Bounded Exhaustive Test Input Generation from Hybrid Invariants

Nicolás Rosner
Dept. of Computer Science
FCEyN, University of Buenos Aires
Buenos Aires, Argentina
nrosner@dc.uba.ar

Valeria Bengolea,
Pablo Ponzio
Dept. of Computer Science
FCEFQyN, University of Rio Cuarto
Rio Cuarto, Argentina
{vbengolea, pponzio}
@dc.exa.unrc.edu.ar

Shadi Abdul Khalek
Google, USA
ak.shadi@gmail.com

Nazareno Aguirre
Dept. of Computer Science
FCEFQyN, University of Rio Cuarto
and CONICET
Rio Cuarto, Argentina
naguirre@dc.exa.unrc.edu.ar

Marcelo F. Frias
Dept. of Software Engineering
Instituto Tecnólogico de Buenos
Aires and CONICET
Buenos Aires, Argentina
mfrias@itba.edu.ar

Sarfraz Khurshid
Electrical and Computer Engineering
The University of Texas at Austin
Austin, USA
khurshid@ece.utexas.edu

Abstract

We present a novel technique for producing bounded exhaustive test suites from hybrid invariants, i.e., invariants that are expressed imperatively, declaratively, or as a combination of declarative and imperative predicates. Hybrid specifications are processed using known mechanisms for the imperative and declarative parts, but combined in a way that enables us to exploit information from the declarative side, such as *tight bounds* computed from the declarative specification, to improve the search both on the imperative and declarative sides. Moreover, our technique automatically evaluates different possible ways of processing the imperative side, and the alternative settings (imperative or declarative) for parts of the invariant available both declaratively and imperatively, to decide the most convenient invariant configuration with respect to efficiency in test generation. This is achieved by transcoping, i.e., by assessing the efficiency of the different alternatives on small scopes (where generation times are negligible), and then extrapolating the results to larger scopes.

We also show experiments involving collection classes that support the effectiveness of our technique, by demonstrating that *(i)* bounded exhaustive suites can be computed from hybrid invariants significantly more efficiently than doing so using state-of-the-art purely imperative and purely declarative approaches, and *(ii)* our technique is able to automatically determine efficient hybrid invariants, in the sense that they lead to an efficient computation of bounded exhaustive suites, using transcoping.

Keywords automated test generation; bounded exhaustive testing; SAT solving; Korat; Alloy; transcoping

OOPSLA '14, October 20–24, 2014, Portland, OR, USA.

http://dx.doi.org/10.1145/2660193.2660232

1. Introduction

It is widely acknowledged that software testing is a major engineering approach for guaranteeing software quality [17]. Software testing is essential for software development, but it is also highly time-consuming, so that automating testing-related tasks becomes crucial in helping software developers and encouraging adoption of testing practices. Some testing tasks, such as test execution, are easily automated. Others, in particular test input generation, are typically very hard to automate. Despite the inherent complexity of automating test input generation, various techniques and tools have been proposed to automatically produce test inputs, including some based on random generation [5, 26] as well as others based on several different forms of constraint solving or model checking [1, 2, 14, 21, 32].

Most tools and techniques for test input generation require a *specification* of the inputs of the program under analysis. This specification can be imperative, as is the case with Korat [2], UDITA [14] and Symbolic PathFinder [32], or it can be declarative, i.e., provided as a formula in some logical formalism. Tools like TestEra [21], TACO [11] and Forge [7] lie in the latter category. This essentially leads to two families of approaches: imperative-specification-based and declarative-specification-based, depending on the kind of language used to express the specifications of inputs.

Although the aforementioned tools for test generation are powerful, they often exhibit some common issues from the point of view of the end user. In particular, since generation approaches are often tightly coupled to particular kinds of specifications (imperative or declarative), once a tool or test generation approach is chosen, one is forced to write specifications within a particular paradigm, i.e., either imperatively or declaratively. This may not be a problem when attempting to generate test cases for programs with simple inputs (e.g., parameters of basic datatypes, or inputs with just a few elementary conditions), where input specification is usually straightforward. But it certainly is a limitation when dealing with programs that manipulate complex data, where input specification often becomes elaborate. Indeed, in many cases, and in particular in the context of heap-allocated data structures, input specifications tend to be complex and comprise several conditions on the inputs. Having to express all of them in the same style may turn out to be unnatural for engineers, and consequently prone to errors.

Input specification is rendered even more complex by the fact that most test generation mechanisms are sensitive to the precise way in which the specification is (syntactically) expressed. This is especially the case with imperative-specification-based approaches. For instance, when checking two or more restrictions on the inputs (e.g., acyclicity and balance in a tree-like structure), doing so in different orders may lead to substantially different running times for test generation (e.g., from 8.6 seconds to over 8.5 hours, depending on the particular ordering, for the generation of red-black trees with up to 8 nodes – see Section 6) from an imperative specification.

To help overcome the problems described above, in this article we present HyTeK, a technique for bounded exhaustive test input generation with the following characteristics:

- HyTeK automatically generates test suites from input specifications given in the form of *hybrid* invariants. These invariants are hybrid in the sense that they may be provided imperatively, declaratively, or as a combination of declarative and imperative predicates. Methodologically, this allows software engineers to design specifications that better reflect the nature of the problem being modeled, or that better fit their specification preferences, and are therefore less error prone.
- Since automated test input generation is highly sensitive to the way in which the invariant is implemented, HyTeK automatically explores alternative orderings of the specification components on the imperative side, and in the event that part of the invariant is provided both declaratively and imperatively, it decides the most convenient setting (imperative or declarative) in which it is to be solved, so that the efficiency of test input generation is improved. This is done by *transcoping* [27], i.e., by assessing the efficiency of the alternatives on small scopes, where generation times are negligible, and then extrapolating the results to larger scopes, where costs are much higher and a single bad decision can render the whole generation task infeasible.
- While approaches based on fully declarative (resp. imperative) specifications have (and, in fact, each particular tool has) their own associated optimization techniques, which are in general difficult to translate to other contexts, the availability of different styles of specification in the same invariant enables HyTeK to benefit from optimization approaches of one side in the other one. This is achieved through two mechanisms. First, by using information obtained while solving declarative portions of the invariant we are able to assist in *pruning* the search for partially valid structures from the imperative portion of the specification. Second, we can compute *tight bounds* [11] from the declarative invariant, and use these during test generation both from the declarative and imperative parts of the specification, to reduce the search space.

The aforementioned technique is proposed for *bounded exhaustive testing*, an approach followed by several testing tools [2, 14, 21, 24, 33], which consists of testing a piece of software on *all* valid inputs within a certain scope (e.g., maximum number of objects involved in heap allocated inputs, ranges for numerical inputs, etc.). Since this testing approach is intrinsically combinatorial, automated input generation is clearly a necessity in this context. Our technique combines a mechanism for processing imperative input specifications introduced in [2] through the Korat tool, with SAT solving for processing the declarative portions of the input specification, in the style put forward through the tool TestEra [21]. The combination of these two approaches motivates the name of our technique: HyTeK stands for **Hy**brid **Te**stEra-**K**orat.

Bounded exhaustive testing has proved to be particularly effective for testing complex data structure implementations. In order to evaluate the effectiveness of HyTeK, we develop experiments involving collection classes, including a red-black-tree-based implementation of sets (TreeSet from the java.util package), which is among the most involved commonly used data structures with respect to invariant complexity.

Our experimental results show that:

- Bounded exhaustive suites can be computed from hybrid invariants significantly more efficiently than doing so using state-of-the-art purely imperative and purely declarative approaches, and
- HyTeK is able to automatically discover efficient hybrid invariants (in the sense that they lead to an efficient automated generation of test suites) using transcoping.

2. Hybrid Input Specifications

Many automated program analysis techniques require specifications of the programs under analysis. Test input generation tools are no exception – in order to automatically generate tests, these tools often require a specification of the valid inputs of the program under consideration [1, 2, 5, 14, 21, 32]. Several different approaches exist for expressing input specifications, which in many cases can become quite intricate and complex to express. Often, particularly in object-oriented programs, these input specifications are given (at least partly) in the form of *class invariants*, also known as *representation invariants*, of the input datatypes. In order to illustrate such specifications, let us consider an interesting and complex data structure, *red-black trees*. Red-black trees are balanced binary search trees. They are used as the implementation of class TreeSet in package java.util. The class invariant for red-black trees comprises the following constraints:

rbt1: the structure is a tree,

rbt2: the tree is a binary search tree,

rbt3: each node has a color, which can be red or black,

rbt4: the root node is black,

rbt5: no two consecutive nodes in a path can be red, and

rbt6: every path from the root to a leaf node contains the same number of black-colored nodes.

