A Case for Efficient Solution Enumeration

Sarfraz Khurshid   Darko Marinov   Ilya Shlyakhter   Daniel Jackson

MIT Computer Science and Artificial Intelligence Laboratory
200 Technology Square
Cambridge, MA 02139 USA
{khurshid,marinov,ilya_shl,dnj}@lcs.mit.edu

Abstract. SAT solvers have been ranked primarily by the time they take to find a solution or show that none exists. And indeed, for many problems that are reduced to SAT, finding a single solution is what matters. As a result, much less attention has been paid to the problem of efficiently generating *all* solutions.

This paper explains why such functionality is useful. We outline an approach to automatic test case generation in which an invariant is expressed in a simple relational logic and translated to a propositional formula. Solutions found by a SAT solver are lifted back to the relational domain and reified as test cases. In unit testing of object-oriented programs, for example, the invariant constrains the representation of an object; the test cases are then objects on which to invoke a method under test. Experimental results demonstrate that, despite the lack of attention to this problem, current SAT solvers still provide a feasible solution.

In this context, symmetry breaking plays a significant, but different role from its conventional one. Rather than reducing the time to finding the first solution, it reduces the number of solutions generated, and improves the quality of the test suite.

1 Introduction

Advances in SAT technology have enabled applications of SAT solvers in a variety of domains, e.g., AI planning [15], hardware verification [7], software design analysis [16], and code analysis [27]. These applications typically use a solver to find one solution, e.g., one plan that achieves a desired goal or one counterexample that violates a correctness property. Hence, most modern SAT solvers are optimized for finding one solution, or showing that no solution exists. That is also how the SAT competitions [1] rank solvers.

We have developed an unconventional application of SAT solvers, for software testing [20]. Our application requires a solver, such as mChaff [21] or relsat [4], that can enumerate *all* solutions. We find it surprising that the currently available versions of most modern SAT solvers, including zChaff [21], BerkMin [12], Limmat [6], and Jerusat [22], do not support solution enumeration at all[1], let

[1] Support in zChaff is under development and available in an internal version. Support in BerkMin is planned for the next version. There is no plan to add support to Jerusat. (Personal communication with the authors of the solvers.)

alone optimize it. We hope that our application can motivate research in solution enumeration.

Software testing is the most widely used technique for finding bugs in programs. It is conceptually simple: just create a test suite, i.e., a set of test inputs, run them against the program, and check if each output is correct. However, testing is typically a labor intensive process, involving mostly manual generation of test inputs, and accounts for about half the total cost of software development and maintenance [5]. Moreover, inputs for modern programs often have complex structural constraints, which makes manual generation of high quality test suites impractical. Automating testing would not only reduce the cost of producing software but also increase the reliability of modern software.

We have developed the TestEra framework [20] for automated specification-based testing [5] of Java programs. To test a method, the user provides a specification that consists of a precondition (which describes allowed inputs to the method) and a postcondition (which describes the expected outputs). TestEra uses the precondition to automatically generate a test suite of *all* test inputs up to a given *scope*; a test input is within a scope of k if at most k objects of any given class appear in it. TestEra executes the method on each input, and uses the postcondition as a test oracle to check the correctness of each output.

TestEra specifications are first-order logic formulas. As an enabling technology, TestEra uses the Alloy toolset. Alloy [13] is a first-order declarative language based on sets and relations. The Alloy Analyzer [14] is an automatic tool that finds *instances* of Alloy specifications, i.e., assignments of values to the sets and relations in the specification such that its formulas evaluate to true. The analyzer finds an instance by: 1) translating Alloy specification into boolean satisfiability formula, 2) using an off-the-shelf SAT solver to find a solution to the formula, and 3) translating the solution back into sets and relations. The analyzer can enumerate all instances (within a given scope) using a SAT solver that supports enumeration, e.g., mChaff or relsat. The analyzer generates *complete* assignments: if the underlying SAT solver generates a solution with "don't care" bits, the analyzer grounds these bits out.

TestEra translates Alloy instances into test inputs that consist of Java objects; notice that the grounding out of "don't care" bits is necessary to build Java objects with fields that are properly initialized. Some inputs are *isomorphic*, i.e., they only differ in the identity of their objects. For example, consider a singly linked list of nodes that contain elements; two lists that have the same elements (or more precisely, isomorphic elements) in the same order are isomorphic irrespective of the node identities. It is desirable to consider only non-isomorphic inputs; it reduces the time to test the program, without reducing the possibility to detect bugs, because isomorphic test inputs form a "revealing subdomain" [28], i.e., produce identical results. The analyzer has automatic symmetry breaking [23] that eliminates many isomorphic inputs; we discuss this further in Section 3.1.

