AlloyMC: Alloy Meets Model Counting

Jiayi Yang  
University of Texas at Austin, USA

Darko Marinov  
University of Illinois at Urbana-Champaign, USA

Wenxi Wang  
University of Texas at Austin, 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 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

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://youtube.be/loglrJ8kHf4.

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 be 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 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 1’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 1 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 1 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 (Section 3.3). 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 1: Acyclic linked list Alloy model](https://example.com/figure1.png)
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 \texttt{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 \texttt{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 “\(\leq 8\)”\(^2\); and 3) expressions of the form \(a \times b\), 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 \(\geq, \leq, <, \text{or} >\))\(^3\), expected exact count (using \(\approx\)), or expected impossible count (using \(!=\)). Following the default behavior of Alloy’s \texttt{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 \texttt{cmd1} counts the number of acyclic lists with up to 5 nodes. Commands \texttt{cmd2}, \texttt{cmd3}, \texttt{cmd4}, and \texttt{cmd5} illustrate some of the different ways of writing count command. Since the expected value for \texttt{cmd5} is negative, this command is equivalent to \texttt{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 \texttt{List}, 5 for \texttt{Node}, 5 for header, and 25 for \texttt{link}). In contrast, once converted to CNF, there are 664 variables. The conversion to CNF creates a formula that is \textit{equi-satisfiable} but \textit{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\(^1\) and ProjMC\(^2\) 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\(^3\), 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\(^4\). ProjMC\(^5\), one of the most recently published model counters, uses a recursive algorithm and employs a disjunctive decomposition method with search for disjoint components. Only the binary code of ProjMC under Linux OS is now available\(^6\).

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 way of 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 \textit{symmetry breaking} setting\(^7\) to be used in model counting. Originally in Alloy, when \texttt{run} or check commands are executed, Alloy adds \textit{symmetry breaking predicates} to the formula with the goal of assisting the backend SAT solver to more efficiently solve the satisfiability problem\(^8\). While the addition of symmetry breaking predicates in Alloy generally makes backend SAT solvers more efficient, an issue with these predicates

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

\(^{2}\)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>
<thead>
<tr>
<th>subject</th>
<th>#primary vars</th>
<th>#vars</th>
<th>#clauses</th>
<th>CNF time [s]</th>
<th>ProjMC time [s]</th>
<th>ApproxMC time [s]</th>
<th>count</th>
<th>ApproxMC count</th>
</tr>
</thead>
<tbody>
<tr>
<td>addressBook1g/delUndoes/Add</td>
<td>36</td>
<td>1,567</td>
<td>2,214</td>
<td>0.26</td>
<td>1.34</td>
<td>5,429</td>
<td>0.79</td>
<td>5,632</td>
</tr>
<tr>
<td>farmer/so1vePuzzle</td>
<td>80</td>
<td>1,137</td>
<td>3,011</td>
<td>0.41</td>
<td>0.08</td>
<td>2</td>
<td>0.01</td>
<td>2</td>
</tr>
<tr>
<td>farmer/NoQuantumObjects</td>
<td>300</td>
<td>1,177</td>
<td>3,079</td>
<td>0.81</td>
<td>1.39</td>
<td>512</td>
<td>0.41</td>
<td>544</td>
</tr>
<tr>
<td>hotel3/NoRadEntry</td>
<td>407</td>
<td>8,640</td>
<td>21,479</td>
<td>0.15</td>
<td>TO</td>
<td>-</td>
<td>18.4</td>
<td>18,176</td>
</tr>
<tr>
<td>life/3Square</td>
<td>95</td>
<td>4,666</td>
<td>13,803</td>
<td>0.27</td>
<td>5.94</td>
<td>962</td>
<td>8.48</td>
<td>976</td>
</tr>
<tr>
<td>life/Show</td>
<td>324</td>
<td>27,572</td>
<td>93,759</td>
<td>0.29</td>
<td>TO</td>
<td>-</td>
<td>10.2</td>
<td>976</td>
</tr>
<tr>
<td>life/interesting</td>
<td>90</td>
<td>3,664</td>
<td>11,372</td>
<td>0.49</td>
<td>TO</td>
<td>-</td>
<td>39.39</td>
<td>12544</td>
</tr>
<tr>
<td>messaging/SomeState</td>
<td>76</td>
<td>609</td>
<td>731</td>
<td>0.5</td>
<td>0.27</td>
<td>88,505,010</td>
<td>1.63</td>
<td>88,080,384</td>
</tr>
<tr>
<td>messaging/OutOfOrder</td>
<td>496</td>
<td>4,513</td>
<td>7,007</td>
<td>0.35</td>
<td>TO</td>
<td>-</td>
<td>TO</td>
<td>-</td>
</tr>
<tr>
<td>p300-hotel/NoIntruder</td>
<td>135</td>
<td>3,798</td>
<td>6,638</td>
<td>0.9</td>
<td>34.66</td>
<td>4,662</td>
<td>3.77</td>
<td>4,688</td>
</tr>
<tr>
<td>queens/NQueensProblem</td>
<td>8,192</td>
<td>141,969</td>
<td>378,236</td>
<td>7.62</td>
<td>28.35</td>
<td>92</td>
<td>24.72</td>
<td>92</td>
</tr>
<tr>
<td>bst/RepOk</td>
<td>425</td>
<td>10,210</td>
<td>27,547</td>
<td>0.24</td>
<td>31.41</td>
<td>1,430</td>
<td>23.73</td>
<td>1,408</td>
</tr>
<tr>
<td>dlist/RepOk</td>
<td>141</td>
<td>4,341</td>
<td>12,016</td>
<td>0.06</td>
<td>19.58</td>
<td>5,080</td>
<td>2.75</td>
<td>4,736</td>
</tr>
<tr>
<td>rbt/RepOk</td>
<td>297</td>
<td>10,945</td>
<td>26,406</td>
<td>1.74</td>
<td>0</td>
<td>0</td>
<td>0.12</td>
<td>0</td>
</tr>
</tbody>
</table>

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 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, dlist, 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 unsatisfisible 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 (“SomeState” predicate in “messageing” 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.
REFERENCES