This representation invariant of red-black trees can be thought of as an input specification of routines handling red-black trees, such as the insertion and deletion routines defined in class `TreeSet`. An *imperative* implementation of the representation invariant is typically given as a `repOK` routine [23], e.g., as a boolean Java function that returns true if and only if a red-black tree object's structure is internally consistent, that is, it satisfies all six constraints above. For instance, assuming that red-black trees are implemented in a way that is consistent with the classes shown in Figure 1, then the associated representation invariant can be imperatively captured by the `repOK` routine partially shown in Figure 2.

A different approach to provide input specifications (or, as in our case, a representation invariant for a given class) is to do so *declaratively*. This involves using some appropriate logical setting, and expressing the input specification or representation invariant as a logical formula. Such logical settings are widely available, for instance as languages for contract specification accompanying object-oriented programming languages (e.g., JML [3]), or as specific formal specification languages. In this article we will use Alloy [16], a popular relational formal specification language that is well suited for this kind of specification. This choice is made without loss of generality – any declarative specification language could be used for this task. The red-black tree invariant can be expressed in Alloy as (partially) shown in Figure 3.

```
class TreeSet {
        Node root;
        int size;
}

class Node {
        Node parent;
        Node left;
        Node right;
        int color;
        int key;
}
```

Figure 1. Classes for red-black trees implementation.

The imperative and declarative invariants above are, of course, very closely related. Indeed, both enforce the same constraints on the input structure – they just express it in different styles. When modeling the invariant, some engineers may find some constraints easier to express in one style, while others may find them more naturally expressible in another. For instance, for many programmers, constraint **rbt1**, which requires that the structure reachable from the root be a tree, would be very directly expressible in an imperative language by means of a structure traversal (as in method `structureOK()` in Figure 2). On the other hand, developers familiar with the logical setting used for declarative specifications might favor a declarative version of this constraint, based on reachability or transitive closure operators. Furthermore, some constraints are inherently easier to express declaratively. Consider, for instance, requirements **rbt4**, **rbt5** and **rbt6**, which constrain the valid colorings of red-black trees. An imperative version of these requirements is shown in Figure 4. The declarative version of `colorsOK()`, shown in Figure 5, is much more concise.

To allow for greater flexibility when specifying inputs, we propose the use of *hybrid* invariants, which may be specified as a mix of procedural and declarative constraints. Hybrid invariants lead to a methodological improvement in specification, allowing software engineers to design specifications that better reflect the nature of the problem being modeled, or that better fit their specification preferences, making the process of specification less error prone. Moreover, enabling the possibility of describing inputs with hybrid invariants may have a significant impact on analysis (in our case, efficiency of the test input generation process). As we will show in later sections, hybrid invariants allow us to leverage optimization approaches from one context in the other one.

```
public boolean repOK() {
    //empty tree has size 0
    if (root == null)
        return size == 0;

    //root is black
    if (root.color != BLACK)
       return false;

    //tree structure is ok
    if (!structureOK())
        return false;

    //size is ok
    if (!sizeOK())
        return false;

    //coloring is ok
    if (!colorsOK())
        return false;

    //stored values are ordered
    return keysOK();
}

public boolean structureOK() {
    if (root.parent != null)
        return false;
    Set visited = new HashSet();
    visited.add(root);
    LinkedList workList = new LinkedList();
    workList.add(root);
    while (!workList.isEmpty()) {
        Node current = workList.removeFirst();
        Node cl = current.left;
        if (cl != null) {
            if (!visited.add(cl))
                return false;
            if (cl.parent != current)
                return false;
            workList.add(cl);
        }
        Node cr = current.right;
        if (cr != null) {
            if (!visited.add(cr))
                return false;
            if (cr.parent != current)
                return false;
            workList.add(cr);
        }
    }
    return true;
}

...
```

Figure 2. (Partial) An imperative implementation, written in Java, of a representation invariant for red-black trees.

In particular, by using information obtained while solving declarative portions of the invariant we are able to help prune the search for partially valid structures from the imperative portion of the specification; and having a declarative invariant allows for querying characteristics of the valid structures that lead to reductions in the search space when solving the

```
// empty tree has size 0
(thiz.root = null => thiz.size = 0) and
// root is black
(thiz.root != null => thiz.root.color = BLACK) and
// tree structure is ok
(this.root != null => thiz.root.parent = null) and
all n:TreeSetNode | {
  n in thiz.root.*(left+right) - null =>
    (
      (n.left != null => n.left.parent = n) and
      (n.right != null => n.right.parent = n) and
      (n.parent != null => n in n.parent.(left+right)) and
      n !in n.+parent
    )
} and
// size is ok
... and
// coloring is ok
... and
// stored values are ordered
...
```

Figure 3. A declarative characterization, written in Alloy, of the representation invariant for red-black trees.

imperative and declarative parts of the input specification. This will be explained in further detail in later sections.

3. Generating Test Inputs from Specifications

Our approach for generating test inputs from hybrid specifications builds on existing approaches for generating test inputs from either imperative or declarative specifications. We describe these processes below.

3.1 Test Inputs from Imperative Specifications

When representation invariants are expressed imperatively, a very efficient mechanism for generating test inputs becomes available, as put forward in [2], and exploited by other approaches [14, 32], known as the *Korat* algorithm. This algorithm, embodied in the associated homonymous tool, is able to generate test inputs from imperative specifications, and is especially targeted at the generation of complex, heap-allocated structures [2]. The Korat tool requires an imperative input specification, i.e., a `repOK()` routine specifying the expected (valid) inputs to be generated, and a *scope* definition, which provides the bounds for the domains involved in the structure. For instance, for red-black trees, the `repOK()` routine would be the one shown in Fig. 2, while the notion of scope (specified as a *finitization procedure*) would indicate the ranges for primitive-type fields as well as the minimum and maximum number of objects of each class involved in the structure. For instance, it may specify that generation has to be performed using at most 1 TreeSet object, 0 to 3 Node objects, 0...3 as the range for `TreeSet.size`, 0...2 as the range for `Node.key`, and 0...1 as the range for `Node.color` (colors being represented here by integers).

Korat generates all valid structures (i.e., structures for which `repOK()` would return true) within the provided bounds. For the red-black trees example, assuming the scope

```
private boolean colorsOK() {
    // Red has only black children
    LinkedList workList = new LinkedList();
    workList.add(root);
    while (!workList.isEmpty()) {
        Node current = workList.removeFirst();
        Node cl = current.left;
        Node cr = current.right;
        if (current.color == RED) {
            if (cl != null && cl.color == RED)
                return false;
            if (cr != null && cr.color == RED)
                return false;
        }
        if (cl != null)
            workList.add(cl);
        if (cr != null)
            workList.add(cr);
    }
    // Simple paths from root to
    // NIL have same number of black nodes
    int numberOfBlack = -1;
    workList = new LinkedList();
    workList.add(new Pair(root, 0));
    while (!workList.isEmpty()) {
        Pair p = workList.removeFirst();
        Node e = p.e;
        int n = p.n;
        if (e != null && e.color == BLACK)
            n++;
        if (e == null) {
            if (numberOfBlack == -1)
                numberOfBlack = n;
            else if (numberOfBlack != n)
                return false;
        } else {
            workList.add(new Pair(e.left, n));
            workList.add(new Pair(e.right, n));
        }
    }
    return true;
}
```

Figure 4. An imperative predicate capturing the correct coloring in a red-black tree.

```
thiz.root != null => thiz.color = BLACK and
all n : thiz.root.*(left + right) - null |
    n.color = RED and n.parent != null
                => n.parent.color = BLACK and
    n.left != null and n.right = null
                => n.left.color = RED and
    n.left = null and n.right != null
                => n.right.color = RED
```

Figure 5. An Alloy declarative constraint capturing correct coloring in red-black trees.

given above, Korat will generate every valid (i.e., well-colored) red-black tree structure of a size no greater than 3 (in this example, the value of the size field coincides with the number of nodes in the structure) and containing keys ranging from 0 to 2. In order to achieve this, Korat builds a tuple where each entry corresponds to a value of a field of the involved objects. In our example, the tuple would have length 17 – two values for the root and size of the `TreeSet` object, and 15 for the five fields of the three nodes that the tree may contain. For instance, the following tuple

$$\langle\, 0, 1, \quad \mathit{NULL}, \mathit{NULL}, \mathit{NULL}, 0, 0, \\ \mathit{NULL}, \mathit{NULL}, \mathit{NULL}, 0, 0, \\ \mathit{NULL}, \mathit{NULL}, \mathit{NULL}, 0, 0 \,\rangle$$

represents the tree of size 1 with a single node (the first zero in the tuple references the 0-th, i.e. the first node object), whose parent, left and right fields are set to null, and whose key and color are zero and black (also represented by a zero), respectively. Each entry in this tuple has a domain, which is defined by the finitization procedure. More precisely, Korat works on *representations* of these tuples, called *candidate vectors*, that represent the candidate tuples by replacing actual entries with indices into the respective domains. For instance, the candidate vector

$$\langle 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 \rangle$$

would correspond to the previously shown candidate tuple (each tuple entry has the first possible value in its domain, except for the tree's root and size). In the context of complex heap-allocated structures (as well as in many other contexts), most of the candidate vectors would correspond to invalid structures, i.e., the structures that do not satisfy the `repOK()` are usually considerably more abundant than those that do. In our example, the candidate space contains $905,969,664$ vectors (4 possibilities for each of the node-typed fields, combined with 4 possibilities for the size field, 3 for each key field, and 2 for each color field). Yet there are only 12 distinct red-black trees (up to isomorphism, as will be explained later on) within the scope of this example.

