Field-Exhaustive Testing

Pablo Ponzio[*‡] Nazareno Aguirre[*‡] Marcelo F. Frias[†‡] Willem Visser[§]

[*]Departamento de Computación, Universidad Nacional de Río Cuarto, Argentina
{pponzio, naguirre}@dc.exa.unrc.edu.ar
[†]Departamento de Ingeniería de Software, Instituto Tecnológico de Buenos Aires, Argentina
mfrias@itba.edu.ar
[‡]Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina
[§]Department of Computer Science, University of Stellenbosch, South Africa
wvisser@cs.sun.ac.za

ABSTRACT

We present a testing approach for object oriented programs, which encompasses a testing criterion and an automated test generation technique. The criterion, that we call *field-exhaustive testing*, requires a user-provided limit n on the size of data domains, and is based on the idea of considering enough inputs so as to exhaustively cover the *extension* of class fields, within the limit n. Intuitively, the extension of a field f is the binary relation established between objects and their corresponding values for field f, in valid instances. Thus, a suite S is field-exhaustive if whenever a field f relates an object o with a value v (i.e., $o.f = v$) within a valid instance I of size bounded by n, then S contains at least one input I' covering such relationship, i.e., o must also be part of I', and $o.f = v$ must hold in I'. Our test generation technique uses incremental SAT solving to produce *small* field-exhaustive suites: field-exhaustiveness can be achieved with a suite containing at most $\#F \times n^2$ inputs, where $\#F$ is the number of fields in the class under test.

We perform an experimental evaluation on two different testing domains drawn from the literature: implementations of data structures, and of a refactoring engine. The experiments show that field-exhaustive suites can be computed efficiently, and retain similar levels of code coverage and mutation killing as significantly larger bounded exhaustive and random suites, thus consuming a fraction of the cost of test execution compared to these automated testing approaches.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

Software Testing, Automated Test Generation, SAT Solving

1. INTRODUCTION

Testing is a powerful and widely used technique for software quality assurance [20, 2]. It essentially consists of assessing the quality of a piece of code by executing it under a number of particular inputs, which are a defining part of *test cases*. For a set of test cases to be adequate, these generally need to exercise the software under varying situations. This of course increases the chances to detect bugs, since the software being assessed is more thoroughly examined. Software testing criteria define concrete mechanisms to decide when a set of tests, or suite, is adequate, or sufficiently thorough in the examination of the software under evaluation [36, 2].

Most testing criteria have been defined under the assumption that tests, including test inputs, are manually written. Thus, even for sophisticated criteria, such as some white-box criteria like condition coverage and MC/DC (Modified Condition/Decision Coverage), suites with very good levels of coverage can be achieved while keeping the size of the suite small, which of course requires in many cases significant efforts from the testing engineers. With the advent of automated test generation techniques, testing approaches that are impractical with manual testing though feasible through automated testing, are emerging. Relevant cases of this are *bounded exhaustive testing* [32], a black box criterion which proposes to test programs on *all* valid inputs bounded by a user provided bound, and *random testing*, a well known testing approach proposing to test programs on randomly generated inputs [7, 27]. These approaches have proved to be effective in various testing domains [27, 5, 9, 30]. However, bounded exhaustive test suites are inherently combinatorial in size, and randomly generated suites typically have to be composed of large test sets to achieve good coverage levels. This fact makes the execution of such suites prohibitively expensive in many situations, especially when they need to be repeateadly used, e.g., in regression testing contexts.

In this paper we present a testing criterion for object oriented programs, that we call *field-exhaustive testing*, which requires a user-provided limit n on the size of data domains. This criterion is based on the idea of considering enough inputs so as to exhaustively cover the *extension* of class fields, within the provided limit n. Intuitively, the extension of a field f is the binary relation established between objects and their corresponding values for field f, in valid instances bounded by n. Thus, a suite S is field-exhaustive if whenever a field f relates an object o with a value v (i.e., $o.f = v$) within a valid instance I of size bounded by n, then S contains at least one input I' covering such relationship, i.e., o must also be part of I', and $o.f = v$ must hold in I'. We also present a technique that automatically generates field-exhaustive suites, using incremental SAT solving [23]. We perform an experimental evaluation on two different testing domains drawn from the literature, the implementations of data structures and of a refactoring engine, that show that field exhaustive suites can be efficiently produced, and if symmetry breaking is imposed, our automatically generated field-exhaustive suites are *small*, while retaining similar levels of code coverage and mutation killing as significantly larger bounded exhaustive and random suites. Moreover, we prove that field-exhaustive suites can be achieved with at most $\#F \times n^2$ inputs, where $\#F$ is the number of fields in the class under test, and n the user-provided bound on the size of data domains. This implies that, in many cases, field-exhaustive suites can be executed at a fraction of the cost of test execution of bounded exhaustive or random suites, to achieve comparable coverage or mutation score.

2. BACKGROUND

Given a program and a test suite for it, a testing criterion enables one to assess how well the suite exercises the program. There exist two broad testing criteria categories, *white-box* and *black-box* [36, 26]. White-box criteria take into account the structure of the program under test, while black-box criteria deal with the program as a black box and may only examine its *specification*. An example of a white-box criterion is *statement coverage*, which requires each program statement to be executed by some test. An example of a black-box criterion is *equivalence partitioning*, which divides the state space of inputs of the software under test into equivalence classes, and requires each equivalence class to be covered by some test in the suite.

Most prevailing testing criteria have been devised under the assumption that test suites are produced *manually* and thus can be satisfied with a (relatively) small number of test cases. Advances in automated testing are enabling novel criteria whose applicability strongly depends on testing automation both at generation time and at execution time. An example of such a testing technique is *random testing*, a well-known testing approach proposing to test programs on randomly generated inputs [7, 27]. While random testing originally applied to programs with inputs of basic datatypes, and simply resorted to random number generators for input generation, more recently the approach has been successfully extended to complex inputs, e.g, by randomly producing sequences of methods that construct the inputs (and exploiting feedback of previously generated cases to avoid redundancies) [27], or employing user-provided input generators to randomly create test cases [7]. Random testing can generate very good test suites (e.g., suites with high coverage or good mutation-killing scores) very efficiently, but at the expense of producing large suites with large tests (composed of large sequences of methods); usually, constraints have to be imposed during generation, e.g., maximum test length, maximum generation time or maximum size of test suite, to limit the size of the produced suites and therefore the time for testing, and a compromise has to be established between the test suite sizes and quality of the suites.

Another important testing technique, that can be effectively applied thanks to automation and is strongly related to the technique we present in this paper, is *bounded exhaustive testing* [32]. Bounded exhaustive testing produces, for a given program and a user-provided bound n on the size of inputs (called the *scope*), all valid inputs whose size is bounded by n, and then tests the program using the produced test suite. For instance, bounded exhaustively testing a routine manipulating binary search trees defined by the classes in Fig. 1, for a scope of up to 4 nodes and key and size values from 0 to 4, consists of checking the routine on *all* valid search trees that can be built using at most 4 nodes, with values of keys and size within 0–4. This testing technique has proved useful in some testing contexts, in particular, for testing code that manipulates complex data structures, or programs that deal with source code, such as compilers and refactoring engines. The rationale behind the approach is based on the *small scope hypothesis* [18]: many bugs in programs manipulating complex data can be reproduced using small instances of such data. Thus, by testing the program on all possible inputs bounded in size by some relatively small scope one would be able to exhibit most bugs.