We initially used TestEra to check several Java programs. TestEra exposed bugs in a naming architecture for dynamic networks [17] and a part of an earlier version of the Alloy Analyzer [20]; these bugs have now been corrected. We have also used TestEra to systematically check methods on Java data structures, such as from the Java Collection Framework [26]. More recently, we have applied TestEra to test a C++ implementation of a fault-tree solver [11].

It is worth emphasizing that our application requires solutions that make complete assignments to primary (independent) variables. The main requirement of TestEra is efficient generation of all these solutions. Since storage of solutions has not (yet) been an issue, generating an implicit representation or a "cover" is not necessary. Moreover, testing necessitates generation of actual solutions and not just their representation. However, a solver that produces only prime implicants can be used as an intermediate step in generating all solutions.

In previous work [20], we presented TestEra as an application of SAT solvers in software testing. This paper makes the following new contributions:

– We describe a compelling application of SAT solvers that suggests that solution enumeration is an important feature that merits research in its own right. To the best of our knowledge, this is the first such application in software testing.
– We provide a set of formulas that can be used to compare different solvers in their enumeration. Our formulas fall into the (satisfiable) "industrial" benchmarks category for SAT competitions [1] and are available online at:

  `http://mulsaw.lcs.mit.edu/alloy/sat03/index.html`

  We also provide the expected number of solutions, which can help in testing a solver's solution enumeration.
– We provide a performance comparison between mChaff and relsat in enumerating a variety of benchmark data structures.
– We show how TestEra users can completely break symmetries, so that each solution of a boolean formula corresponds to a non-isomorphic test input.

2 Test generation for modern software

Structurally complex data abounds in modern software. A textbook example is a data structure such as red-black trees [9] that implement balanced binary search trees. Another example is intentional names used in the intentional naming systems [2] for dynamic networks; an intentional name describes properties of a service by a hierarchical arrangement of *attributes* and *values*, which enables clients to access services by describing the desired functionality without a priori knowing service locations. Fault trees used in fault tree analysis systems [11] are also complex structures; a fault tree models system failure by relating it to failure of basic events in the system, and a fault tree analyzer computes the likelihood of system failure in the input fault tree over a given period of time. What makes such data complex is not only organization but also their structural constraints. For example, for red-black trees, one such constraint is that the number of black nodes along any path from root to a leaf is the same.

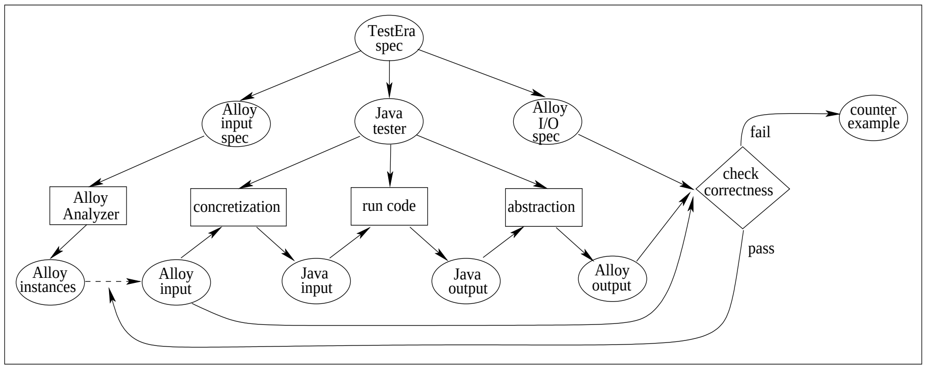

Fig. 1. Basic TestEra framework

Automating generation of input data is the key issue in automating systematic testing of software with structurally complex data. Generation at the representation level by a random assignment of values to object fields is infeasible since the ratio of the number of valid structures and the number of candidate structures tends to zero. A random assignment is very likely to violate (at least) one of the constraints. Generation at an abstract level using a construction sequence (e.g., building a red-black tree with 3 nodes by inserting sequentially 3 elements into an empty tree) is also inefficient: systematic generation of all inputs within a given input size can result in a very large number of construction sequences, most of which generate isomorphic structures. For example, to build all red-black trees with 10 nodes, this approach may require 10! or about 3.6×10^6 sequences, whereas there are only 240 non-isomorphic red-black trees with 10 nodes.

3 TestEra

The TestEra framework [20] provides a novel SAT-based approach for systematic generation of complex structures and automates specification-based testing [5] of Java programs.