To generate test inputs, Korat exhaustively explores the space of candidate vectors. As shown in Fig. 6, Korat starts with the initial candidate vector (all indices are zero) and executes `repOK()` on this candidate, monitoring the fields accessed during execution and storing them in a stack. Korat uses this stack in order to backtrack over candidate vectors. If the current candidate satisfies `repOK()`, it is considered a valid test input. If `repOK()` fails, then the candidate is discarded. In both cases, to build the next candidate, the last accessed field is incremented to its next value. If one or more of the last accessed fields are already in their corresponding maximum values, then these are reset to 0, and the field accessed before them is incremented. If all fields are already at their maximum values, then the state space of candidate vectors has been exhaustively explored, and Korat terminates.

By backtracking only on accessed fields, Korat is able to prune large parts of the candidate vector space at a time. In fact, its backtracking process leads Korat-savvy users to get used to writing `repOK()` in particular ways that seem to benefit test input generation. Note that in order to determine that a candidate is invalid, it is often not necessary to access all reachable fields. The fewer fields accessed,

```
Algorithm korat() {
    Vector curr = initVector;
    Stack fields = new Stack();
    boolean ok;
    do {
        (ok, fields) = curr.repOK();
        if (ok) {
            reportValid(curr);
            fields.push(curr.reachFields − fields);
        }
        field = fields.pop();
        while (!fields.isEmpty() &&
                curr[field] >=
                nonIsoMax(curr, fields, field)) {
            curr[field] = 0;
            field = fields.pop();
        }
        if (!fields.isEmpty()) curr[field]++;
    } while (curr != lastVector &&
                !fields.isEmpty())
}
```

Figure 6. Pseudo code describing the Korat algorithm for test generation from imperative specifications.

the better the pruning. Therefore, the sooner `repOK()` returns `false` (without modifying field values unnecessarily), the better. This pruning mechanism is sound because, if the last accessed field has not been modified, then the output for `repOK()` would not change (assuming that the routine is deterministic), i.e., the parts of the structure visited by `repOK()` would remain the same, and therefore `repOK()` would fail again.

Korat also incorporates a mechanism that avoids generating isomorphic candidates [2]. In this context, two candidates are isomorphic if they only differ in the object identities of their constituents (i.e., if one of the candidates can be obtained from the other by permuting object identities). Since most applications never depend on the actual identities of objects (which represent the memory addresses or heap references of objects), once a structure is generated, all its isomorphic structures may be safely disregarded, since they are redundant (represent already explored cases). Korat avoids generating isomorphic candidates by forcing a canonical representation of explored candidate vectors. More precisely, a lexicographic order between candidate vectors is defined, and then incorporated into the generation process in a way that allows the latter to only generate the smallest element (according to the total order) among all isomorphic candidates. This mechanism works as follows. When considering the range (its possible values) of a class-typed field in the construction of candidates during the search, it is restricted to no more than one untouched (i.e., not previously referenced in the structure) object of its corresponding domain. For example, suppose that in the construction of candidates one needs to consider different values for a given position i in the candidate vector. Now suppose that the i-th position corresponds to a class domain D, and that no fields of said domain have been accessed before i in the last invocation of `repOK()`. The only possible value for the i-th position is 0. More generally, if k objects of domain D have been accessed before in the last invocation of `repOK()`, these must be indexed 0 to $k-1$, and thus the i-th position can range from 0 to k, but may not exceed k. Notice that the canonical representation of candidate vectors depends on how `repOK()` traverses the structure.

In many cases, the above-described backtracking process, pruning and isomorphism-elimination mechanisms allow the Korat tool to significantly reduce the search space when generating test inputs. The efficiency of the tool will greatly depend on how the `repOK()` routine is implemented. The `repOK()` for red-black trees that was (partially) shown in the previous section is, in fact, taken from the Korat distribution: the order in which the different invariant components are checked (structure, then size, then color, then keys) has been manually fine-tuned by the developers of Korat for use with the tool. For instance, for the abovementioned scope, this implementation of `repOK()` allows Korat to find all 12 valid structures after only exploring a mere 200 out of the $905,969,664$ possible candidate vectors. For further details, we refer the reader to [2, 24].

3.2 Test Inputs from Declarative Specifications

When input specifications are provided declaratively, these can be processed to produce inputs by resorting to some form of constraint solving. By solving the input description, we are able to obtain data that satisfy the specification, which can be used as input for the program under analysis. Let us describe a mechanism for test data generation that fits our *bounded exhaustive* test generation scenario, as previously described, and uses SAT solving as a constraint solving approach. This approach was first proposed in [21] with the TestEra tool; we shall refer to it as the *TestEra approach*. Since its introduction, it has been extended and generalized to deal with any testing criterion [1], although in this project we use it to build bounded exhaustive test input suites.

As mentioned above, our process for generating test inputs from declarative specifications is based on SAT solving, which is the process of, given a propositional formula φ, finding a satisfying valuation for φ if one exists, or returning *unsat* if φ is unsatisfiable. Despite the fact that propositional satisfiability is an NP-complete problem, there are many SAT solvers that work very efficiently on large classes of satisfiability problems, and are able to effectively deal with propositional specifications involving millions of variables and clauses. In order to automatically build test inputs from a declarative specification, we translate the specification into a propositional satisfiability problem (which requires imposing a *scope* for the generation, so that the original specification can be "flattened" into a purely propositional specification) in such a way that satisfying valuations of the resulting propositional formula correspond to valid inputs within the provided scope.

```
Algorithm IncTestGeneration {
    Suite = empty;
    rm_val = true;
    while (I–SAT(alpha, rm_val)) {
        val = getValuation();
        Suite = Suite + getTestInput(val);
        rm_val = rm_val and getBlockingClause(val);
    }
}
```

Figure 7. Bounded Exhaustive Generation using incremental SAT Solving.

Since we want to generate the whole space of valid inputs within a given scope, the above process needs to be iterated. For this purpose we rely on *incremental* SAT solving: when a satisfying valuation for a formula φ is found, a new formula α can be added to the constraints in order to forbid the part of the search space that has already been traversed. Thus, subsequent searches for valuations that satisfy $\varphi \wedge \alpha$ will not revisit states that have already been visited.

Let `alpha` be a formula characterizing the state space $\mathbb{S}$ of valid inputs (in our case, the propositional formula obtained from the input specification and the scope). When a satisfying valuation `val` is found by the incremental SAT-solver fed with `alpha`, it determines a test input, captured by the values $v\_1, \ldots, v\_k$ of the primary variables `p_1`, ..., `p_k` in the propositional specification (secondary variables are introduced in order to maintain the size of the formula tractable when translating to CNF, but other than that, do not add any new information concerning the specification). Then, by simply adding an extra clause $\bigvee_{1 \leq i \leq k} \neg(\texttt{p\_i} = \texttt{v\_i})$ to `alpha`, we can guarantee that any new satisfying valuations will differ from the already-produced inputs. Figure 7 illustrates the above-described process. Within the algorithm, `I-SAT`(φ, α) denotes invocation of the incremental SAT-solver on a formula φ, with the added information provided by formula α; `getBlockingClause(val)` corresponds to the previously described clause, whose effect is to remove the last input produced by `val`. Note how these clauses are accumulated in `rm_val`. The algorithm uses incremental SAT-solving, as we mentioned, to avoid starting from scratch each time a new input is queried from the solver. As in the imperative case, we prevent the generation of isomorphic structures by using appropriate, automatically generated symmetry-breaking axioms, as explained in [11], which force a canonical breadth-first labeling of the nodes.

The bounded exhaustive coverage obtained by the algorithm in Fig. 7 is *optimal*, in the sense that only valid inputs within the scope are produced, and each valid input within the scope is produced exactly once.