The scope establishes a maximum number of objects in the heap (e.g., 1 binary search tree object `T0`, 4 node objects `N0`, `N1`, `N2`, `N3`), and a domain for each field of each of these objects (e.g., for a node `N0`, according to its type, field `N0.left` can be assigned either `null` or one element from {`N0`, `N1`, `N2`, `N3`}, the 4 nodes available in the heap). A specification of the *valid* structures, e.g., a representation invariant for search trees such as the one in Fig. 2, distinguishes between well-formed bounded structures and ill-formed ones, that can be disregarded for testing. A crucial mechanism that tools for bounded exhaustive testing base their efficiency on is the ability to remove, disregard and avoid visiting a particular kind of redundant instances during generation. These are the so called *isomorphic* instances, i.e., instances which only differ in the object identifiers assigned to their composing nodes. Consider for instance the two binary search trees in Fig. 3. These two trees are exactly the same, hold the same information, but their nodes identifiers differ. In fact, one is a permutation of the other. Intuitively, these identifiers correspond to memory locations, and therefore these trees differ in where their nodes are located in the heap. Such difference is irrelevant in most applications, and thus one may want to avoid generating both cases, since one represents the other. *Symmetry-breaking* approaches allow us to deal with this issue, by enforcing a canonical order in nodes and thus allowing for only one (canonical) representative for all isomorphic instances. For example, Korat's symmetry-breaking approach is based on a rule enforcing that while `repOK()` (the operational implementation of the representation invariant) is generating a candidate structure during the search for valid structures, when attribute f is defined for node N_i, its value $N_i.f$ must be either *null*, or must be within the set $\{N_0, N_1, \ldots, N_m\}$, where m is the smallest identifier not already used. Notice that for the `repOK()` routine in Fig. 2, the first tree in Fig. 3 is accepted, while the second is not. A similar symmetry breaking approach, employed by some tools based on declarative (logical) languages for invariant specification such as TACO [15, 16] and FAJITA [1], is based on the addition of further (logical) restrictions, imposing that, when a structure is traversed in breadth-first order, node identifiers are sequentially labeled.

3. MOTIVATING EXAMPLE

Let us consider again binary search trees, already em-

```
public class BinaryTree {
    private Node root;
    private int size;
    ...
}

public class Node {
    private int key;
    private Node left;
    private Node right;
    ...
    // setters and getters of the
    // above fields
    ...
}
```

Figure 1: Java classes defining binary search trees.

ployed for illustration in the previous section. The representation invariant for this structure states that the linked structure formed from the `root` node following the `left` and `right` fields must be acyclic, that each node in it except the root (which must have no parent) must have exactly one parent, and that the tree must be ordered. This invariant can be captured operationally, as in the `repOK()` Java predicate shown in Fig. 2, or logically (declaratively), as shown in Fig. 4, in this case expressed using Alloy's relational logic [18] (other logical formalisms, such as JML [6], can be also employed to declaratively capture the invariant). A user-provided *scope* defines a finite set of possible structures, but only a portion of these are *valid*, i.e., satisfy the representation invariant. For instance, for up to 4 nodes, sizes and keys within ranges $[0, 4]$ and $[0, 3]$, respectively, a total of 2,500,000,000 structures can be built, but only 977 satisfy the invariant. Notice that the stronger the representation invariant, the smaller the set of valid structures. Symmetry-breaking also has a significant impact on the number of valid structures: out of the 997 valid structures for the above scope, only 51 are non-isomorphic. Moreover, symmetry breaking also impacts test input generation's efficiency; for instance, the bounded exhaustive testing tool Korat [5] takes a third of the time to produce the 51 non-isomorphic structures, compared to the time it takes to produce the 997 structures, when symmetry breaking is disabled.

Despite the improvements in suite's size and generation time that symmetry-breaking provides, bounded exhaustive suites inherently grow combinatorially as the scope increases, even for strong representation invariants. This fact makes it costly to compute bounded exhaustive suites for larger scopes, and the corresponding large testing times makes it often prohibitive to use bounded exhaustive testing, especially when these suites are incorporated into processes that require their repetitive use along the development process, as in regression testing. A similar case can be made for other automated generation techniques, random testing in particular. For our running example, for instance, Randoop [27] can achieve a similar mutation killing score and branch coverage as the bounded exhaustive suite of scope 4 only after producing 5,000 tests (see section on validation). Again, such test suite sizes make the use of the produced test suites impractical in many situations.

Our proposal is related to bounded exhaustive testing in the sense that it requires a user-provided scope. But as opposed to bounded exhaustive testing, our approach is based on the observation that while the scope determines a set of

```
public boolean repOK() {
   if (root == null) return true;
   if (!isAcyclic()) return false;
   if (!isOrdered(root, -1, -1)) return false;
   return true;
}

private boolean isAcyclic() {
   Set visited = new HashSet();
   visited.add(root);
   LinkedList workList = new LinkedList();
   workList.add(root);
   while (!workList.isEmpty()) {
      Node current = (Node) workList.removeFirst();
      if (current.getLeft() != null) {
         if (!visited.add(current.getLeft())) return false;
         workList.add(current.getLeft());
      }
      if (current.getRight() != null) {
         if (!visited.add(current.getRight())) return false;
         workList.add(current.getRight());
      }
   }
   return true;
}

private boolean isOrdered(Node n, int min, int max) {
   if (n.info == -1) return false;
   if ((min != -1 && n.getKey() <= (min)) ||
       (max != -1 && n.getKey() >= (max))) return false;
   if (n.left != null)
      if (!isOrdered(n.getLeft(), min, n.getKey())) return false;
   if (n.right != null)
      if (!isOrdered(n.getRight(), n.getKey(), max)) return false;
   return true;
}
```

Figure 2: Operational version of the representation invariant for binary search trees.

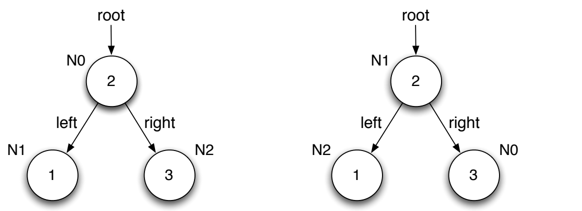

Figure 3: Two isomorphic binary search trees.

possible target values for each field of each heap object, this target domain may be restricted to a smaller set of *feasible values* both by the representation invariant and symmetry-breaking predicates, and a smaller set of valid instances may be needed to *cover* these feasible values. As an example, let us consider again the previously mentioned scope, and let us concentrate for the moment on the values of field `left` for node `N0`. This field's type and the scope define $\{$`N0`, `N1`, `N2`, `N3`, `null`$\}$ as the range set of the field, but we know that there are only two possible values for `N0.left`, namely, `null` and `N1`. Any other value is forbidden either by the representation invariant, or by symmetry-breaking. Let us call then $\{$`null`, `N1`$\}$ the *extension* of `N0.left`. Following a similar reasoning, the extension of field `N0.right` is $\{$`N1`, `N2`, `null`$\}$, and so on. Field-exhaustive testing, as opposed to bounded exhaustive testing, requires considering enough inputs so that the extension of every field, within a given scope, is *covered*. For instance, for the above scope,

```
( all n : Node | n in thiz.root.*(left + right) implies
  (
    no (( (n.left).*(left+right) & (n.right).*(left+right) ) - null )          -- no node can be reached along two different paths
    and
    (n !in n.^(left + right))                                                    -- the structure is acyclic
    and
    (all m : Node | m in n.left.*(left + right) implies gt[n.key, m.key])       -- left-sorted
    and
    (all m : Node | m in n.right.*(left + right) implies gt[m.key, n.key])      -- right-sorted
  )
) and
  thiz.size = #thiz.root.*(left + right)                                         -- field size is well-defined
```

Figure 4: Logical invariant for binary search trees.