Figure 1 illustrates the main components of the TestEra framework. Given a method precondition or an input specification (which describes the constraints that define valid method inputs) in Alloy, TestEra uses the Alloy Analyzer to generate all instances that satisfy the precondition. TestEra automatically *concretizes* these instances to create Java objects that form the test inputs for the method under test. TestEra executes the method on each input and automatically *abstracts* each output to an Alloy instance. TestEra then uses the analyzer to check if the input/output pair satisfy the constraints represented by the postcondition or the input/output specification (which describes the expected outputs). If the postcondition constraints are violated, TestEra reports a concrete counterexample, i.e., an input/output pair that is a witness to the violation. TestEra can graphically display the counterexample, e.g., as a heap snapshot, using the visualization facility of the analyzer.

To perform translations (concretizations and abstractions) between Alloy instances that the analyzer generates and Java objects that are inputs to, or outputs of, the method, TestEra automatically generates a test driver for a given method and relevant classes.

3.1 Symmetry breaking

The analyzer adapts symmetry-breaking predicates [10] to reduce the total number of instances generated: the original boolean formula that corresponds to the Alloy specification is conjoined with additional clauses in order to generate only a few instances from each isomorphism class [23]. There is a trade-off, however: the more clauses that the analyzer generates, the more symmetries it breaks. But the larger boolean formula can become so large so that solving takes significantly more time, despite the fact that there are fewer instances. The goal of symmetry breaking in the analyzer was to make the analysis faster and not to generate exactly non-isomorphic instances. Therefore, with default symmetry breaking, it can significantly reduce the number of instances, but it is not always optimal, i.e., it may generate more than one instance from some isomorphism classes. (A discussion of algorithms for constructing symmetry-breaking clauses, such as the algorithm implemented in the Alloy Analyzer [23] or a more recent approach [3] is beyond the scope of this paper.)

The analyzer has special support for total orders: for each set $\{a_1, \ldots, a_n\}$ of n elements that is declared to have a total order, the analyzer generates only one order $\{\langle a_1, a_2\rangle, \langle a_2, a_3\rangle, \ldots, \langle a_{n-1}, a_n\rangle\}$, out of $n!$ (isomorphic) orders. This support has previously been used for faster analysis. We show in Section 4.3 how TestEra and Alloy users can also use total orders to constrain specifications so that the analyzer generates exactly one instance from each isomorphism class. Conceptually, the idea is to add constraints to ensure that the analyzer generates, from each isomorphism class, only the instance that is the *smallest* with respect to the total orders on the sets whose elements appear in the instance.

4 Non-isomorphic generation

We next give a simple example that we use throughout this section to explain the fundamental aspects of TestEra's SAT-based generation of non-isomorphic inputs. Consider the following Java code that declares a binary tree and its `removeRoot` method, which removes the root node of its input tree and arranges any remaining nodes as a tree:

```
package testera.example;

class BinaryTree {
    Node root; // root node
    int size;  // number of nodes in the tree
    static class Node {
        Node left;  // left child
        Node right; // right child
    }

    void removeRoot() { ... }
}
```

Each object of the class `BinaryTree` represents a binary tree; objects of the inner class `Node` represent nodes of the trees. The method `removeRoot` has one input (i.e., the implicit `this` argument), which is a `BinaryTree`. Let us consider systematic input generation to test this method.

4.1 Specification

For the classes declared in the Java code above, TestEra produces the following Alloy specification:

```
module testera/example/BinaryTree

sig BinaryTree {
  root: option Node,
  size: Integer
}

sig Node {
  left: option Node,
  right: option Node
}
```

The declaration `module` names the specification. The keyword `sig` introduces a *signature* representing a set of indivisible atoms. We use Alloy atoms to model objects of the corresponding classes. Each signature can have *field* declarations that introduce relations between atoms. By default, fields are total functions; `size` is a total function from `BinaryTree` to `Integer`, where `Integer` is a predefined signature. The modifier `option` is used for partial functions (and the modifier `set` for general relations); e.g., `root` is a partial function from `BinaryTree` to `Node`. Partiality is used to model `null`: when the Java field `root` of some object `b` has the value `null`, i.e., points to no object, then the function `root` does not map the atom corresponding to `b` to any other atom.