Besides providing an alternative approach for specifying invariants, declarative specifications allow us to query specifications in order to compute *tight relational bounds* [11] in the context of SAT-based analysis. A tight bound for a field f of an object o is a restriction on the domain of $o.f$, i.e., on the set of possible values that $o.f$ may be assigned, that removes from said domain all cases that can be deemed infeasible based on the specification. Since the possible values of $o.f$ are captured as propositional variables in the encoding of the test generation problem as a boolean satisfiability problem, reducing the domains of object fields implies being able to eliminate variables from the SAT problem, thus increasing its scalability. In this way, tight bounds can be used to preprocess a satisfiability problem, removing infeasible variables from the problem (or, more precisely, replacing these by false) in order to simplify it. For instance, consider our red-black trees example again. Let us now focus on a particular object and field, N_0.*left*. Notice that, according to the scope we previously indicated, there are four possibilities for this field: it can be assigned *null*, N_0, N_1 or N_2. Now suppose that, as part of the declarative invariant, we have a declarative version of `structureOK()` (i.e., capturing the fact that the structure is a tree). We can query the SAT solver about the feasibility of N_0.*left* = *null*, N_0.*left* = N_0, N_0.*left* = N_1 and N_0.*left* = N_2. Notice that each query Q corresponds to asking the SAT solver whether it is possible to build a valid tree structure (with symmetry breaking) in which Q holds. Certainly, N_0.*left* = *null* is satisfiable (think of a tree with a single node, N_0, at the root); N_0.*left* = N_0 is, on the other hand, infeasible, since it violates acyclicity; N_0.*left* = N_1 is again feasible (think of a tree whose root has a nonempty left subtree), and finally N_0.*left* = N_2 is infeasible due to symmetry breaking (any tree, when traversed in breadth-first fashion, will assign N_1 to the left node of the root, which by symmetry breaking can only be N_0). Moreover, observe that as the scope is increased, the number of alternatives for N_0.*left* grows as well, yet its tight bound (for this example) remains the same, so the reduction becomes more profitable. Computing tight bounds essentially consists of performing these queries to simplify the corresponding satisfiability problem. It can be done effectively using a cluster to parallelize the large number of independent queries to the SAT solver. More importantly, it can be computed, stored and reused for many different analyses [11].

4. Generating Test Inputs from Hybrid Specifications

The two mechanisms described in the previous section allow us to generate inputs from either fully imperative or fully declarative input specifications. Let us discuss the problem of generating inputs from *hybrid* specifications, that is, from representation invariants given as a combination of imperative and declarative descriptions. As an example, consider the specification given in Figure 8. In this hybrid specification of red-black trees, the binary tree structure, sortedness and correct size constraints are checked imperatively, whereas the coloring is checked declaratively (assuming the availability of a `hybridSpecEval()` routine that combines imperative and declarative constraints).

```
public boolean repOK() {
  Formula colorsOK = new Formula("
    root != null => color = BLACK and
    all n:TreeSetNode | {
      n in root.*(left+right) - null =>
      (
        (n.left != null => n.left.parent = n) and
        (n.right != null => n.right.parent = n) and
        (n.parent != null =>
        n in n.parent.(left+right)) and
          n !in n.+parent
      )
    }");
  return hybridSpecEval(partialRepOK(), colorsOK);
}

public boolean partialRBTRepOK() {
  //empty tree has size 0
  if (root == null)
    return size == 0;

  //tree structure is ok
  if (!structureOK())
    return false;

  //size is ok
  if (!sizeOK())
    return false;

  //stored values are ordered
  return keysOK();
}
```

Figure 8. Hybrid representation invariant for red-black trees.

At first glance, the generation of inputs from such hybrid specifications may seem like a simple task, made possible by a simple combination of the two techniques presented in the previous section – one would merely need to generate (partial) structures from one of the specifications, using the corresponding generation approach, and complete the obtained structures using the other. However, there are important mismatches that make this combination less direct. In particular, notice that the two partial input specifications refer to the whole structure, i.e., even though they may constrain disjoint portions of the structure, each generation approach will provide data for the portion of the structure that needs to be fixed (in the sense that it cannot be changed) by the other generation mechanism. After applying the first generation mechanism, one may simply check with the generation mechanism applied in the second place whether what was fixed by the first one is correct or not, and backtrack (or continue the search) if not. For instance, one may perform bounded exhaustive generation from the declarative `colorsOK` specification, producing all correctly colored structures, which include those in which keys are not sorted, the size field does not match the actual number of nodes in the structure, etc., and then apply `partialRBTRepOK()` to each of these, discarding those that do not satisfy this imperative constraint. This results in a very ineffective "generate and filter" mechanism, since the first generation will enumerate all possibilities for whatever it is not constraining, and the second will only serve as a filter of what is acceptable and what is not. Therefore, we will avoid following this approach.

To solve the abovementioned problem with the combination of the generation mechanisms, we need to effectively determine which portion of a structure obtained by the first generation mechanism has been fixed by it, i.e., to precisely capture the partial structure that the generation mechanism to be applied in the second place has to consider as rigid. Doing this when the generation from the logical specification is applied first is not direct, and cannot in principle be solved by a simple syntactic analysis, for two reasons: parts of the structure may be indirectly constrained by a logical formula even if these parts are not mentioned in the constraining formula, and (most importantly) the constraining formula may mention parts of the structure and not be willing to constrain them. Constraint `colorsOK` is an example of the latter: it refers to the whole tree structure of a red-black tree through the expression `root.*(left + right)`, yet its purpose is simply to constrain which color gets assigned to each node. On the other hand, when the generation from an imperative specification is applied first, determining which parts have been fixed is straightforward: when executing the imperative specification, the algorithm keeps track of the stack of fields accessed in the process. When a (partially) valid structure is found, the accessed fields tell us exactly which fields were visited to determine the validity, and so we must fix these fields (notice that changing any of the fields may result in the imperative specification changing its verdict on the structure, i.e., what was found to be valid may become invalid and vice versa).

Considering the above observation, our approach for the combination of the processes for solving imperative and declarative specifications works as follows. We first solve the imperative partial specification using the Korat mechanism, and then apply the SAT-based generation to the remaining part of the structure. Let `partialRepOK()` and `declSpec` be the partial imperative and declarative specifications of the inputs of interest, respectively (in our example, these would be `partialRBTRepOK()` and `colorsOK`, respectively). We start using `partialRepOK()` to produce valid structures as explained in the previous section. When a valid structure s is obtained, note that although s is a fully concrete structure, `partialRepOK()` only refers to a part of s. To determine the partial structure that has been "fixed" by `partialRepOK()` we look at the stack of fields accessed during the execution of the routine that led to finding the partially valid structure. Let us call this partial structure s'. We encode s' as an additional constraint for the SAT problem `declSpec`; this forces considering partial structure s' to be "rigid" in the constraint problem. By solving the resulting formula we obtain, as satisfiable valuations, fully complete structures completing what `partialRepOK()` had provided.

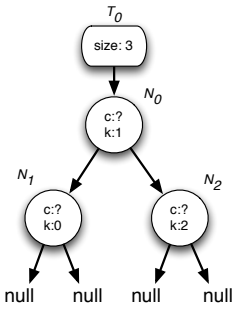

Figure 9. A valid partial structure produced by `partialRBTRepOK()`, which admits multiple colorings.

As an example, let us consider our previous hybrid specification of red-black trees, and assume that, during the execution of `partialRBTRepOK()`, the structure in Figure 9 is produced. Then, based on the fields accessed when executing `partialRBTRepOK()` on this structure (cf. `partialRBTRepOK()` definition), the additional constraint for the SAT problem `colorsOK` would be the following:

```
this = T0 and T0.root = N0 and T0.size = 3  and
N0.parent = null  and   N0.left = N1        and
N0.right = N2     and   N0.key = 1          and
N1.parent = N0    and   N1.left = null      and
N1.right = null   and   N1.key = 0          and
N2.parent = N0    and   N2.left = null      and
N2.right = null   and   N2.key = 2
```

Since only the color attributes of nodes are left free (all other attributes are in the accessed fields stack, and thus fixed as shown by the constraint above), the SAT-based exhaustive search for structures will produce all possible colorings for this structure. In this case, it will return two colorings, namely, one with all nodes colored black, and another with the root colored black and the other two nodes colored red.