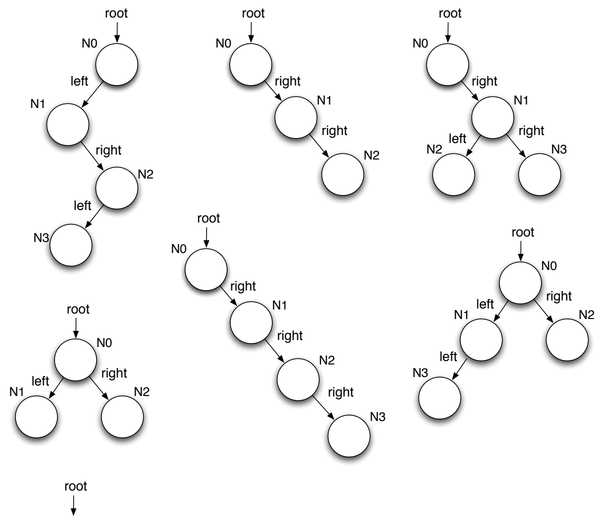

Figure 5: Field-exhaustive suite for binary search trees, with scope 4 (keys ignored).

17 inputs suffice to achieve field exhaustiveness, whose 7 shapes are shown in Fig. 5 (we ignore key values for presentation purposes), as opposed to the 51 non-isomorphic instances that constitute the bounded exhaustive suite for the same scope. This difference is more pronounced as the scope increases; for instance, for scope 10 (10 nodes, size within 0-10, keys within 0-9), field exhaustiveness can be achieved with 86 instances, while 223,191 non-isomorphic instances constitute the bounded exhaustive suite for the same scope. Randoop still needs to produce more than 5,000 tests to achieve the same mutation killing and coverage as the field-exhaustive suite composed of 86 structures.

In the next section we formally introduce field-exhaustive testing, while in Section 5 we concentrate on a technique to automatically build field-exhaustive suites.

4. FIELD-EXHAUSTIVE TESTING

In this section we formally define field-exhaustive testing, as a testing criterion. Let us assume, without loss of generality, that the testing subject is a collection $C_1, \ldots, C_n$ of classes, where C_1 is the main class. Each class C_i defines a set $f_1^{C_i}, \ldots f_{l(C_i)}^{C_i}$ of fields, whose types are among $C_1, \ldots, C_n$ and primitive datatypes. The *scope* characterizes finite sets of objects for each of the involved classes, and finite ranges for basic datatypes. We often simplify the scope as a number k, typically referring to the number of "node" objects, and define ranges for primitive types as a function of k. The set of possible structures or instances is composed of all possible assignments of values within the scope, for fields of the scope's objects, respecting the fields' types. We also assume that a predicate $inv(o)$, defined on instances of the main class C_1, is provided. This specification defines the *valid* instances of C_1, and for a given scope k, identifies a subset of the possible structures, the *valid* structures. Given a valid instance c of C_1, we say that c *involves* an object o, if o is reachable from c in the memory heap.

Definition 1. Let C_i and C_j be classes involved in a testing subject, k a scope definition, and c_i, c_j objects of classes C_i, C_j, respectively, within scope k. Given a field f of type C_j in class C_i, we say that $\langle c_i, c_j \rangle$ is *feasible* for f within scope k iff there exists a valid structure o of C_1 (i.e., $inv(o)$ holds) that involves c_i, and such that $c_i.f = c_j$. We refer to the set of all feasible tuples for field f within scope k as the *extension* of f (for scope k).

As an example clarifying the above definition, let us consider again binary search trees, with a scope of 1 binary search tree object, 4 nodes, size ranging within 0-4 and keys ranging within 0-3. Assuming that inv is in this case the representation invariant of binary search trees supplemented with symmetry breaking, as previously described, $\langle N0, N1 \rangle$ is feasible for `left` within the scope (the first tree in Fig. 5 witnesses this feasibility), and

$$\{\langle \texttt{N0}, \texttt{null} \rangle, \langle \texttt{N0}, \texttt{N1} \rangle, \langle \texttt{N1}, \texttt{null} \rangle, \langle \texttt{N1}, \texttt{N2} \rangle, \langle \texttt{N1}, \texttt{N3} \rangle,$$
$$\langle \texttt{N2}, \texttt{null} \rangle, \langle \texttt{N2}, \texttt{N3} \rangle, \langle \texttt{N3}, \texttt{null} \rangle\}$$

is the extension of field `left`, for the given scope. Notice how the breadth-first ordering on node identifiers imposed by symmetry-breaking prevents pair $\langle N0, N2 \rangle$ from being part of the extension of field `left`.

Definition 2. Given a testing subject $C_1, \ldots, C_n$, a scope k, and a set $S = \{o_1, \ldots, o_x\}$ of objects of class C_1 within scope k, we say that S achieves field-exhaustive coverage for scope k iff for every feasible tuple $\langle c_i, c_j \rangle$ of every field f, there exists an object o in S that involves object c_i and such that $c_i.f = c_j$. That is, a set S of objects is said to achieve field-exhaustive coverage for a given scope iff the extension of each field if covered by S.

Two examples of field-exhaustive suites for the previously mentioned scope are the suite composed of objects in Fig. 5 (with respect to `left` and `right`, keys are ignored), and the bounded exhaustive suite for the same scope (it contains *all* valid instances, therefore it satisfies field-exhaustiveness). Although we do not do so in this paper, quantitative variants of this criterion's satisfaction are straightforward to define. For instance, we may say that a suite S satisfies field exhaustiveness by a $N\%$ for a given scope k if $N\%$ of the feasible tuples for scope k are covered by S.

It is worth noticing that symmetry-breaking plays an important role on the size of suites that achieve field exhaustiveness. For instance, if symmetry-breaking is disregarded, we then need at least 27 test inputs to achieve field exhaustiveness for scope 4 on binary search trees (as opposed to the 17 that suffice with symmetry breaking).

5. AUTOMATED GENERATION OF FIELD-EXHAUSTIVE SUITES

We now turn our attention to automatically producing field-exhaustive suites. While any mechanism capable of building bounded exhaustive suites achieves by definition field-exhaustiveness, our motivation is to avoid the large size of such suites, and the associated limitations such sizes bring into the testing process. Therefore, we aim at producing *small* field-exhaustive suites. In this section we present a technique that in practice produces field-exhaustive test suites that are significantly smaller than bounded exhaustive suites for the same scope, and randomly generated suites of similar quality (coverage, mutation score). This technique is based on the use of incremental SAT solving.

Our procedure to compute field-exhaustive suites requires *inv*, the specification of valid instances, to be expressed in a language amenable to satisfiability analysis, in our case via SAT solving. While we use invariants expressed in Alloy's relational logic, other languages may be employed; for instance, invariants specified in JML [6] can be automatically translated into SAT as described in [15], and operational invariants such as those used in Korat [5] can also be automatically translated into propositional formulas (for a given scope), following translations such as those embedded in tools like CBMC [22] and DynAlloy [14]. Let us then assume that *inv* is expressed as (or can be translated into) a propositional formula. Moreover, let us further assume that *inv* subsumes a symmetry-breaking predicate forcing canonical orderings of heap nodes in structures (this is without any loss of generality, since as described in [15, 16], such symmetry-breaking predicates can be automatically built).