The method `removeRoot` has only the implicit `this` argument, which is a `BinaryTree`. We consider a simple specification for this method: both precondition and postcondition require only that `this` satisfy the *representation invariant* (also known as a "class invariant") [19] for `BinaryTree`. For `BinaryTree`, the invariant requires that the graph of nodes reachable from `root` indeed be a tree (i.e., have no cycles) and that the `size` be correct; in Alloy, it can be written as follows:

```
fun repOk(t: BinaryTree) {
  all n: t.root.*(left + right) {
    n !in n.^(left + right) // no directed cycle
    sole n.~(left + right)  // at most one parent
    no n.left & n.right }   // distinct children
  t.size = #(t.root.*(left + right)) }   // size is consistent
```

The Alloy *function* `repOk` records constraints that can be invoked elsewhere in the specification. This function has an argument `t`, which is a `BinaryTree`. The function body contains two formulas, implicitly conjoined. The first formula constrains `t` to be a valid binary tree. The expression `left + right` denotes the union of relations `left` and `right`; the prefix operator '`*`' is reflexive transitive closure, and the dot operator '`.`' is relational composition. The expression `root.*(left + right)` denotes the set of all nodes reachable from `root`. The quantifier `all` denotes universal quantification: the formula `all n: S { F }` holds iff the formula `F` holds for `n` bound to each element in the set `S`. The operators '`^`', '`~`', and '`&`' denote transitive closure, transpose, and intersection, respectively. The formulas `sole S` and `no S` hold iff the set `S` has "at most one" and "no" elements, respectively. If all nodes `n` are not reachable from itself, have at most one parent, and have distinct children, then the underlying graph is indeed a tree. The second formula constrains the `size` field of `t` to match the size of the tree; '`#`' denotes set cardinality.

4.2 Instance generation

The Alloy command `run repOk for` N `but 1 BinaryTree` instructs the analyzer to find an instance for this specification, i.e., a valuation of signatures (sets) and relations that makes the function `repOk` evaluate to true. The parameter N needs to be replaced with a specific constant that determines the scope, i.e., the maximum number of atoms in each signature, except those mentioned in the `but` clause. In our example, N determines the maximum number of `Node`s, and the instance has only one `BinaryTree`. Note that *one* instance has one tree (with several nodes) corresponding to `this` argument. Enumerating all instances using the Alloy Analyzer generates all trees with up to the given number of nodes.

The following two assignments of sets of *atoms* and *tuples* to signatures and relations in the specification represent binary tree instances for N=3:

```
Instance 1:
            BinaryTree = { BT0 }, Node = { N0, N1, N2 }

            root = { (BT0, N1) }, size = { (BT0, 3) }
            left = { (N1, N0) }, right = { (N1, N2) }

Instance 2:
            BinaryTree = { BT0 }, Node = { N0, N1, N2 }

            root = { (BT0, N0) }, size = { (BT0, 3) }
            left = { (N0, N1) }, right = { (N0, N2) }
```

These instances are isomorphic since we can generate the second instance from the first one by applying to it the permutation that swaps atoms `N0` and `N1` (i.e., `(N0 -> N1, N1 -> N0, N2 -> N2)`).

In the sequel, we focus on enumerating instances for test input generation. To compare different ways of enumeration, we consider test inputs of size exactly N. For illustration, consider $N = 5$ in our running example. There are 14 non-isomorphic trees with five nodes [24]. If we use the analyzer without any symmetry breaking, the analyzer generates 1680 instances/trees, i.e., for each of the 14 isomorphism classes, the analyzer generates all 120 distinct trees corresponding to the 5! permutations/labelings of the five nodes. If we use the analyzer with symmetry breaking [23], we can tune how many symmetries to break. With the default symmetry breaking parameter values, the analyzer generates 17 trees with five nodes. If we set these parameter values to break all symmetries, the analyzer generates exactly 14 trees. Notice, however, that doing so can make generation significantly slower since the goal of symmetry breaking in the analyzer was to make analysis faster but not to generate exactly non-isomorphic instances.

4.3 Complete symmetry-breaking using total order

We next describe an approach that uses total orders to efficiently break all symmetries in our example. Unlike the built-in symmetry breaking of the analyzer, this approach provides domain specific symmetry breaking and in particular, requires the user to manually add symmetry-breaking predicates.

The analyzer's standard library of models provides a polymorphic signature `Ord[t]`. Each instantiation of `Ord` with some set (Alloy signature) `t` imposes a total order on the elements in `t`. In consequence, these elements are not indistinguishable any more, and the analyzer does not break any symmetries on that set. However, the analyzer considers only one total order, instead of $(\#t)!$ possible total orders.