Let us turn our attention to the case in which the SAT solver returns *unsat*. In this case we may simply force the imperative process to backtrack, and continue. But we can better profit from the unsatisfiability to determine whether we can further prune the imperative search. To help prune the imperative side, we try to find a substructure of s' that, in combination with `declSpec`, still leads to unsatisfiability. Let $af = [(f_1, v_1), (f_2, v_2), \ldots, (f_k, v_k)]$ be the fields and corresponding values that together form s', given in the order in which they have been visited by `partialRepOK()` (i.e., f_k is the top of the stack of accessed fields). We may start checking for the unsatisfiability of `declSpec` with prefixes of *af* of increasing size, until the first unsatisfiable case is found. That is, we could start checking `declSpec` $\wedge$ [] (no structure fixed), then `declSpec` $\wedge [(f_1, v_1)]$ (only the value of f_1 is fixed to v_1), then `declSpec` $\wedge [(f_1, v_1), (f_2, v_2)]$, and so on, until the first unsatisfiable case if found. Note that we can be certain that the whole structure will be unsatisfiable, since we started with an unsatisfiable case to begin with. If we end up finding that no smaller prefix of *af* is inconsistent with `declSpec`, we simply backtrack the imperative generation as usual. If, on the other hand, we find that a smaller prefix *af'* $= [(f_1, v_1), (f_2, v_2), \ldots, (f_i, v_i)]$ is inconsistent with `declSpec`, we can pop the remaining fields from the stack of accessed fields, i.e. set the stack of accessed fields to *af'*, and continue the "imperative" search from there. The smaller the inconsistent prefix found, the better the pruning on the imperative search. This pruning is sound: we only discard invalid structures. Indeed, note that if *af* changes but *af'* does not, the corresponding structures will continue to be invalid, since *af'* is inconsistent with `declSpec`. Furthermore, by respecting the order in which *af* accessed the structure's fields, we do not interfere with Korat's backtracking mechanism, ensuring that we do not miss unexplored cases (in addition to those identified as invalid).

The pruning mechanism described above is sound, but it may be costly. The reason is that, in the worst case, it could require as many calls to the SAT solver as there are prefixes of the accessed fields stack. Hence, instead of going with this approach, we follow the same principle, but with a significantly more efficient process. Many SAT solvers provide a useful mechanism: when an *unsat* verdict is obtained, they can trace back the *unsat reason*, a (not necessarily minimal) subset of the set of assumptions that explains the cause of the unsatisfiability in terms of assumptions. Basically, we can use the unsat reason to perform the aforementioned process more efficiently. Instead of performing additional queries to the SAT solver, we simply pop elements from the stack of accessed fields until the first element belonging to the unsat reason is found. Since the unsat reason is *conservative* (i.e., it may identify a superset of the minimal set of assumptions leading to unsatisfiability, but never a subset), this more efficient pruning approach is also sound.

Finally, notice that we can also use this approach to help the imperative generation when the Korat process finds an invalid structure. Normally, Korat would just backtrack, and continue the search. If we combine this invalid structure with `declSpec`, the result may be:

- unsatisfiable, i.e., the invalid partial structure s' together with `declSpec` leads to an unsatisfiability from which we can benefit, using the corresponding unsat reason to help pruning the imperative search, or
- satisfiable, in which case we simply let the imperative search continue, without producing the corresponding input, since it subsumes an invalid structure s'.

This is the pruning mechanism that we incorporate in our approach. Let us illustrate it with a concrete example. Consider red-black trees once again, as well as the hybrid input specification given previously, which comprises the imperative `partialRBTRepOK()` and the declarative `colorsOK`. Suppose that the test input generation is being performed for the following scope: up to 6 nodes, size within [0...6], keys

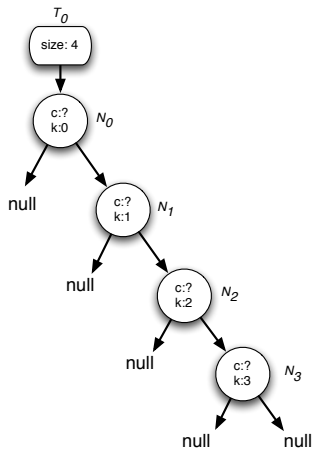

Figure 10. An invalid partial structure produced by `partialRBTRepOK()`.

within [0...5], and 2 colors. Consider the partial structure given in Figure 10, produced by `partialRBTRepOK()` as a valid partial structure. When attempting to color this structure, the SAT solver will fail, informing us that the satisfiability problem (corresponding to the coloring) is infeasible, i.e., `colorsOK` is inconsistent with the current partially valid structure. Due to the visit that `structureOK()` performs, the stack of accessed fields for the current partial structure is the following:

$$[T_0.root, N_0.left, N_0.parent, N_0.right,\\ N_1.left, N_1.parent, N_1.right,\\ N_2.left, N_2.parent, N_2.right,\\ N_3.left, N_3.parent, N_3.right, T_0.size,\\ N_0.key, N_1.key, N_2.key, N_3.key]$$

To continue the input generation, a standard Korat backtracking will attempt first to provide alternative key values until these are exhausted (many of which will be valid key assignments), then different sizes will be tried until these are exhausted, and so on, until eventually it will start changing $N_3.right$; given the scope, the changes to the latter field will lead to many additional attempts to extend the tree to the right. Moreover, the number of attempted invalid structures increases as backtracking progresses, since after exhausting the extensions of $N_3.right$, all extensions of $N_3.left$ will be combined with extensions of $N_3.right$, and so on.

However, when analyzing the intersection between the unsat reason and the accessed fields, our approach is able to pop various fields, and obtain:

$$[T_0.root, N_0.left, N_0.parent, N_0.right,\\ N_1.left, N_1.parent, N_1.right]$$

Essentially, the unsat reason indicates that as long as we do not change $N_1.right$ (taking into account that $N_0.left = null$, which is deeper in the stack), the coloring of the structure remains infeasible. We can then safely pop fields

$$N_2.left, N_2.parent, N_2.right,\\ N_3.left, N_3.parent, N_3.right, T_0.size,\\ N_0.key, N_1.key, N_2.key, N_3.key$$

from the stack of accessed fields before continuing the search. Next, since $N_1.right$ is already at its highest possible value (due to Korat's symmetry-breaking approach), the backtracking will continue checking extensions to $N_1.left$ (which will fail) until a change in $N0.left$ is forced. This pruning has an important impact on test input generation.

The resulting algorithm, which combines generation from the imperative part of the invariant `partialRepOK()` with generation from the declarative part of the invariant `declSpec`, and performs unsat-reason-based pruning, is shown in Figure 11. Let us ignore, for the time being, the use of tight bounds (line 2, and the uses of `bounds` in lines 10, 18, 23 and 35). Note that the algorithm essentially maintains Korat's structure (i.e., the imperative search drives the process), yet as opposed to Korat, when the imperative invariant, which is in this case partial, succeeds in finding a valid candidate (lines 8-21), the SAT-based generation completes the found partial structure via exhaustive enumeration based on the declarative specification (lines 9-20), by following the `IncTestGeneration` approach; observe how the partial structure built by `partialRepOK()` is fixed before starting the SAT-based generation (line 9 and its use in lines 10 and 11). Also, as we explained earlier, when the SAT-based generation is exhausted, or the partial structure found by `partialRepOK()` is invalid (`partialRepOK()` returned false) or infeasible (unsatisfiable when combined with `declSpec`), fields in the stack of accessed fields are popped until the first field in the unsat reason is found (lines 17-20 and 22-28), prior to advancing to the next candidate in the Korat fashion (lines 29-36).

Let us now move on to discussing tight bounds. If we have part of the invariant specified declaratively, it can be used to compute tight bounds. Since these would be computed from a partial specification, they might not be the tightest, but they are certainly valid: whatever is determined to be infeasible from the partial specification will remain infeasible for the whole specification (i.e., when combined with the imperative generation side). Therefore, the tight bounds that we automatically compute from the declarative part are also valid for the imperative side, and thus can be used to restrict the domains of fields in order to improve the search. These tight bounds will provide greater profits when at least part of the invariant is provided *both* declaratively and imperatively, since bounds in such cases will definitely restrict the domains of fields over which the imperative side will iterate. This is a point in favor of having this kind of redundancy in specification, if possible.

In HyTeK, tight bounds are employed in two ways. First, they are incorporated into SAT queries involving the hybrid specification in order to improve SAT solving (lines 10, 11,

```
1  Algorithm HyTeKGen() {
2      Bounds bounds = computeBounds(declSpec);
3      Vector curr = initVector;
4      Stack fields = new Stack();
5      boolean ok;
6      do {
7          (ok, fields) = curr.partialRepOK();
8          if (ok) {
9              rm_val = getPartialValuation(curr, fields);
10             if (SAT(declSpec && bounds && rm_val)) {
11                 while (I-SAT(declSpec && bounds, rm_val)) {
12                     val = getValuation();
13                     reportValid(getTestInput(val));
14                     rm_val = rm_val && getBlockingClause(val);
15                 }
16             }
17             else {
18                 unsatReasonFields = getFields(unsatReason(declSpec && bounds && rm_val));
19                 while (!unsatReasonFields.contains(fields.top())) fields.pop();
20             }
21         }
22         else {
23             rm_val = getPartialValuation(curr, fields);
24             if (!SAT(declSpec && bounds && rm_val)) {
25                 unsatReasonFields = getFields(unsatReason(declSpec && bounds && rm_val));
26                 while (!unsatReasonFields.contains(fields.top())) fields.pop();
27             }
28         }
29         repeat {
30             field = fields.pop();
31             while (!fields.isEmpty() && curr[field] >= nonIsoMax(curr, fields, field)) {
32                 curr[field] = 0;
33                 field = fields.pop();
34             }
35             if (!fields.isEmpty()) curr[field]++;
36         } until (fields.isEmpty() || isCompatible(curr,bounds))
37     } while (curr != lastVector && !fields.isEmpty())
38 }
```

Figure 11. Pseudo code describing HyTeK's approach to generating inputs from hybrid specifications.