Intuitively, the propositional encoding of heap structures captures values for object fields through propositional variables. That is, given a field $f : C \rightarrow C'$, an object a of class C having value b of class C' in $a.f$ in a heap structure is characterized through the validity of a variable $var_f(\langle a, b\rangle)$ in a satisfying valuation of the propositional encoding of heap structures. Then, in order to produce new test inputs that contribute to field-exhaustively covering f we may feed the SAT solver with a clause:

$$\bigvee_{a:C, b:C'} (var_f(\langle a, b\rangle)). \qquad (1)$$

Notice then that the satisfiability of (1) implies that there exist $a : C$ and $b : C'$ such that variable $var_f(\langle a, b\rangle)$ evaluates to true in a satisfying valuation, or in other words, such as $a.f = b$ in a corresponding heap structure. To guarantee that such structure indeed contributes to field-exhaustively covering f, we need to ensure: *(i)* that some satisfied $var_f(\langle a, b\rangle)$ corresponds to a pair $\langle a, b\rangle$ not yet covered, and *(ii)* that $var_f(\langle a, b\rangle)$ indeed characterizes a feasible value for field f, in a *valid* heap structure of class C_1 (the main class of the testing subject). The first property can be guaranteed by keeping track of those pairs that have already been covered, and restricting the above disjunction (1) only to pairs not previously covered. The second requirement is achieved by enforcing the validity of $var_f(\langle a, b\rangle)$ only in *valid* heap structures that involve a and b, captured by formula:

$$var_f(\langle a, b\rangle) \iff \bigvee_{o:C_1} (inv(o) \wedge a \text{ reachable\_from } o \ \wedge \langle a, b\rangle \in f) . \qquad (2)$$

Notice that since instances are bounded by the scope k, reachability is expressible in propositional logic (see for instance the translation of transitive and reflexive-transitive closure operators into propositional logic employed by the Alloy Analyzer tool [18]). Formula (2) states that variable $var_f(\langle a, b\rangle)$ encodes the *feasibility* of $\langle a, b\rangle$ as part of the extension of f in valid canonical structures. Thus, by considering Formula (2), in case the satisfiability verdict for a variable $var_f(\langle a, b\rangle)$ is *true*, the returned valuation corresponds to a *valid* instance, containing that tuple. Furthermore, since the valuation encodes a valid instance, we can retrieve for each field f the pairs of object identifiers in f in that particular instance. For example, if we require $var_{\mathsf{left}}(\langle N0, N1\rangle)$ to hold, the first binary search tree in Fig. 5 fulfills the requirement. As a side-effect of selecting this instance, we may conclude that, besides pair $\langle N0, N1\rangle$, also pair $\langle N2, N3\rangle$ belongs to the extension of field left.

We use the above introduced variables to automatically produce field-exhaustive suites by incrementally calculating field extensions within a provided scope, and collecting the instances obtained in this process. Suppose that $f_1, \ldots, f_m$ is the sequence of all fields in the testing subject. Our algorithm to produce field-exhaustive suites will maintain partial extensions for fields $f_1, \ldots, f_m$ in a vector whose i-th position, for $1 \leq i \leq m$, stores a partial extension for f_i. In this way, our algorithm works on a vector `curr_sets` consisting of m sets (the current partial field extensions for $f_1, \ldots, f_m$). The algorithm makes use of an incremental SAT solver, represented by a module *solver*, with the following routines:

- *load*: receives as arguments a formula ϕ (capturing the invariant and other constraints), and a scopes definition for the domains involved in the specification. It generates a propositional formula (in conjunctive normal form) representing the satisfiability of formula ϕ within the provided scopes, and loads it into the solver.
- *addClause*: (incrementally) adds a clause to the current formula in the solver for future solving invocations.
- *solve*: calls the SAT-solving procedure, which decides whether the formula currently loaded in the solver is satisfiable or not.
- *getInstance*: if the formula is satisfiable, it retrieves a satisfying instance from the valuation produced by the SAT-solver.

Finally, vectors of partial field extensions support a routine *extend*, which receives as a parameter a heap instance,

Algorithm 1 Field exhaustive suite computation algorithm

```
1: function COMPUTE-SUITE($inv$, $scopes$)
2:     $solver$ = new $Solver()$
3:     $solver.load(inv, scopes)$
4:     $curr\_sets = [\emptyset, \ldots, \emptyset]$
5:     $suite = \emptyset$
6:     while $True$ do
7:         $solver.addClause\left(\bigvee_{j \in 1,..,m,\ \langle a,b\rangle \in \mathrm{dom}(f_j)\setminus curr\_sets[j]} var_{f_j}(\langle a,b\rangle)\right)$
8:         if $solver.solve()$ then
9:             $curr\_sets.extend(solver.getInstance())$
10:            $suite.add(solver.getInstance())$
11:        else
12:            break
13:        end if
14:    end while
15:    return $suite$
16: end function
```

and augments the current field extensions with the field tuples contributed by the provided instance (that are not yet present in the current field extensions).

Our algorithm initializes `curr_sets` as a vector of empty sets, which are iteratively augmented according to heap instances computed via calls to the SAT-solver, collecting the instances obtained in the process. The execution terminates when `curr_sets` cannot be augmented any further, in which case, as we will prove below, the collected instances form a field-exhaustive suite. Pseudocode for the algorithm is shown in Algorithm 1. It takes as inputs the invariant of the testing subject (assumed to include symmetry-breaking predicates), and the scopes for which the field-exhaustive suite is to be computed. The algorithm begins by creating a solver and loading it with the invariant, translated, for the given scopes, as a propositional formula (lines 2–3). It then initializes the (partial) field extensions with empty sets for each of the heap fields (line 4), and the suite as the empty set (line 5). The while-loop in lines 6–14 augments the field extensions while collecting instances, until they cannot be further increased. At this point, the vector holds full field extensions, the suite achieves field-exhaustiveness and it is returned by the algorithm as its result (line 15).

A crucial step in our algorithm is performed at line 7, where the formula currently loaded into the SAT solver is extended, exploiting incremental SAT solving [23], with a progress-ensuring constraint on heap field extensions. Variable $var_{f_j}(\langle a, b\rangle)$ in this line (see Formula (2)), denotes the propositional variable that, after translation to a SAT problem, models the membership of pair $\langle a, b\rangle$ to field f_i. The purpose of this constraint is then to ensure that instances returned by $solver.solve()$ contain at least one pair that does not belong to the partial field extensions already held in `curr_sets`. Intuitively, by adding the clause in line 7, the call to $solver.solve()$ in line 8 can be interpreted as *"find a valid instance of the structure that can be used to augment at least one of the current field extensions in `curr_sets`"*. If such an instance exists, `curr_sets` is augmented (line 9).

The extending clause at line 7 is a crucial element in the design of our algorithm to compute field-exhaustive suites. More precisely, by encoding the new constraint as a clause, we enable the possibility of using *incremental SAT solving* [23], a feature supported by many modern SAT solvers, including MiniSat [11], the one we use in our experimental evaluation for field-exhaustive suite computation. Essentially, incremental SAT solvers allow one to append further constraints written in conjunctive normal form, after each satisfying valuation is found. These constraints are conjuncted to the main formula, and used in computing the "next" satisfying instance incrementally from a currently computed one, without having to restart the solving process, and therefore exploiting already learnt clauses (a critical part of SAT solving based on conflict analysis and clause recording). Notice that, if the new constraints are not in the right format, the whole resulting formula has to be translated to conjunctive normal form and the SAT process restarted from scratch. The formula added in line 7 is indeed a clause (and thus it is written in conjunctive normal form), allowing us to employ incremental SAT solving.

Theorem 1 proves that our algorithm terminates and returns a field-exhaustive test suite.

THEOREM 1. *Algorithm 1 terminates and returns a field exhaustive suite.*

PROOF. Termination easily follows from the following two facts: *(i)* for given partial field extensions of the testing subject whose sizes are limited by a scope, the number of pairs that can be added to a field's extension is finite; and *(ii)* each while-loop iteration either adds at least an extra pair to the partial field extensions, or otherwise returns *unsat*, in which case the loop terminates.