In addition to the definition of total order, the analyzer's standard library also provides several Alloy functions for totally-ordered sets. We use two of those functions in the following fact:

```
fact BreakSymmetries {
  all b: BinaryTree {
    all n: b.root.*(left + right) {
      // uses library function to instantiate Ord[Node]
      n.left.*(left + right) in OrdPrevs(n)
      n.right.*(left + right) in OrdNexts(n) } } }
```

The functions `OrdPrevs` and `OrdNexts` return the sets of all elements that are smaller and larger than the given element. A *fact* is a formula that expresses (additional) constraints on the instances. The fact `BreakSymmetries` requires that all trees in the instance (the example instances have only one tree) have nodes in an *in-order* [9]: the nodes in the left (right) subtree of the node `n` are smaller (larger) than `n` with respect to the `Ord[Node]` order. Note that the comparisons are for *node identities*, not for the values in the nodes. (For simplicity of illustration, our example does not even have values.)

We add the above fact to the specification for binary trees, effectively eliminating isomorphic instances. Indeed, the analyzer now generates exactly 14 non-isomorphic trees but it does so faster than using automatically generated symmetry-breaking predicates. In general, the user can break all symmetries by: 1) declaring that each set has a total order and 2) defining a traversal that linearizes the whole instance. The combination of the linearization and the total orders gives a lexicographic order that is used to compare instances. This process enforces generation to produce the canonical structure for each isomorphism class, but it requires the user to manually add symmetry-breaking predicates; this is straightforward to do for data structures, but it can become difficult for large complex specifications.

5 Results

We next present some performance results for solution enumeration obtained with mChaff [21]; we also compare performance of mChaff with that of relsat [4]. To discount the time it takes to write solutions to a file, we slightly modified the standard distributions of mChaff and relsat to disable solution reporting so that the solvers only report the total number of solutions found for each formula. The experiments were performed on a 1.8 GHz Pentium 4 processor with 2 GB of RAM.

5.1 Benchmarks

Table 1 presents the results of mChaff for a set of benchmark formulas that represent structural invariants. Each benchmark is named after the class for which data structures are generated; the structures also contain objects from other classes.

`BinaryTree` is our running example. `LinkedList` is the implementation of linked lists in the Java Collections Framework, a part of the standard Java libraries. This implementation uses doubly-linked, circular lists that have a `size` field and a `header` node as a sentinel node [9]. (Linked lists also provide methods that allow them to be used as stacks and queues.) `TreeMap` implements the `Map` interface using red-black trees [9]. This implementation uses binary trees with `parent` fields. Each node has a `key` and a `value`. `HashSet` implements the `Set` interface, backed by a hash table [9]; this implementation builds collision lists for buckets with the same hash code. `HeapArray` is an array-based implementation of a heap (or a priority queue) [9].

5.2 Performance of mChaff

Table 1 shows results for several input sizes for each benchmark. All scope parameters are set exactly to the given size; e.g., all lists have exactly the given

			manual symmetry breaking				automatic symmetry breaking			
benchmark	size	#prim	#vars	#clauses	#sols	time	#vars	#clauses	#sols	time
`BinaryTree`	6	86	2120	6686	132	1.05	2333	7018	357	1.50
	7	114	3165	10375	429	6.46	3439	10786	1866	7.45
	8	146	4504	15216	1430	40.46	4831	15682	10286	64.40
	9	182	7775	29618	4862	548.69	8141	30103	60616	1049.93
`LinkedList`	6	146	2017	6597	203	0.38	2520	7419	5975	3.46
	7	191	2834	9834	877	1.04	3559	11021	52392	68.71
	8	242	3837	14007	4140	4.76	4432	14939	734296	4637.99
	9	299	5852	24411	21147	36.52	6629	25630	—	—
`TreeMap`	6	203	5203	15162	20	9.10	5542	15668	322	10.85
	7	263	7578	22095	35	110.42	8076	22842	1160	69.09
	8	331	10578	30896	64	254.13	11265	31930	4185	583.62
	9	407	16111	51115	122	741.55	17017	52482	16180	3873.99
`HashSet`	6	285	5254	19079	462	6.06	5798	19865	693	7.04
	7	373	7540	28881	1716	31.52	8270	29918	3172	30.04
	8	473	10392	41430	6435	151.42	11102	42342	15011	167.30
	9	585	15380	63308	24310	511.51	16277	64441	73519	1587.72
`HeapArray`	5	56	544	1178	1919	0.55				
	6	72	704	1611	13139	5.10				
	7	90	884	2128	117562	62.62				
	8	110	1084	2735	1005075	1171.64				

Table 1. mChaff performance. All times are in seconds (of total elapsed wall-clock time). For sizes larger than presented, enumeration of solutions for automatically constructed symmetry-breaking predicates takes longer than 1 hour.

number of nodes and the elements come from a set with the given size. For each size, we use mChaff to enumerate solutions for two CNF formulas:

– one with symmetry-breaking predicates generated fully automatically (using the default values of the Alloy Analyzer);
– one with symmetry-breaking predicates added entirely manually to Alloy specifications (as described in Section 4.3).

