A Case for Efficient Solution Enumeration

Sarfraz Khurshid Darko Marinov Ilya Shlyakhter Daniel Jackson

MIT Laboratory for Computer Science
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 [12] or software verification [22]. 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 present a novel application of SAT solvers in software testing. Our application requires a solver that can enumerate *all* solutions. We find it surprising that most modern SAT solvers, including zChaff [16], BerkMin [9], Limmat [4], and Jerusat [17], do not support solution enumeration at all, let alone that they do not optimize enumeration. We hope that our application can motivate research in solution enumeration.

Software testing is the most widely used method for establishing correctness of 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, manually generating test suites is tedious, and automated testing can significantly reduce the cost of software development and maintenance [3].

We have developed the TestEra framework [15] for automated specification-based testing [3] 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 [10] is a first-order declarative language based on sets and relations. Alloy Analyzer [11] is an automatic tool that finds *instances* of Alloy specifications, i.e., finds assignments of values to the sets and relations in the specification such that the specification 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 [16] or relsat [2].

TestEra translates Alloy instances into test inputs. Some of these inputs are *isomorphic*, i.e., they only differ in the identity of their objects, e.g., two lists that have the same elements (more precisely isomorphic elements) in the same order are isomorphic regardless of the identity of the actual nodes in the lists. 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" [3], i.e., produce identical results. The analyzer has automatic symmetry breaking [18] that eliminates many isomorphic inputs; we discuss this further in Section 2.1.

We initially used TestEra to check several Java programs. TestEra exposed bugs in a naming architecture for dynamic networks [13] and a part of the Alloy-alpha analyzer [15]; 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 [21]. More recently, we have applied TestEra to test a C++ implementation of a fault-tree solver [8] and a system for data management in distributed environments (industrial study covered by a NDA).

In previous work [15], 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.
– 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 show how TestEra users can completely break symmetries, such that each solution of a boolean formula corresponds to a non-isomorphic test input.

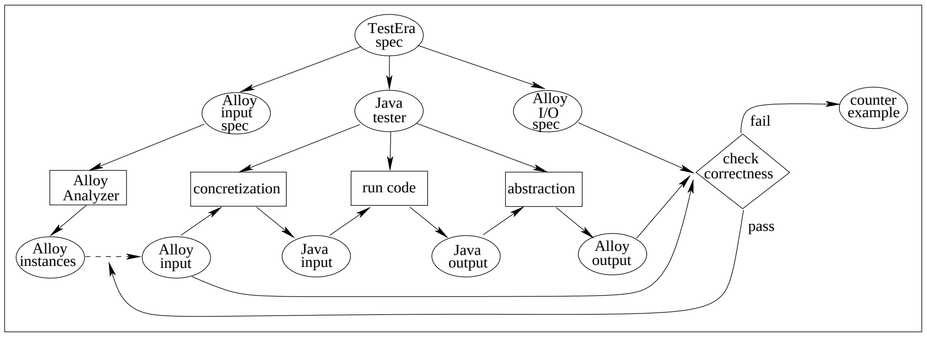

Fig. 1. Basic TestEra framework

2 TestEra

Figure 1 illustrates the main components of the TestEra framework. Given a method precondition 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 this instance satisfies the postcondition. If it does not, TestEra reports a concrete counterexample, i.e., an input/output pair that violates the correctness specification. TestEra can graphically display the counterexample, e.g., as a heap snapshot, using the visualization facility of the analyzer.

2.1 Symmetry Breaking

The analyzer adapts symmetry-breaking predicates [7] 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 [18]. There is a trade-off, however: the more clauses that the analyzer generates, the more symmetries it breaks, but the boolean formula also becomes larger, and it can become too large so that solving takes more time, although 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.

The analyzer has a 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 been used for faster analysis. We show in Section 3.1 how TestEra and Alloy users can also use total orders to constrain specifications such that the analyzer generates exactly one instance from each isomorphism class. Conceptually, the idea is to conjoin Alloy specification with additional constraints such that the analyzer generates, from each isomorphism class, only the instance that is the *smallest* with respect to the total orders on the sets.