18, 24 and 25 of Figure 11), since they lead to removing infeasible propositional variables, as we explained before. The satisfiable and unsatisfiable instances of the specification are exactly the same with and without tight bounds, so the algorithm's behavior is not altered by their use – its SAT queries are just made more efficient.

Second, tight bounds are used to restrict field domains when calculating the next candidate to try on the imperative side (see line 36 in Figure 11).

As an example of the latter use of tight bounds, let us once again consider the structure in Figure 9, with the imperative partial specification `partialRBTRepOK()` and the partial declarative specification `colorsOK` and `treeStructureOK` (notice the redundancy of the tree structure specification, provided both declaratively and imperatively in this case). Now, after enumerating all possible colorings for structure in Figure 9, the next partial candidate has to be computed.

Assuming that the scopes allow up to 1 `TreeSet` object, up to 3 `Node` objects, range $[0\ldots3]$ for `TreeSet.size` and range $[0\ldots2]$ for `Node.key`, then, if the stack of accessed fields is:

$$[T_0.root, N_0.left, N_0.parent, N_0.right,\\ N_1.left, N_1.parent, N_1.right,\\ N_2.left, N_2.parent, N_2.right, T_0.size,\\ N_0.key, N_1.key, N_2.key]$$

since no other valid key and size assignments are possible for this tree (according to the scope), we will eventually reach the following stack (and the same structure):

$$[T_0.root, N_0.left, N_0.parent, N_0.right,\\ N_1.left, N_1.parent, N_1.right,\\ N_2.left, N_2.parent, N_2.right]$$

Now the process should attempt different assignments to $N_2.right$, namely, N_0, N_1 and N_2 ($null$ is the current assignment). In this case, due to tight bounds computed for tree structures of up to 3 nodes satisfying symmetry breaking, the only possible right child for N_2 is $null$, and therefore the process will be forced to backtrack, without attempting any other case for $N_2.right$ (all cases will be "advanced" in loop 29-36).

There is a subtle technical issue related to the use of tight bounds from the declarative part for pruning the imperative search; it has to do with the fact that tight bounds, since they are computed from the declarative invariant, will assume a breadth-first traversal of the structure, whereas the imperative side will label nodes according to the order in which these are visited by `partialRepOK()`. This mismatch can be solved by maintaining sets of labels corresponding to the tight bounds, associated with the nodes obtained in the `repOK()` traversal, and checking bound feasibility through label intersection, as explained in [13].

5. Optimizing Test Input Generation

Bounded exhaustive analyses are becoming more popular, especially in automated verification and test generation contexts, as various tools based on bounded exhaustive approaches (e.g., Korat [2], UDITA [14], Alloy [16], TestEra [20], Forge [7], Symbolic PathFinder [32], to name a few) demonstrate. In these contexts, it is typically the case that the developer wants to perform analyses on *increasingly large scopes*. Indeed, as the scope is increased, bounded verification is able to find errors that were not detectable at smaller scopes, and bounded test generation is able to produce more interesting and complex tests (or test inputs) whose generation was infeasible with smaller scopes. Thus, incrementing the scope in bounded exhaustive analyses has desirable effects on bug finding, test generation and coverage.

Increasing the scope, however, typically has undesirable effects on efficiency. While a particular analysis may be efficient for small scopes, as scope is increased, the running times for bounded exhaustive analyses inherently grow very quickly, making the analyses infeasible. So one has to deal with contradictory concerns – increasing scopes is needed to improve analysis impact, while decreasing scopes is necessary in order to keep analysis times affordable.

In many cases, one cannot be satisfied with small scopes, since the program under analysis may require larger scopes to exercise particular portions of the code. For instance, if we are generating tests for a routine that manipulates balanced trees, e.g., insertion on red-black trees, then one would need sufficiently large scopes to force some rebalancing and rotations. If targeting a particular scope is a necessity, and cannot be directly achieved due to efficiency reasons, one has to start considering different variables that may make the analysis more efficient. For instance, in Korat, SPF, UDITA and other tools, and in the approach presented in this paper, the analysis is highly sensitive to the way in which `repOK()` is implemented, since the backtracking over candidate structures depends directly on how this routine visits structures. So, a possible way of making the analysis more efficient is to "play" with `repOK()`, looking for variants that may affect (in many cases substantially) the efficiency of the exploration of candidate structures, and therefore the test generation process as a whole. A way of considering variants of this routine is by checking different aspects or parts of the input specification (i.e., valid structural organization, size, sortedness, balance, etc.) in different orders. For instance, there may be a huge difference in the number of explored candidates (and the efficiency of the generation process) if one checks structural organization, sortedness and balance –in that order– compared to first checking structural organization, then balance and finally sortedness. In our case, since we allow for hybrid input specifications, an additional variable is which part of the input specification is to be left imperative, and which part is to be left declarative, assuming, of course, that there is some degree of redundancy in this respect (i.e., that at least some part of the input specification is available both declaratively and imperatively). Anyone familiar with a bounded exhaustive tool knows that choosing the appropriate combination is usually far from straightforward, and although experience may dictate some general strategy, one typically has to experiment with several alternatives before finding the right "form" of the input specification with respect to generation efficiency.

Since our approach for test input generation is bounded exhaustive, it suffers from this problem as well. We deal with the problem by resorting to *transcoping*. This technique, put forward in [27] for the parallel analysis of Alloy models, originally consisted of examining alternative partitions of a problem for small scopes, and extrapolating this information to select an adequate partition for larger scopes. In our present context, we will use this approach towards two goals:

- to select the best ordering among the parts of an imperative input specification, and
- if some parts of the input specification are given both declaratively and imperatively, to determine the most convenient setting in which each of these are to be solved, i.e., the most appropriate way of partitioning the specification into (disjoint) declarative and imperative portions.

If a specification is given entirely imperatively, then our approach will attempt to find the optimal ordering of the components of the imperative `repOK()`. If the input specification is given partly operationally and partly declaratively, with no intersection, our approach will search for an optimal ordering of the imperative part. Finally, if the input specification is given partly operationally and partly declaratively, but with some intersection, meaning that at least one aspect of the input specification is given both imperatively and declaratively, our approach will decide: which parts of this intersection are better solved imperatively, which parts are better solved declaratively, and for the imperative part, what the optimal ordering is. The decision is made by trying all alternatives on small scopes, where test input generation times are negligible, and extrapolating the results to larger scopes, while progressively narrowing down the number of candidates in order to keep the total cost reasonable.

As an example, consider a representation invariant for red-black trees that consists of the following 6 parts:

1. If root is null, size is zero.
2. If root is not null, its color is black.
3. The structure reachable from the root is a binary tree.
4. The size field correctly captures the number of nodes in the structure.
5. The coloring is valid (no two consecutive red nodes in any path, same number of black nodes on all paths).
6. The keys are correctly sorted (the tree is a search tree).

For this case, let us suppose that imperative versions of constraints 1–5 are available, while constraints 5 and 6 are available declaratively (notice the redundancy that exists due to the imperative and declarative versions of constraint 5). Then we have 5! possibilities corresponding to the cases in which the first five portions are solved imperatively, plus 4! possibilities corresponding to the cases in which the last 2 are solved declaratively (or, equivalently, the first 4 are solved imperatively), for a total of 144 hybrid invariants.

Of course, one might envision many different approaches to transcope information. Essentially, the main steps along the transcoping process are:

- Choosing scopes to be used for the initial assessment, which will then be extrapolated to larger scopes.
- Choosing the criterion for the extrapolation, e.g., running time, number of produced candidates, etc.
- Selecting some fraction or subset of the best-performing configurations to be promoted to the next scope.

