise quantitative information flow analysis using symbolic model counting. *QASA* (2012).

[21] Jean-Marie Lagniez and Pierre Marquis. 2019. A Recursive Algorithm for Projected Model Counting. *AAAI* 33 (2019), 1536–1543.

[22] Michael L Littman, Stephen M Majercik, and Toniann Pitassi. 2001. Stochastic boolean satisfiability. *Journal of Automated Reasoning* 27, 3 (2001), 251–296.

[23] Suhas Pai, Yash Sharma, Sunil Kumar, Radhika M Pai, and Sanjay Singh. 2011. Formal verification of OAuth 2.0 using Alloy framework. In *2011 International Conference on Communication Systems and Network Technologies*. IEEE, 655–659.

[24] James D. Park. 2002. MAP Complexity Results and Approximation Methods. In *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence* (Alberta, Canada) *(UAI'02)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 388–396.

[25] Quoc-Sang Phan and Pasquale Malacaria. 2014. Abstract model counting: a novel approach for quantification of information leaks. In *9th ACM Symposium on Information, Computer and Communications Security*. 283–292.

[26] Dan Roth. 1996. On the hardness of approximate reasoning. *Artificial Intelligence* 82, 1-2 (1996), 273–302.

[27] Tian Sang, Paul Beame, and Henry A Kautz. 2005. Performing Bayesian inference by weighted model counting. In *AAAI*, Vol. 5. 475–481.

[28] Danhua Shao, Sarfraz Khurshid, and Dewayne E Perry. 2007. Whispec: White-box testing of libraries using declarative specifications. In *Proceedings of the 2007 Symposium on Library-Centric Software Design*. 11–20.

[29] Mate Soos and Kuldeep S. Meel. 2019. BIRD: Engineering an Efficient CNF-XOR SAT Solver and its Applications to Approximate Model Counting. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*.

[30] Mate Soos, Karsten Nohl, and Claude Castelluccia. 2009. Extending SAT Solvers to Cryptographic Problems. In *Theory and Applications of Satisfiability Testing (SAT)*. 244–257.

[31] Emina Torlak and Daniel Jackson. 2007. Kodkod: A Relational Model Finder. In *TACAS*.

[32] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2018. Checkmate: Automated synthesis of hardware exploits and security litmus tests. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 947–960.

[33] Caroline June Trippel. 2019. *Concurrency and Security Verification in Heterogeneous Parallel Systems*. Ph.D. Dissertation. Princeton University.

[34] E. Uzuncaova, S. Khurshid, and D. Batory. 2010. Incremental Test Generation for Software Product Lines. *IEEE Transactions on Software Engineering* 36, 3 (2010), 309–322.

[35] 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 *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 115–134.