3 Example

We next show a simple example that illustrates the use of TestEra. Consider the following Java code that declares a binary tree and its `removeRoot` method:

```
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. For these classes, 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*, i.e., 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 class invariant) [14] for `BinaryTree`. A predicate that checks the invariant is typically called `repOk` (or `checkRep`) [14]. For `BinaryTree`, this predicate 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 has two formulas. They are within (outer) curly braces, and thus implicitly conjoined. The first formula, which has three subformulas, 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 entire `root.*(left + right)` denotes the set of all nodes reachable from `root`. The quantifier `all` denotes for universal quantification: the formula `all n: S { F }` holds iff the formula `F` holds for 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 `t` to have the correct `size`; '`#`' denotes set cardinality.

We add the function `repOk` to the above Alloy specification to obtain the entire specification for `removeRoot`'s inputs. The Alloy command `run repOk for` N `but 1 BinaryTree` instructs the Analyzer to find an instance for this specification, i.e., the valuation of signatures (sets) and relations that make 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, but we further instruct the Alloy Analyzer to generate all instances, effectively generating all trees with up to the given number of nodes.

In the first phase, TestEra uses the analyzer to generate all (non-isomorphic) instances. In the second phase, TestEra operates on each instance in turn: 1) translates it to appropriate Java test input by creating objects (of classes `BinaryTree` and `Node`) that correspond to the atoms in the instance and setting the object fields to correspond to the relations in the instance; 2) executes `removeRoot` on the obtained test input; 3) translates the resulting Java objects back into an Alloy instance by translating the values of object fields into relations; and 4) evaluates the postcondition on this translated output and the original input instance. (In general, a postcondition can refer to both input and output, but our simplified example considers only output.) If the code contains a bug that can be observed for one of these trees, e.g., the code does not decrement the number of nodes after deleting the root, TestEra readily exposes the bug.

In the sequel, we focus on test input generation. To compare different ways for generation, we consider test inputs of size exactly N. To this end, we add to the specification the following:

```
fact Connected { BinaryTree.root.*(left+right) = Node }
```

A *fact* is a formula that puts more constraints on the instances: `run`ning a function finds instances that satisfy the function body conjoined with all the facts in the specification. `Connected` states that the set of nodes reachable from `BinaryTree` is the same as the universe of `Node`s, whose cardinality is exactly N.

For illustration, consider $N = 5$. There are 14 non-isomorphic trees with five nodes [19]. 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 [18], we can tune how many symmetries to break. With the default value of symmetry breaking, the analyzer generates 17 trees with five nodes. If we increase symmetry breaking, the analyzer generates exactly 14 trees.

3.1 Complete Symmetry-Breaking using Total Order

We next show how to use the special support that the analyzer has for total orders to completely break all symmetries in our example. 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) {
      n.left.*(left + right) in OrdPrevs(n) // library function that instantiates Ord[Node]
      n.right.*(left + right) in OrdNexts(n) } }
```

The function `OrdPrevs`, respectively `OrdNexts`, returns the set of all elements that are smaller, respectively larger, than the given element. The fact requires that all trees in the instance (the example instances have only one tree) have nodes in an *in-order* [6]: the nodes in the left, respectively right, subtree of the node `n` are smaller, respectively 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 so that each instance can have nodes in only one order, effectively eliminating isomorphic instances. Indeed, the analyzer now generates exactly 14 non-isomorphic trees, as expected. 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.

4 Results

We next present some performance results for solution enumeration obtained with mChaff [16]. Table 1 presents the results 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`

			manual symmetry breaking				automatic symmetry breaking			
benchmark	size	#prim	#vars	#clauses	#sols	time	#vars	#clauses	#sols	time
`BinaryTree`	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`	7	191	2834	9834	877	1.04	3559	11021	26551	35.38
	8	242	3837	14007	4140	4.76	4432	14939	356276	736.30
	9	299	5852	24411	21147	36.52	6629	25630	/	mem.
`TreeMap`	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`	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`	6	72	704	1611	13139	5.10				
	7	90	884	2128	117562	62.62				
	8	110	1084	2735	1005075	1171.64				

Table 1. Performance. All times are in seconds (of total elapsed wall-clock time); the experiments were performed on a 1.8 GHz Pentium 4 processor. For sizes larger than presented, enumeration of solutions for automatically constructed symmetry-breaking predicates takes longer than 1 hour.