We tabulate the number of primary variables, the total number of variables, the number of clauses, the number of solutions, and the time it takes to generate all solutions. The time shows the total elapsed time from the start of mChaff with an input being a formula file to the end of generation of all solutions (without writing them in a file). It is worth noting that the time to generate solutions often accounts for more than one-half of the time TestEra takes to test a benchmark data structure implementation [20]; thus, improving efficiency of solution enumeration can significantly improve TestEra's performance.

For `BinaryTree`, `LinkedList`, `TreeMap`, and `HashSet`, the numbers of non-isomorphic structures are given in the Sloane's On-Line Encyclopedia of Integer Sequences [24]. For all sizes, formulas with manually added symmetry-breaking predicates have as many solutions as the given number of structures, which shows that these predicates eliminate all symmetries. For `HeapArray`, no symmetry-breaking is required: two array-based heaps are isomorphic iff they are identical, since they consist only of integers (i.e., array indices and heap elements) that are not permutable. In TestEra, it is very desirable to generate only non-isomorphic inputs since without breaking isomorphisms it would be impractical to systematically test on all inputs. The factor by which the total number of solutions (including isomorphic solutions) is more than the total number of non-isomorphic solutions, is exponential in the input size. For example, for `TreeMap` and size nine, there are more than 44 million total solutions.

In all cases, formulas with automatic symmetry breaking (using default parameter values) have more solutions than formulas with manual symmetry breaking. Also, in most cases it takes longer to generate the solutions for formulas with automatic symmetry breaking; a simple reason for this is that enumerating a larger number of solutions usually takes a larger amount of time. This is not always the case, however: for `HashSet` and `TreeMap` of size seven, it takes less time to enumerate more solutions. This illustrates the general trade-off in (automatic) symmetry breaking: adding more symmetry-breaking predicates can reduce the number of (isomorphic) solutions, but it makes the boolean formula larger, which can increase the enumeration time. Note that having more variables and clauses (more symmetry-breaking predicates) does not necessarily imply breaking more symmetries. For example, in all examples but `HeapArray`, manual approach generates fewer variables and clauses than the automatic approach, yet manual break more symmetries. The reason for this is that manual approach breaks independent symmetries whereas the automatic approach can break dependent symmetries. In other words, a manual predicate rules out more isomorphic instances per literal of the predicate, so it is "denser". For details, see [23]. The Alloy Analyzer allows users to tune symmetry breaking; we have experimented with different parameter values and the analyzer's default values seem to achieve a sweet spot for our benchmarks.

Note that we do not present numbers for `LinkedList` of size nine with automatic symmetry breaking; for this formula mChaff runs out of memory (2 GB). This suggests that the scheme for clause learning in mChaff [21] may need to be modified when enumerating all solutions. If there is no effective pruning or simplification of clauses added in order to exclude the already found solutions, complete solution enumeration can become infeasible. For all other benchmark formulas, mChaff is able to enumerate all solutions, even when there are more than a million of them. Test inputs that correspond to these solutions, for the sizes from the table, are sufficient to achieve complete code and branch coverage [5] for methods in the respective Java classes.

5.3 Performance comparison of mChaff with relsat

Table 2 presents the performance comparison of mChaff with relsat in enumerating *all* solutions for benchmark formulas with manually added symmetry breaking constraints. Enumeration by mChaff seems to be more efficient than that of relsat for the benchmark data structures. The results indicate that the techniques

benchmark	size	# sols	mChaff time	relsat time
BinaryTree	6	132	1.05	4.81
	7	429	6.46	36.28
	8	1430	40.46	268.22
LinkedList	6	203	0.38	1.21
	7	877	1.04	9.08
	8	4140	4.76	78.40
TreeMap	6	20	9.10	19.22
	7	35	110.42	128.27
	8	64	254.13	665.50
HashSet	6	462	6.06	52.49
	7	1716	31.52	475.00
	8	6435	151.42	4100.99
HeapArray	5	1919	0.55	6.71
	6	13139	5.10	77.12
	7	117562	62.62	1073.49