To prove that the algorithm yields a field-exhaustive suite, we must prove that the computed suite covers the extension of all the fields. Let us assume this is not the case and arrive to a contradiction. Assume Algorithm 1 terminates and there is a field f and a pair $\langle a, b\rangle$ in the extension of f that was not covered. Let us focus on the last iteration of the loop prior to termination of the algorithm. Since pair $\langle a, b\rangle$ was not yet covered, propositional variable $var_f(\langle a, b\rangle)$ must be one of the disjuncts in the clause fed to the solver in line 7. Since this is the last loop iteration, the solver call in line 8 must return an UNSAT verdict, which contradicts the fact $\langle a, b\rangle$ is in the extension of field f. The contradiction arises from assuming field-exhaustiveness was not achieved. □

While our algorithm produces field-exhaustive suites that are significantly smaller than bounded exhaustive suites (see the evaluation in Section 6), this algorithm does not necessarily compute *minimal* field exhaustive suites. As an example, consider again the binary search tree case, and the suite for scope 4 in Fig. 5. Suppose that structures are produced in the order shown in the figure. Then, the second and fourth structures do not contribute new tuples to the extensions of fields `left` and `right`, if the remaining structures are included in the suite. Since we are concerned with producing small test suites, this negative result stating that field-exhaustive suites computed by our algorithm may not be minimal needs to be further examined. As we will prove in Theorem 2, field-exhaustive suites computed by our algorithm are bounded by the size of field extensions.

THEOREM 2. *Let us consider a testing subject $C_1, \ldots, C_n$. Let $F = \bigcup_{1 \le i \le n}\{f_1^{C_i}, \ldots f_{l(C_i)}^{C_i}\}$ be the set of class fields in the testing subject. Given a scope k, for each $f \in F$ let $E(f)$ be the extension of field f in scope k. Let $\#E(f)$ be the number of pairs in $E(f)$. Then, the size of the field-exhaustive test suite produced by Algorithm 1 is at most $\sum_{f \in F} \#E(f)$.*

PROOF. Each iteration of the loop in lines 6–14 covers at least one new pair in the extension of some field $f \in F$. Therefore, in at most $\sum_{f \in F} \#E(f)$ iterations the loop must terminate. Since each iteration adds one new valid instance to the suite, the size of the resulting field-exhaustive test suite must be at most $\sum_{f \in F} \#E(f)$. $\square$

Notice that, for every field $f : C \to C'$, its extension in scope k is a subset of the set of all (binary) tuples formed with objects of type C and objects/values of type C', in scope k. Since both these sets are bounded by k, the extension of f is bounded by k^2, and therefore, according to Theorem 2, field-exhaustive suites computed by `ComputeSuite` can have at most $\#F \times k^2$, for scope k.

6. EVALUATION

In this section we perform an experimental assessment of field-exhaustive testing, and the algorithm for automatically computing field-exhaustive suites. All experiments were run on a workstation with Intel Core i5 4460 processor, 6Mb Cache, 3.2Ghz (3.4 Turbo), and 16 Gb of RAM. The first part of the evaluation compares field-exhaustive test generation with bounded exhaustive test generation and random test generation, in various aspects such as size of suites, generation and testing times, as well as the quality of the obtained suites in terms of mutation score and branch coverage (a mutant is considered killed if, when run on an input, violates the corresponding method postcondition, which includes the structure's invariant). We do so on a number of data structure implementations with increasingly complex invariants. These are: *(i)* an implementation of sorted singly linked lists taken from Korat's benchmarks (we test routines `get`, `add` and `remove`); *(ii)* an implementation of binary search trees (we test routines `add`, `find` and `remove`); *(iii)* an implementation of red-black trees (we test routines `add`, `remove` and `contains`); and *(iv)* an implementation of binomial heaps (we test routines `extractMin`, `delete` and `insert`). In these analyses, we impose as a restriction a maximum of 1 hour for generation time and 1 hour for testing time, and 1 million structures for suite size. Notice that the 1 million limit is on structures, not inputs; inputs are, in most cases, composed of the structure where the method is executed, combined with the additional parameters received by the corresponding routine (e.g., `find` in binary search trees receives the key to be searched for). Since in tables we report inputs, we have cases that exceed 1 million, although these are always built with at most 1 million structures. Inputs corresponding to the additional method parameters are also generated using the corresponding approach (i.e., bounded exhaustively for bounded exhaustive testing experiments, field-exhaustively for field-exhaustive testing experiments, etc.). When generation or testing times are exceeded, we mark so in the table with TO (timeout). When the limit on the number of structures is exceeded, we mark so in the table with IL (input limit). The results comparing field-exhaustive (FE) testing with bounded exhaustive (BE) testing are summarized in Table 1. The results for random testing are shown in a separate table, Table 3, since random testing does not depend on a scope, as the other two techniques. In both tables, size reports number of test cases (test suite size). Generation and testing times are in seconds (testing time is the sum of testing all routines for each case study). Mutation score and branch coverage report the percentage of killed mutants and branches covered, respectively, again as a whole for all routines of each case study. Mutants were computed using muJava [25]. We used Korat [5] for bounded exhaustive suite generation, and Randoop [27] for random test suite generation, in these sets of experiments. For the case of random testing, we ran the tool setting a maximum number of test inputs (column size in Table 3), three times for each limit with different seeds, and report the maximum coverage/mutation score obtained.

The second part of the evaluation measures the ability of our computed field-exhaustive suites in finding real bugs, on the refactoring engine that ships with Eclipse 3.3. We compare with ASTGen [9], a tool for testing programs manipulating abstract syntax trees that bounded exhaustively combines input parts produced by user-written generators, and QuickCheck [7], a tool for the generation of random suites (in this case our testing subject does not admit the use of Randoop, since we do not have Java classes capturing the generators). We chose as case studies the 3 ASTGen generators that produced fewer inputs, and the 3 ASTGen generators the produced more inputs, among all cases in [9]. So, we compare our approach with ASTGen and random testing on six different testing cases, corresponding to three refactoring operations, namely `EncapsulateField`, `pullUpField` and `pullUpMethod`. Since this comparison is black-box, we report only time for test generation and text execution in each of the approaches, size of the corresponding suites, and bugs found. Since ASTGen does not discriminate generation time from testing time, we report these as a whole. For running ASTGen, we used the generators provided in [3]. For running QuickCheck, we translated ASTGen generators as grammars in Haskell, and used QuickCheck on the resulting Haskell programs to produce the inputs; we produced randomly generated suites of various sizes, covering the sizes of our field-exhaustive suites, and of those produced with ASTGen. For random testing, for each size, we generated 3 suites and took the average testing time, and maximum found bugs across runs. In order to apply our field-exhaustive approach, we manually wrote corresponding Alloy specifications that capture ASTGen generators, and used these specifications to produce field-exhaustive suites using incremental SAT (intuitively, our Alloy specifications captured the elements of Java programs as Alloy signatures, and their relationships, e.g., a field belonging to a class, as Alloy fields). Table 2 compares ASTGen, our field-exhaustive approach, and random testing. The first two produce fixed numbers of tests (basically, the scopes in this case are established by the alternatives for basic program elements in ASTGen, which are fixed). In the case of random testing, we produced increasingly larger sets of tests until the maximum number of bugs were found (known from the results in [9]). Table 2 also reports the sum of test generation and execution times (test generation is negligible both in ASTGen and random testing, so time in these cases are essentially just test execution), and the number of bugs found in each case. The latter are computed examining the refactorings outputs, helped by ASTGen's oracles.