Choosing values for these parameters is not necessarily straightforward; careless choices could affect the appropriateness of the extrapolation and the performance of the whole approach. To continue with the above example, suppose that we use scope 3 to run the initial assessment of alternatives, that we choose running time as the promotion criterion and that we select only the best-performing 10% (that is, the fastest 10%) of the invariants to be promoted to the next scope. This means that we run the test input generation process described in the previous section for the 144 invariants for scope 3, sort them by their running times, then select the fastest 14 candidates for scope 4, and so on.

As an additional example, our empirical evaluation (more details of which will be explained later on) uses input generation time as the promotion criterion. It starts at scope 1 and keeps all candidates until reaching the first scope where the difference between the fastest and slowest candidates ceases to be negligible, promotes the top $\frac{1}{3}$ of the candidates to the next scope, and keeps iterating until a single-digit number of candidates remain. This is more than what one would be willing to run in a concrete analysis scenario, but we do run these additional experiments for evaluation purposes (of the general efficiency of the approach, and that of transcoping in particular).

6. Evaluation

In previous sections, we proposed the use of hybrid input specifications; we introduced a technique for bounded exhaustive input generation from such specifications, and we presented an approach for automatically tailoring these specifications for the sake of efficiency. In this section, we perform an empirical evaluation of these proposals. We shall focus on the following research questions, associated with hybrid specifications and our proposed techniques:

Q_1) Can the generation from hybrid specifications perform better on typical data structures than state-of-the-art fully imperative and fully declarative techniques?

Q_2) Can the right combination of imperative and declarative parts of a hybrid specification be determined effectively by means of transcoping?

Our assessments to answer the above questions involve two data structures: red-black trees and AVL trees. The red-black tree implementation is based on the core of the implementation of class TreeSet in package java.util.Collections, well-known for being representative of a class of structures that challenge the efficacy of test generation tools. The AVL implementation was taken from class TreeList in package apache.commons.collections4, and its invariant is somewhat less stringent than that of red-black trees. Both data structures are present in most benchmarks for bounded exhaustive test generation.

6.1 Experimental Setup

All experiments were run on an Intel Core i5-750 processor running at 2.67 GHz with 8 GB of 1,333 MHz DDR3 main memory, running Debian GNU/Linux. Since our prototype tool is developed in Java, it runs on a JVM, for which we set a 4 GB heap usage limit. Java version 1.7.0 (OpenJDK 64-Bit Server VM) was used to run all experiments. In all cases where a SAT-solver was needed, we used Minisat 2.2.0 [9].

Scopes in this Section are referred to as a single number. Scope n means, for both case studies, up to n nodes, keys in the range $0 \ldots (n-1)$ and size field in the range $0 \ldots n$.

6.2 Experimental Design

Let us now describe the design of our experimental evaluation. Both of the data structures that we analyze feature representation invariants that are expressed as a series of constraints; essentially, the invariant is the conjunction of said constraints. The 6 constraints of the invariant for red-black trees are those numbered 1–6 in Section 5, and are taken from the invariant of this data structure in the Korat distribution. The five constraints that compose the representation invariant for AVL trees are the following:

1. If root is null, size is 0.

2. The structure reachable from the root is acyclic.
3. The keys are correctly sorted.
4. The size matches the number of nodes in the structure.
5. The tree structure is balanced.

We assume that all constraints of both invariants are available both declaratively and imperatively, i.e., that we have full redundancy in this regard. In other words, for each data structure, each constraint enforced by the procedural specification has a corresponding (equivalent) constraint in the declarative specification, and vice versa. Notice that this implies having to decide on which side (imperative or declarative) each constraint should be solved.

For a given invariant expressed in this way, its constraints can be represented by their indices, and hybrid invariants as tuples with a sequence (for the imperative portion) and a set (for the declarative portion) of said indices. As pointed out previously, the order in which the constraints are solved is irrelevant on the declarative side, whereas on the imperative side it is not; hence the use of sequences (resp. sets) to represent the imperative (resp. declarative) portion of a hybrid invariant. More precisely, let $C_{rbt} = \{1, 2, 3, 4, 5, 6\}$ and $C_{avl} = \{1, 2, 3, 4, 5\}$ be the sets of constraints characterizing the invariants of red-black trees and AVL trees, respectively. For each set C_x, given any permutation P of any subset of C_x, we can build a hybrid specification H whose imperative part includes the constraints in P in the order in which they appear in P, and whose declarative part contains those constraints from C_x that are not in P. For instance, if P is sequence $\langle 2, 1, 5\rangle$, taken from the red-black tree constraints, then H must be $\{3, 4, 6\}$. Notice that, for a given structure, the declarative part of a hybrid invariant is implied by the the imperative portion of the invariant (it is its complement), so that henceforth, when referring to a particular hybrid invariant, we shall simply refer to the corresponding imperative portion (i.e., its list of indices, such as $\langle 2, 1, 5\rangle$).

For our experiments, we assembled *all* the hybrid invariant combinations that can be built in the manner just described. In particular, notice that due to the full redundancy in the invariants, the assembled hybrid invariants include fully imperative and fully declarative ones, and for the case of imperative invariants, all permutations of constraints are included, too (i.e., all possible hybrid invariants buildable with the available constraints are considered). There are 1,957 and 326 different hybrid invariants for red-black trees and for AVLs, respectively. Of course, for each data structure, all those invariants are equivalent: each of them characterizes exactly the same set of valid structures. But the running times for test input generation using each of them can differ significantly, either because some constraints are solved on different sides (declarative or imperative), or because the order of constraints on the imperative side is different. Although taking into account all possible hybrid predicates, due to the full redundancy in specification, makes the transcoping phase much harder (since many alternatives have to be considered), it also enables us to evaluate how different partitions of the specification into declarative and imperative portions, and orderings of the latter, may lead to different performances.

For each analyzed data structure, we carried out bounded exhaustive input generation for *all* of its different hybrid invariants, for scopes 1 to 6 (in the case of red-black trees), and for scopes 1 to 9 (in the case of AVL trees), and we recorded the corresponding running times. We could afford to run all possible invariants for a few more scopes in the case of AVLs due to both the number of different hybrid invariants and the average running times being somewhat smaller than for red-black trees (probably because the latter invariant is more complex). Note that, since fully imperative and fully declarative invariants are among the possible hybrid invariants, running times for these candidates (associated with using solely Korat-like generation, and solely TestEra-like generation, respectively) are also recorded as part of the experiments.

Due to the number of different hybrid invariant combinations and the increasing computational cost per unit as scope is increased, we ruled out exhaustively running all combinations for scopes larger than 6 (in the case of red-black trees) or larger than 9 (in the case of AVL trees). Similarly, scopes smaller than 3 (in the case of red-black trees) or smaller than 6 (in the case of AVL trees) involve running times that are too short and volatile to provide useful transcoping information (either because total running time is too small a fraction of a second or the max-min ratio is too close to 1).

We promote to the next scope those hybrid predicates that constitute the best third of the experiments (i.e., the fastest 33% when sorted by total running time for test input generation). We also keep the fully imperative and fully declarative invariants, regardless of whether or not they remain in the top third, for control purposes. We stop refining the short list (that is, we start running all remaining candidates in all larger scopes) once the number of candidates falls below 10.

6.3 Experimental Results

When running all possible hybrid invariants (which, as we explained, subsume the fully imperative and fully declarative invariants), given a fixed scope, groups of hybrid specifications (for which generation can be done efficiently and for which generation takes significantly longer) can be clearly distinguished. This can be observed in Figures 12 and 13, which plot all the experiments by scope in logarithmic scale. Scopes 3–6 and 6–9 are shown for red-black trees and for AVL trees, respectively (some of the smaller scopes are omitted from the charts when they would yield a flat line). Notice how, in both cases and despite the difference in test input generation times, the curves have a very similar shape. At this point, we can start answering our research questions. Indeed, the best hybrid specifications in scopes 3 to 6, for red-black trees, and scopes 6 to 9, for AVLs, outperform all fully imperative and fully declarative specifications, in

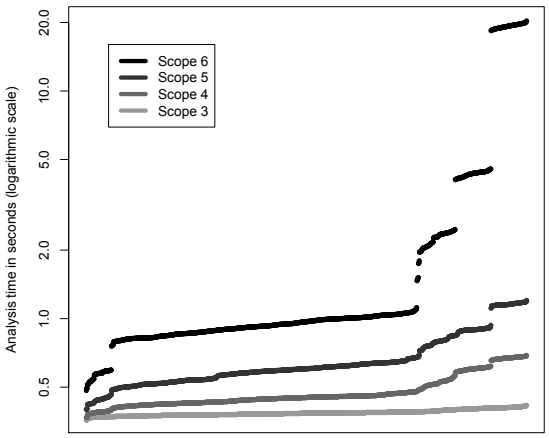

Figure 12. Analysis time, all red-black tree hybrid predicates, scopes 3–6.