Table 2. Performance comparison of mChaff with relsat in solution enumeration for benchmark formulas with manually added symmetry breaking predicates. All times are in seconds (of total elapsed wall-clock time).

introduced by mChaff for finding the first solution, such as efficient unit propagation, fare reasonably well for solution enumeration. For these benchmarks, it happens that mChaff's default enumeration does not generate any solutions with "don't care" bits. However, we believe mChaff's enumeration technique of obtaining partial solutions with don't-care variables (such that any completion of the solution satisfies the CNF) would also be useful for complete enumeration as grounding out partial solutions with "don't care" bits takes time linear in the number of new solutions generated. Perhaps this technique would also outperform relsat's technique of always producing complete solutions.

5.4 Binary decision diagrams

We also conducted some very preliminary experiments using Binary Decision Diagrams (BDDs) in place of SAT solvers. Intuitively, BDDs seem attractive because they make it easier to read off all solutions, once a BDD for a formula has been obtained. Of course, the construction of a BDD itself may be infeasible and can take a long time (and exponential space). We experimented with the CUDD [25] BDD package. We constructed BDDs bottom-up, using automatic variable reordering via sifting [8], from the boolean DAGs from which the CNFs were produced. For all benchmarks, the BDD approach scaled poorly; for non-trivial sizes (over five), the BDD construction led to unmanageably large BDDs (over a million nodes) and did not finish within the alloted time limit of 10 minutes. These results are preliminary and we believe BDD experts can fine tune the performance of BDDs to provide efficient enumeration.

6 Conclusions and discussion

We have developed an unconventional application of SAT solvers, for software testing. Our application requires a solver that can enumerate all satisfying assignments; each assignment provides a (non-isomorphic) input for the program. In this context, symmetry breaking plays a significant, but different role from its conventional one: rather than reducing the time to finding the first solution, it reduces the number of solutions generated, and improves the quality of the suite of test inputs. The experimental results indicate that it is feasible to use a SAT solver to systematically generate a high quality test suite comprising of structurally complex inputs that would be hard to generate manually.

We envision various other applications of solution enumeration. One natural application is in checking certain classes of logic formulas. For example, consider the formula $\forall x \in D.P(x) \Rightarrow Q(x)$, where D is some (finite) domain, and P and Q are arbitrary predicates. We can simply use a solver to enumerate all x that satisfy $P(x)$ and then for each such x check that $Q(x)$ holds. Alternatively, we can check the validity of the implication (without requiring solution enumeration) by using a solver to directly check satisfiability of the negation: $P(x) \wedge \neg Q(x)$. Usually, the latter approach is preferred because it "opens" Q for the sophisticated optimizations that SAT solvers perform. However, when Q is a very large formula (or a formula that cannot be easily constructed explicitly), the approach with solution enumeration can work better.

Conceptually, TestEra checks that the code under test satisfies the formula $\forall i \in I.\ \mathit{pre}(i) \Rightarrow (\forall o \in O.\ \mathit{code}(i, o) \Rightarrow \mathit{post}(i, o))$, where pre is precondition, I is input domain, O is output domain, $\mathit{code}(i, o)$ denotes execution on input i that results in output o, and post is postcondition. It is possible in some cases to translate (Java) code into a formula *code* and look for a counter-example using a SAT solver (see e.g. [7, 18, 27]). These translation-based approaches typically build a formula, namely $\mathit{pre} \Rightarrow (\mathit{code} \Rightarrow \mathit{post})$, which is much bigger than the formula that TestEra builds, namely *pre*. Therefore, TestEra works better for larger code that does not have many inputs, whereas the traditional approach works better for smaller code that has many possible inputs. Notice that the traditional approach can use any SAT solver, but TestEra requires a solver that can enumerate solutions.

A desirable feature for solvers that can enumerate solutions is to allow users to control the order of enumeration. For example, for testing databases, we would like to get "similar" test cases one after the other so that we can restore the state by using "deltas" and built-in support for rollback, instead of always building the state from scratch. For checking programs, it is desirable to have a solver generate all solutions in the neighborhood (as defined by a given metric) of a particular solution; this would enable testing, for example the entire neighborhood of an execution that gets "close" to a bug.

For testing programs on input sizes for which there exist a very large number of inputs, it is desirable to have a solver that can generate solutions in a random order and thus generate a high quality input sample. Another effective approach for testing on such input sizes is to define a stronger notion of isomorphism, taking into account the domain of application or even the implementation code, and then to enumerate inputs (which are now potentially fewer in number than before). Thus even though it may seem that a solver that specializes on enumeration still must suffer due to the large number of satisfying instances (for a given size), developing such solvers is practically useful.