6.1 Assessment

Our first set of experiments evaluates various aspects at the same time. First, it measures how efficiently our algorithm can compute field-exhaustive suites, on data structure case studies. Table 1 shows that, for the evaluated

Table 1: Bounded exhaustive vs field exhaustive on data structures

Sorted Singly Linked List

Scope	Size		Gen. time		Test. time		Mut. score		Branch cov.	
	BE	FE	BE	FE	BE	FE	BE	FE	BE	FE
3	90	45	0.12	0.45	0.03	0.02	92.58	88.76	100	100
4	420	96	0.13	0.54	0.04	0.02	92.81	92.13	100	100
5	1890	168	0.13	0.61	0.07	0.03	92.81	92.81	100	100
6	8316	264	0.15	0.70	0.15	0.03	92.81	92.81	100	100
7	36036	384	0.18	0.84	0.29	0.04	92.81	92.58	100	100
8	154440	720	0.25	1.12	0.50	0.04	93.03	91.69	100	100
9	656370	870	0.44	1.36	1.29	0.05	93.03	91.69	100	100
10	2771340	1125	0.89	1.60	4.49	0.05	93.03	91.69	100	100
11	11639628	1335	2.72	1.91	19.03	0.06	93.03	91.69	100	100
12	IL	1665	IL	2.68	IL	0.06	IL	91.69	IL	100
20		6300		8.87		0.10		91.69		100
30		14364		60.76		0.18		93.03		100
64		77637		3078.10		0.91		91.69		100
65		TO		TO		TO		TO		TO

Binary Search Tree

Scope	Size		Gen. time		Test. time		Mut. score		Branch cov.	
	BE	FE	BE	FE	BE	FE	BE	FE	BE	FE
3	135	99	0.19	0.47	0.03	0.02	89.69	88.70	98	98
4	612	216	0.18	0.55	0.04	0.03	91.36	90.77	100	98
5	2820	336	0.23	0.72	0.10	0.04	91.36	91.36	100	100
6	13158	456	0.31	0.85	0.20	0.05	91.36	91.36	100	100
7	61950	612	0.62	1.14	0.36	0.05	91.36	91.36	100	100
8	293640	810	2.14	1.33	0.73	0.06	91.36	91.36	100	100
9	1399194	1365	14.85	1.89	2.40	0.07	91.36	91.36	100	100
10	6695730	1290	123.02	2.06	10.71	0.08	91.36	91.36	100	100
11	32156091	1905	1024.21	2.87	51.56	0.08	91.36	91.36	100	100
12	TO	2145	TO	4.32	TO	0.09	TO	91.36	TO	100
15		3015		17.55		0.11		91.36		100
20		6624		982.50		0.16		91.36		100
21		7704		2022.15		0.17		91.36		100
22		TO		TO		TO		TO		TO

TreeSet

Scope	Size		Gen. time		Test. time		Mut. score		Branch cov.	
	BE	FE	BE	FE	BE	FE	BE	FE	BE	FE
3	144	90	0.59	0.63	0.05	0.03	60.39	53.53	74	67
4	492	216	0.22	0.81	0.04	0.03	75.65	72.68	86	84
5	1725	324	0.28	1.00	0.08	0.04	82.50	81.04	90	89
6	5886	492	0.39	1.21	0.15	0.05	83.22	82.92	90	90
7	19131	684	0.55	1.55	0.27	0.06	83.53	79.26	90	86
8	59736	1035	0.99	1.97	0.40	0.08	85.24	81.93	90	88
9	182331	1230	2.93	2.50	0.69	0.08	85.42	84.34	90	90
10	553170	1575	10.52	4.12	1.39	0.09	85.49	85.31	90	90
11	1690986	1800	52.00	5.31	3.72	0.11	85.56	84.27	90	87
12	5258628	2685	283.42	7.82	10.78	0.14	85.56	84.67	90	88
13	16706859	2730	1531.76	10.83	35.00	0.13	85.56	85.38	90	90
14	TO	2730	TO	14.13	TO	0.13	TO	85.44	TO	90
15		3195		21.24		0.15		84.30		87
20		7326		197.29		0.20		84.69		89
25		10332		1022.75		0.27		84.70		88
30		14346		2894.94		0.34		83.81		87
31		TO		TO		TO		TO		TO

Binomial Heap

Scope	Size		Gen. time		Test. time		Mut. score		Branch cov.	
	BE	FE	BE	FE	BE	FE	BE	FE	BE	FE
3	197	63	0.22	0.68	0.03	0.03	77.28	76.36	82	82
4	1081	126	0.23	0.85	0.07	0.03	78.40	78.13	84	84
5	10297	216	0.34	1.42	0.24	0.03	80.09	78.39	84	84
6	98827	315	0.41	2.55	0.57	0.05	85.28	83.52	89	89
7	1611241	423	0.90	2.91	4.65	0.05	85.85	85.25	90	90
8	10263649	671	3.11	2.82	27.37	0.07	86.00	83.01	90	87
9	IL	847	IL	3.67	IL	0.07	IL	78.70	IL	84
10		1078		5.13		0.09		85.12		89
11		1298		6.55		0.09		82.37		89
15		2453		27.42		0.14		90.22		90
20		5109		325.55		0.19		94.01		93
25		8099		1174.16		0.24		94.74		93
26		TO		TO		TO		TO		TO

case studies, field-exhaustive generation generally scales to more than twice the scope of bounded exhaustive generation, measured as the maximum scope that can be covered within 1 hour. It is worth observing that the size of our computed field-exhaustive suites grows linearly as the scope is increased, as opposed to the known exponential growth of bounded exhaustive suites. Regarding the quality of the computed suites, Table 1 show that for scopes that

Table 2: ASTGen vs random vs field exhaustive

	ASTGen			Random			Field-Exhaustive		
Refactor	Inputs	Time	Bugs found	Inputs	Time	Bugs found	Inputs	Time	Bugs found
EncapsulateField.SingleClassFieldRef	5,280	4,218	4	10	9.13	0	107	113.4	4
				100	80.36	1			
				200	160.32	3			
				300	240.81	3			
				400	321.85	2			
				500	401.00	3			
				600	482.79	3			
				700	553.80	4			
pullUpMthd.TripleClassMethodChild	1,152	1,091.53	2	10	10.75	1	16	18.57	2
				20	18.65	2			
EncapsulateFld.DoubleClassFieldRef	23,760	16,480	4	100	76.81	0	167	176.69	**3**
				1,000	769.40	3			
				10,000	7,689.75	4			
EncapsulateField.ClassArrayField	72	75	1	10	10.97	1	16	18.25	1
EncapsulateFld.SingleClassTwoFields	60	57	1	10	10.53	1	10	12.05	1
pullUpField.DoubleClassChildField	72	68	1	10	9.97	1	10	12.15	1

Table 3: Random Testing on data structs.