field and a `header` node as a sentinel node [6]. (Linked lists also provide methods that allow them to be used as stacks and queues.) `TreeMap` implements the `Map` interface using red-black trees [6]. Each node has a `key` and a `value`. (Setting all `value` fields to `null` corresponds to the set implementation in `java.util.-TreeSet`.) `HashSet` implements the `Set` interface, backed by a hash table [6]. This implementation builds collision lists for buckets with the same hash code. `HeapArray` is an array-based implementation of heap (priority queue) data structure [6]. (`HeapArray`s are similar to array-based stacks and queues, as well as `java.util.Vector`s, so the results presented here are similar to those results.)

We show results for several size values for each benchmark. All scope parameters are set exactly to the given size; e.g., all lists have exactly the given 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: 1) one with symmetry-breaking predicates generated automatically (using the default values of the Alloy Analyzer) and 2) one with symmetry-breaking predicates added manually to Alloy specifications (as described in Section 3.1). 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.

For `BinaryTree`, `LinkedList`, `TreeMap`, and `HashSet`, the numbers of non-isomorphic structures appear in the Sloane's On-Line Encyclopedia of Integer Sequences [19]. For all sizes, formulas with manually added symmetry-breaking predicates have as many solutions as the actual number of structures, which shows that these predicates eliminate all symmetries. (For this comparison, we generated inputs with exactly the given size; for software testing in practice, we generate all inputs *up to* the given size.) For `HeapArray`, no symmetry-breaking is required: two array-based heaps are isomorphic iff they are identical.

In all cases with symmetry breaking, formulas with automatic symmetry breaking 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. However, note that it is not always the case: 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. The Alloy Analyzer allows users to tune symmetry breaking; we have experimented with different parameter values and the 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 [16] 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 [3] for methods in the respective Java classes.

We also conducted some 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 there's a BDD for a formula. 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 [20] BDD package. We constructed BDDs bottom-up, using automatic variable reordering via sifting [5], from the boolean DAGs from which the CNFs were produced. For all benchmarks, the BDD approach scaled poorly; for nontrivial sizes (over five), the BDD construction led to unmanageably large BDDs (over a million nodes). We plan to further evaluate BDD-based approaches.

5 Conclusions

We have presented a novel application of SAT solvers in software testing. Our application requires a solver that can enumerate all satisfying assignments; each assignment provides a (non-isomorphic) input for the program. The experimental results indicate that it is feasible to use a SAT solver to systematically generate structurally complex inputs that would be hard to generate manually. We hope that our work provides motivation for exploring efficient solution enumeration in modern SAT solvers.

References

1. SAT competitions. `http://www.satlive.org/SATCompetition/`.
2. 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.

3. B. Beizer. *Software Testing Techniques.* International Thomson Computer Press, 1990.
4. A. Biere. Limmat satisfiability solver. `http://www.inf.ethz.ch/personal/biere/projects/limmat/`.
5. 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.
6. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1990.
7. 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.
8. 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.
9. E. Goldberg and Y. Novikov. BerkMin: a fast and robust SAT-solver. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, Mar. 2002.
10. D. Jackson. Micromodels of software: Modelling and analysis with Alloy, 2001. `http://sdg.lcs.mit.edu/alloy/book.pdf`.
11. 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.
12. H. Kautz and B. Selman. Planning as satisfiability. In *Proc. European Conference on Artificial Intelligence (ECAI)*, Vienna, Austria, Aug. 1992.
13. 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.
14. B. Liskov. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.* Addison-Wesley, 2000.
15. 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.
16. 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.
17. A. Nadel. Jerusat. `http://www.geocities.com/alikn78/`.
18. I. Shlyakhter. Generating effective symmetry-breaking predicates for search problems. In *Proc. Workshop on Theory and Applications of Satisfiability Testing*, June 2001.
19. 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`.
20. F. Somenzi. CUDD: CU decision diagram package. `http://vlsi.colorado.edu/~fabio/CUDD/`, 2001.
21. Sun Microsystems. *Java 2 Platform, Standard Edition, v1.3.1 API Specification.* `http://java.sun.com/j2se/1.3/docs/api/`.
22. 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, Apr. 2003.