We hope that our work provides motivation for exploring efficient solution enumeration in modern SAT solvers.

Acknowledgments

This research was funded in part by grant 0086154 from the ITR program of the National Science Foundation and in part by the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298. We thank the anonymous referees and Viktor Kuncak for comments on earlier drafts. We also thank Sharad Malik, Darsh Ranjan, and Zhaohui Fu for discussing and sharing with us the latest implementation of zChaff, and Eugene Goldberg and Alexander Nadel for discussing plans regarding BerkMin and Jerusat.

References

1. SAT competitions for comparing state-of-the-art solvers. `http://www.satlive.org/SATCompetition/`.
2. W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proc. 17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, Dec. 1999.
3. F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Solving difficult SAT instances in the presence of symmetry. In *Proc. 39th Conference on Design Automation*, pages 731–736, 2002.
4. R. J. Bayardo Jr. and R. C. Schrag. Using CSP look-back techniques to solve real world SAT instances. In *Proc. National Conference on Artificial Intelligence*, pages 203–208, 1997.
5. B. Beizer. *Software Testing Techniques.* International Thomson Computer Press, 1990.
6. A. Biere. Limmat satisfiability solver. `http://www.inf.ethz.ch/personal/biere/projects/limmat/`.
7. A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *36thConference on Design Automation (DAC)*, New Orleans, LA, 1999.
8. K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a BDD package. In *Proc. of the Design Automation Conference (DAC)*, pages 40–45, 1990.
9. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1990.
10. J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Proc. Fifth International Conference on Principles of Knowledge Representation and Reasoning*, 1996.
11. J. B. Dugan, K. J. Sullivan, and D. Coppit. Developing a low-cost high-quality software tool for dynamic fault tree analysis. *Transactions on Reliability*, pages 49–59, 1999.

12. E. Goldberg and Y. Novikov. BerkMin: a fast and robust SAT-solver. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, Mar. 2002.
13. D. Jackson. Micromodels of software: Modelling and analysis with Alloy, 2001. `http://sdg.lcs.mit.edu/alloy/book.pdf`.
14. D. Jackson, I. Schechter, and I. Shlyakhter. ALCOA: The Alloy constraint analyzer. In *Proc. 22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, June 2000.
15. H. Kautz and B. Selman. Planning as satisfiability. In *Proc. European Conference on Artificial Intelligence (ECAI)*, Vienna, Austria, Aug. 1992.
16. S. Khurshid and D. Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *Proc. 15th IEEE International Conference on Automated Software Engineering (ASE)*, Grenoble, France, Sep 2000.
17. S. Khurshid and D. Marinov. Checking Java implementation of a naming architecture using TestEra. In S. D. Stoller and W. Visser, editors, *Electronic Notes in Theoretical Computer Science (ENTCS)*, volume 55. Elsevier Science Publishers, 2001.
18. K. R. M. Leino. A SAT characterization of boolean-program correctness. In *Proc. 10th International SPIN Workshop on Model Checking of Software*, 2003.
19. B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.* Addison-Wesley, 2000.
20. D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc. 16th IEEE International Conference on Automated Software Engineering (ASE)*, San Diego, CA, Nov. 2001.
21. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 39th Design Automation Conference (DAC)*, June 2001.
22. A. Nadel. Jerusat SAT solver. `http://www.geocities.com/alikn78/`.
23. I. Shlyakhter. Generating effective symmetry-breaking predicates for search problems. In *Proc. Workshop on Theory and Applications of Satisfiability Testing*, June 2001.
24. N. J. A. Sloane, S. Plouffe, J. M. Borwein, and R. M. Corless. The encyclopedia of integer sequences. *SIAM Review*, 38(2), 1996. `http://www.research.att.com/~njas/sequences/Seis.html`.
25. F. Somenzi. CUDD: CU decision diagram package. `http://vlsi.colorado.edu/~fabio/CUDD/`.
26. Sun Microsystems. *Java 2 Platform, Standard Edition, v1.3.1 API Specification.* `http://java.sun.com/j2se/1.3/docs/api/`.
27. M. Vaziri and D. Jackson. Checking properties of heap-manipulating procedures with a constraint solver. In *Proc. 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Warsaw, Poland, April 2003.
28. E. J. Weyuker and T. J. Ostrand. Theories of program testing and the the application of revealing subdomains. *IEEE Transactions on Software Engineering*, 6(3), May 1980.