Sorted S. Linked List

Size	Gen. time	Test. time	Mut. score	Branch cov.
1000	0	0.12	92.81	100
5000	1	0.41	93.03	100

Binary Search Tree

Size	Gen. time	Test. time	Mut. score	Branch cov.
1000	0	0.10	88.90	98
5000	1	0.38	90.96	100
10000	1	0.71	91.36	100

TreeSet

Size	Gen. time	Test. time	Mut. score	Branch cov.
1000	0	0.11	62.53	75
5000	1	0.51	78.55	88
10000	1	0.83	82.64	89
50000	4	3.39	85.17	90
100000	9	6.95	85.19	90
300000	34	20.68	85.21	90
1000000	118	48.67	85.35	90

Binomial Heap

Size	Gen. time	Test. time	Mut. score	Branch cov.
1000	0	0.10	78.62	84
5000	1	0.43	85.06	89
10000	1	0.86	85.70	90
50000	6	3.05	85.78	90
100000	13	6.38	86.12	90
300000	42	18.44	86.14	90
700000	110	43.07	94.74	93

are reached both by field-exhaustive and bounded exhaustive suites, the mutation score and branch coverage obtained are very similar, with less that 2% difference from scope 6. Considering the largest produced suites in each case, field-exhaustive suites generally outperform bounded exhaustive suites both in branch coverage and mutation score, in a case in more that 8% of its mutation score, with significantly smaller suites and corresponding testing times. Regarding the comparison with random testing, Randoop produces suites very efficiently, outperforming in test generation both the bounded-exhaustive and field-exhaustive approaches. However, the sizes of test suites required to reach similar mutation scores and coverage as bounded-exhaustive and field-exhaustive suites are in most cases significantly larger. This can be observed in all cases except linked lists. For instance, it takes 700,000 tests for Randoop to achieve 93% branch coverage for binomial heaps, while field-exhaustive testing achieves the same coverage with 5109 tests.

The second part of the experiments concentrates on the ability of field-exhaustive suites to catch real bugs. We compared with ASTGen, a tool particularly tailored to testing source code manipulating programs, which has been used to find real bugs in refactoring engines [9], and QuickCheck [7], a random test input generation tool. Out of the 13 bugs found by ASTGen with a total of 30,396 test inputs, our field-exhaustive approach was able to catch 12, with a total of 326 test inputs. Random testing, on the other hand, had to make the test set grow up to 10,750 to catch the 13 bugs. It is however important to notice that random testing performed very well on most case studies; it outperformed ASTGen in number of tests to catch all bugs. Still, field-exhaustive testing showed in this case the best ratio in number of inputs per bugs found.

All the presented experiments can be reproduced following the instructions available in [12].

6.2 Threats to Validity

Our experimental evaluation is limited to data structures and a refactoring engine. Both are good representatives of data characterized by complex constraints, which are challenging for automated testing techniques. From the vast domain of data structures, we have selected a few that we consider representative for several reasons, because they are often used as case studies in the evaluation of other analysis tools [5, 10, 19, 35], and their invariants have varied complexity (which is a dimension that affects field extensions' size, and thus also affects the computation of field-exhaustive suites). One might argue that restricting the analysis to these case studies might favor our results. While an exhaustive evaluation of data with complex constraints is infeasible, we identified that invariant complexity is a dimension that affects field extensions' size, and thus their computation, and designed the experiments to take this problem into account. We selected structures with a broad range of complexity, going from those with simple invariants to some with rather complex constraints.

The selection of tools for comparison might have accidentally favored our techniques. We selected Korat based on the evaluation of several alternatives, and the analysis in [31]; despite being older than other tools, it is the most efficient choice for our data structure case studies. Also for this tool, the implementation of imperative representation invariants affects its efficiency. We used the imperative invariants provided with the tool's examples, which are tailored to Korat's bounded exhaustive suites' computation approach.

Regarding our comparison with ASTGen and random testing on bug finding, we manually produced representations of ASTGen generators both for QuickCheck and Alloy, to enable our field-exhaustive approach. We did not formally verify the equivalence with the generators, but we manually corroborated that the inputs produced by the three alternatives were consistent across tools, to discard the possibility of improved efficiency due to errors in the translations. Also, the selection of the six cases for refactoring was done taking into account the corresponding numbers of inputs generated by ASTGen, and we included the three with fewer cases, where ASTGen shows better ratio between tests and number of bugs catched. More recent tools such as UDITA [17] may be used instead of ASTGen. However UDITA improves only test generation time, not the tests produced (which are the same as ASTGen's) nor the test execution time; thus, UDITA would not produce noticeable improvements in our experiments. In [12], we provide some additional experiments using UDITA, confirming this observation.

7. RELATED WORK

Automated test case generation is currently a very active area of research, and many tools and techniques have been developed in recent years. Among these approaches, the most effective and successful are either based on random generation [27, 24, 7], evolutionary computation [13], model checking [35], constraint solving (including SMT [33] and SAT solving [1]) or some forms of exhaustive search [5].

Test suites generated by tools based on random generation and evolutionary computation such as Randoop [27], AutoTest [24], QuickCheck [7] and EvoSuite [13], tend to generate *large* test sets, which has as a consequence an increased testing time, a problem we try to avoid with field-exhaustive testing. Our approach corresponds to *systematic* test generation, in the terminology of [30], which makes it closer to tools like Pex [33], FAJITA [1], Symbolic PathFinder [34] and Korat [5]. These tools are also *specification based*, i.e., they produce tests from input specifications, as opposed to tools like EvoSuite or Randoop, that use routines/methods of the testing subject to produce tests. With respect to Korat, field-exhaustive testing is already extensively compared to bounded exhaustive testing in the paper, in particular against this tool. Tools that produce small test sets, generally driven by some testing criterion (e.g., Pex, FAJITA, Symbolic PathFinder), usually concentrate on white box coverage, as opposed to our automated testing approach.

In [30] a set of experiments compare random testing and systematic testing for container classes, arriving to the conclusion that random testing produces suites that are comparable in quality (coverage, mutation score) to systematically produced suites (using shape abstraction), while consuming less computational resources (less generation time). We compared our approach with random testing, in the first set of examples against Randoop (because the testing subject admits such comparison), and in the second case against QuickCheck. Our results show that, while we still consume more computation resources, the produced suites are comparable in quality with those produced with random testing, but achieve such quality with significantly fewer test cases.

Incremental SAT has been used by other tools, in particular in FAJITA [1] for automated test input generation, and in [8] for generating combinatorial interaction tests for product families. Other tools use SAT/SMT solving as part of the test generation process. TestEra [21] uses incremental SAT-solving for exhaustive bounded generation of input data. It is then very close to Korat, while Korat shows better performance. Whispec [29] builds on specification-based testing and focuses on maximizing code coverage by iteratively running a conjunction of method preconditions and path conditions. Pex uses SMT solving to produce minimal suites maximizing branch coverage.

In our approach, computing field-exhaustive suites produces field extensions. These extensions are equivalent to (upper) *tight field bounds*, and thus our work is related to approaches to compute such bounds, e.g., [15, 28, 4]. However, none of these related approaches focuses on test generation; [15] does not collect instances in the tight bound computation process (and even if it did so, it would be significantly more inefficient than our approach, since it requires a cluster for tight bound computation); [28, 4] follow a bounded exhaustive enumeration of instances to compute tight bounds, as opposed to our field-exhaustive mechanism.

8. CONCLUSION AND FUTURE WORK

Thanks to advances in automated testing technologies, new testing criteria, infeasible in manual testing contexts, are now emerging. Among the many tools that produce test suites automatically from code, some produce large suites that make their use difficult in settings that demand repetitive use, like regression testing, while others produce small suites, typically driven by traditional (most often white box) testing criteria. We proposed a new *black box* testing criterion, that we called *field-exhaustive testing*, whose satisfaction is associated with the coverage of feasible values for object fields. In effect, the criterion requires a bound (or scope) to be provided, so the space of valid object instances is finite and limited, and demands enough inputs to be produced so that every feasible value for every field is covered. Besides formally defining the criterion, we developed an algorithm that automatically produces field-exhaustive suites, and showed that suites can be generated efficiently, are composed of relatively small numbers of tests, and are of very good quality, measured in terms of mutation score and branch coverage. Moreover, the produced suites are comparable in coverage and mutation score to suites produced using bounded exhaustive testing and random testing tools, with significantly fewer tests.

Our testing criterion and test generation algorithm lead to various new research problems, that we plan to study. The test generation algorithm produces field extensions, which correspond to tight bounds [15, 4], and can be used to improve SAT-based analysis. While in this paper we concentrated in testing, this algorithm can be used to compute tight field bounds, and exploit these for other analyses. We plan to study this further in the future. Also, while in this paper we produced field-exhaustive suites for *all* fields in the corresponding testing subjects, one may select an appropriate subset of fields, and generate field-exhaustive suites just for this set. The right set of fields to select will generally depend on characteristics of the testing subject, and the kind of bugs one is searching for. For instance, choosing only reference fields in heap allocated data structures would imply concentrating on structures' shapes, not the data, and thus would target bugs related to (re)linking structures' objects.

9. REFERENCES

[1] P. Abad, N. Aguirre, V. Bengolea, D. Ciolek, M. F. Frias, J. P. Galeotti, T. Maibaum, M. Moscato, N. Rosner and I. Vissani, *Improving Test Generation under Rich Contracts by Tight Bounds and Incremental SAT Solving*, in Proceedings of Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, IEEE, 2013.

[2] P. Ammann and J. Offutt, *Introduction to Software Testing*, Cambridge University Press, 2008.

[3] ASTGen project site: `http://mir.cs.illinois.edu/astgen/`

[4] H. Bagheri and S. Malek, *Titanium: Efficient Analysis of Evolving Alloy Specifications*, in Proceedings of ACM SIGSOFT International Symposium on the Foundations of Software Engineering FSE 2016, ACM, 2016.

[5] C. Boyapati, S. Khurshid and D. Marinov, *Korat: Automated Testing based on Java Predicates*, in Proceedings of International Symposium on Software Testing and Analysis ISSTA 2002, ACM, 2002.

[6] P. Chalin, J. Kiniry, G. Leavens and E. Poll, *Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2*, in Proceedings of the 4th International Conference on Formal Methods for Components and Objects FMCO 2005, 2005.

[7] K. Claessen and J. Hughes, *QuickCheck: a lightweight tool for random testing of Haskell programs*, in Proceedings of the fifth ACM SIGPLAN international conference on Functional programming ICFP 2000, ACM, 2000.

[8] M. Cohen, M. Dwyer and J. Shi, *Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach*, Trans. Software Eng. 34(5), IEEE, 2008.

[9] B. Daniel, D. Dig, K. Garcia and D. Marinov, *Automated Testing of Refactoring Engines*, in Proceedings of 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering ESEC/FSE 2007, ACM, 2007.

[10] G. Dennis, F. Chang and D. Jackson, *Modular Verification of Code with SAT*, in Proceedings of the 2006 International Symposium on Software Testing and Analysis ISSTA 2006, ACM, 2006.

[11] N. Eén and N. Sörensson, *An Extensible SAT-Solver*, in Proceedings of the 6th International Conference on the Theory and Applications of Satisfiability Testing, SAT 2003, Santa Margherita Ligure, Italy, May 5-8, Selected Revised Papers, Lecture Notes in Computer Science, Springer, 2004.

[12] Field-Exhaustive Testing, experiments site: `https://sites.google.com/site/fieldexhaustivetesting/`

[13] G. Fraser and A. Arcuri, *EvoSuite: automatic test suite generation for object-oriented software*, in Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering ESEC/FSE 2011, ACM, 2011.

[14] M. Frias, J. Galeotti, C. López Pombo and N. Aguirre, *DynAlloy: Upgrading Alloy with Actions*, in Proceedings of the 27th International Conference on Software Engineering, ICSE 2005, 15-21 May 2005, St. Louis, Missouri, USA, ACM, 2005.

[15] J. P. Galeotti, N. Rosner, C. López Pombo and M. F. Frias, *Analysis of invariants for efficient bounded verification*, in Proceedings of the 19th International Symposium on Software Testing and Analysis ISSTA 2010, ACM, 2010.

[16] J. P. Galeotti, N. Rosner, C. López Pombo and M. Frias, *TACO: Efficient SAT-Based Bounded Verification Using Symmetry Breaking and Tight Bounds*, IEEE Transactions on Software Engineering 39(9), IEEE, 2013.

[17] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak and D. Marinov, *Test Generation through Programming in UDITA*, in Proceedings of International Conference on Software Engineering ICSE 2010, ACM, 2010.

[18] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, The MIT Press, 2006.

[19] D. Jackson and M. Vaziri, *Finding Bugs with a Constraint Solver*, in Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis ISSTA 2000, ACM, 2000.

[20] C. Kaner, J. Bach and B. Pettichord, *Lessons Learned in Software Testing*, Wiley, 2001.

[21] S. Khurshid and D. Marinov, *TestEra: Specification-Based Testing of Java Programs Using SAT*, Autom. Soft. Eng. 11(4), Kluwer Academic, 2004.

[22] D. Kroening and M. Tautschnig, *CBMC – C Bounded Model Checker*, in Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS 2014, LNCS 8413, Springer, 2014.

[23] J. N. Hooker, *Solving the incremental satisfiability problem*, Journal of Logic Programming 15(1), Elsevier, 1993.

[24] L. Liu , B. Meyer and B. Schoeller, *Using contracts and boolean queries to improve the quality of automatic test generation*, in Proceedings of the 1st International Conference on Tests and Proofs TAP 2007, LNCS 4454, Springer, 2007.

[25] Y.-S. Ma, J. Offutt and Y.-R. Kwon, *MuJava : An Automated Class Mutation System*, Journal of Software Testing, Verification and Reliability, 15(2), Wiley, 2005.

[26] G. Myers and C. Sandler, *The Art of Software Testing*, John Wiley & Sons, 2004.

[27] C. Pacheco, S. K. Lahiri, M. D. Ernst and T. Ball, *Feedback-Directed Random Test Generation*, in Proceedings of the 29th international conference on Software Engineering ICSE 2007, IEEE, 2007.

[28] P. Ponzio, N. Rosner, N. Aguirre and M. Frias, *Efficient Tight Field Bounds Computation based on Shape Predicates*, in Proceedings of the 19th International Symposium on Formal Methods FM 2014, May 12-16, Singapore, Lecture Notes in Computer Science, Springer, 2014.

[29] D. Shao, S. Khurshid and D. Perry, *Whispec: white-box testing of libraries using declarative specifications*, in Proceedings of the Symposium on Library-Centric Software Design LCSD 2007, ACM, 2007.

[30] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser and D. Marinov, *Testing container classes: random or systematic?*, in Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering FASE 2011, LNCS 6603, Springer, 2011.

[31] J. Siddiqui and S. Khurshid, *An Empirical Study of Structural Constraint Solving Techniques*, in Proceedings of 11th International Conference on Formal Engineering Methods ICFEM 2009, LNCS 5885, Springer, 2009.

[32] K. Sullivan, J. Yang, D. Coppit, S. Khurshid and D. Jackson, *Software assurance by bounded exhaustive testing*, in Proceedings of International Symposium on Software Testing and Analysis ISSTA 2004, ACM, 2004.

[33] N. Tillmann, J. de Halleux, *Pex: White Box Test Generation for .NET*, in Proceedings of the 2nd International Conference on Tests and Proofs TAP 2008, LNCS 4966, Springer, 2008.

[34] W. Visser, C. S. Pasareanu and S. Khurshid, *Test Input Generation with Java PathFinder*, in Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis ISSTA 2004, ACM, 2004.

[35] W. Visser, C. S. Pasareanu and R. Pelánek, *Test Input Generation for Java Containers using State Matching*, in Proceedings of the International Symposium on Software Testing and Analysis ISSTA 2006, ACM, 2006.

[36] H. Zhu, P. Hall and J. May, *Software Unit Test Coverage and Adequacy*, Computing Surveys 29(4), ACM, 1997.