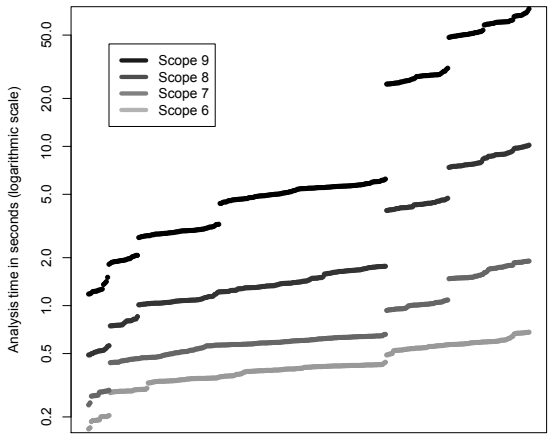

Figure 13. Analysis time, all AVL hybrid predicates, scopes 6–9.

running times for bounded exhaustive generation. Moreover, the top 33% best hybrid specifications do not contain fully imperative invariants, for either of the case studies, at any scope. The fully declarative invariant does appear in some of the top thirds of both case studies, but is not among the final short list in either of the case studies.

Also note how the noticeable differences in the performance of hybrid specifications as well as the visible preservation of the curves' shape in the graphs seem to provide initial evidence that transcoping indeed works as expected, since the worst specifications continue to worsen as scope increases, while the fastest specifications also keep performing well as scope is increased.

When taking the best 33% and promoting those candidates to the next scope, from scope 3 (resp. 6) in the case of red-black trees (resp. AVLs) onwards we consistently retain hybrid predicates that scale better for test input generation; notice that up to scope 6 (resp. 9) in the case of red-black trees (resp. AVLs), we have the running times for *all* the possible hybrid predicates, so that scalability is measured with respect to the whole population of hybrid specifications. Figure 14 and Figure 15 illustrate the transcoping process for red-black trees and for AVL trees, respectively.

On the other hand, it is also important to note that the candidates that were identified as bad at smaller scopes (for instance, the 5 hybrid predicates $\langle 1, 4, 2, 6, 3\rangle$, $\langle 4, 6, 2, 1, 3, 5\rangle$, $\langle 4, 1, 2, 6, 3, 5\rangle$, $\langle 4, 6, 3, 1, 2, 5\rangle$ and $\langle 1, 4, 2, 6, 3, 5\rangle$ for red-black trees) maintained their condition when transcoping: test input generation exceeded 5 hours as early as scope 8 for all these specifications. Considering that the set of 8 predicates selected through transcoping can generate equivalent tests at that scope in 2 seconds, the predicates identified as best exhibit a speedup of at least 9,000X with respect to the bad candidates. These results show that transcoping indeed allowed us to make well-informed decisions, answering Q_2.

Let us return to research question Q_1 – whether hybrid predicates can lead to better test generation times with respect to non-hybrid (fully imperative and fully declarative) specifications. Table 1 compares test input generation times using the final, single-digit set of hybrid predicates selected by transcoping (which are truly hybrid, with nonempty declarative and imperative portions) with test input generation times using Korat and the SAT-based test input generation approach described in Section 3.2, on red-black trees. Korat is executed on the fully imperative invariant for red-black trees that shows the best performance, which corresponds to the sequence of constraints $\langle 1, 2, 3, 4, 5, 6\rangle$, and coincides with the way this invariant is structured in the case study accompanying Korat's distribution. The SAT-based test generation approach uses a fully declarative specification of red-black trees that has been improved in previous work to optimally exploit SAT solving for bounded analysis [11]. Table 1 reports, for each scope between 3 and 15 and for red-black trees, the following information:

- The average test input generation time required by the 8 best hybrid invariants identified by transcoping ($\langle 1, 3\rangle$, $\langle 3, 1\rangle$, $\langle 1, 2, 3\rangle$, $\langle 1, 3, 4\rangle$, $\langle 2, 3, 1, 4\rangle$, $\langle 2, 1, 3\rangle$, $\langle 2, 3, 1\rangle$, $\langle 2, 3, 4, 1\rangle$), all of which are truly hybrid.
- The time required by Korat on the purely imperative invariant $\langle 1, 2, 3, 4, 5, 6\rangle$.
- The time required by the SAT-based approach (in the style of TestEra) on the fully declarative invariant.
- The speed-ups obtained with respect to each of the latter.

Although it is not possible to determine the precise speedups for scopes where timeouts occur, we can certainly guarantee

	H	K	TE
Scope 3	0.37	0.36	0.75
speed up		1X	2X
Scope 4	0.38	0.39	0.91
speed up		1X	2X
Scope 5	0.42	0.49	1.27
speed up		1X	3X
Scope 6	0.52	0.61	1.73
speed up		1X	3X
Scope 7	0.83	0.91	2.18
speed up		1X	2X
Scope 8	1.86	2.05	4.54
speed up		1X	2X
Scope 9	5.61	7.28	13.15
speed up		1X	2X
Scope 10	14.46	34.72	43.02
speed up		2X	3X
Scope 11	23.73	176.16	147.06
speed up		7X	6X
Scope 12	37.41	959.39	504.84
speed up		25X	13X
Scope 13	104.46	5,616.48	1,915.75
speed up		53X	18X
Scope 14	300.07	TO	7,933.30
speed up		>60X	26X
Scope 15	1,194.79	TO	TO
speed up		>15X	>15X

Table 1. Comparison of HyTeK (H) with Korat (K) and TestEra (TE), for red-black trees. HyTeK considers average of the 8 best hybrid invariants found via transcoping. Times are reported in seconds. The timeout (TO) is set at 5 hours.

that the actual speedup exceeds the ratio between the timeout and the time required by HyTeK.

In summary, our experiments allowed us to answer both of our research questions affirmatively. For both of our case studies, bounded exhaustive test generation from hybrid specifications, using the technique presented in previous sections, was more efficient than state-of-the-art techniques limited to fully imperative and to fully declarative specifications, respectively. Transcoping, at least for our case studies, proved to be an effective mechanism to identify the right hybrid specifications, i.e., to decide which parts to express declaratively and which ones to express imperatively (and, for the latter, what ordering to use) in order to improve efficiency and maximize scalability of test input generation.

6.4 Implementation Details

Our prototype tool is implemented on top of the standard Korat distribution. The prototype contributes to the Korat codebase as a set of additions, while trying to keep modifications to a minimum. By adjusting command-line options, the user can choose whether to run the HyTeK prototype or the original Korat tool (more details can be found in the README.HyTeK.txt file).

In addition to the Java implementation, the following are included in the current HyTeK distribution [35]:

	H	K	TE
Scope 7	0.27	1.63	1.37
speed up		6X	5X
Scope 8	0.49	9.68	2.51
speed up		19X	5X
Scope 9	1.20	72.94	4.56
speed up		61X	4X
Scope 10	3.40	544.42	10.41
speed up		160X	3X
Scope 11	10.38	4,060.85	26.64
speed up		391X	3X
Scope 12	42.12	TO	92.67
speed up		>427X	2X
Scope 13	133.16	TO	290.38
speed up		>135X	2X
Scope 14	435.91	TO	979.33
speed up		>41X	2X
Scope 15	1,562.05	TO	3,505.84
speed up		>12X	2X

Table 2. Comparison of HyTeK (labeled H), with Korat (K) and TestEra (TE), for AVL trees. HyTeK considers average of the 4 best hybrid invariants found via transcoping. Times are reported in seconds. The timeout (TO) is set at 5 hours.

- Already-generated support files, including all the .java, .bounds, .pvars, .als and .cnf files needed to reproduce all experiments.
- Two scripts: one to automate the creation of support files for all combinations of imperative and declarative parts, and another one to convert previously computed tight bounds from the format used by [11] to the format used by the HyTeK prototype. (These scripts are not needed to reproduce the experiments, since all support files are already provided with the distribution. They are only needed to add new test cases or larger scopes.)
- A command-line version of the Alloy Analyzer [36] with batch model enumeration capabilities (needed to generate new test cases, and to reproduce TestEra-like behavior).
- The C++ source code for the solver used by HyTeK via Java Native Interface (Minisat 2.2.0 plus JNI wrappers) and a precompiled 64-bit ELF Linux binary thereof.

6.5 Threats to Validity

The first possible threat to the validity of our experiments is the selection (and the number) of case studies. We deal with complex data structures because the technique presented in this paper is meant to improve bounded exhaustive test generation, and this testing approach is known for being particularly well-suited for code that handles complex heap-allocated data structures. Our experimental evaluation is based only on two case studies of this kind. These were carefully selected for fairness in the comparison with other generation techniques, and to correctly assess the viability of transcoping. The red-black tree data structure has been identified by previous research [2] as one of the most complex and hardest to deal with, from the point of view of in-

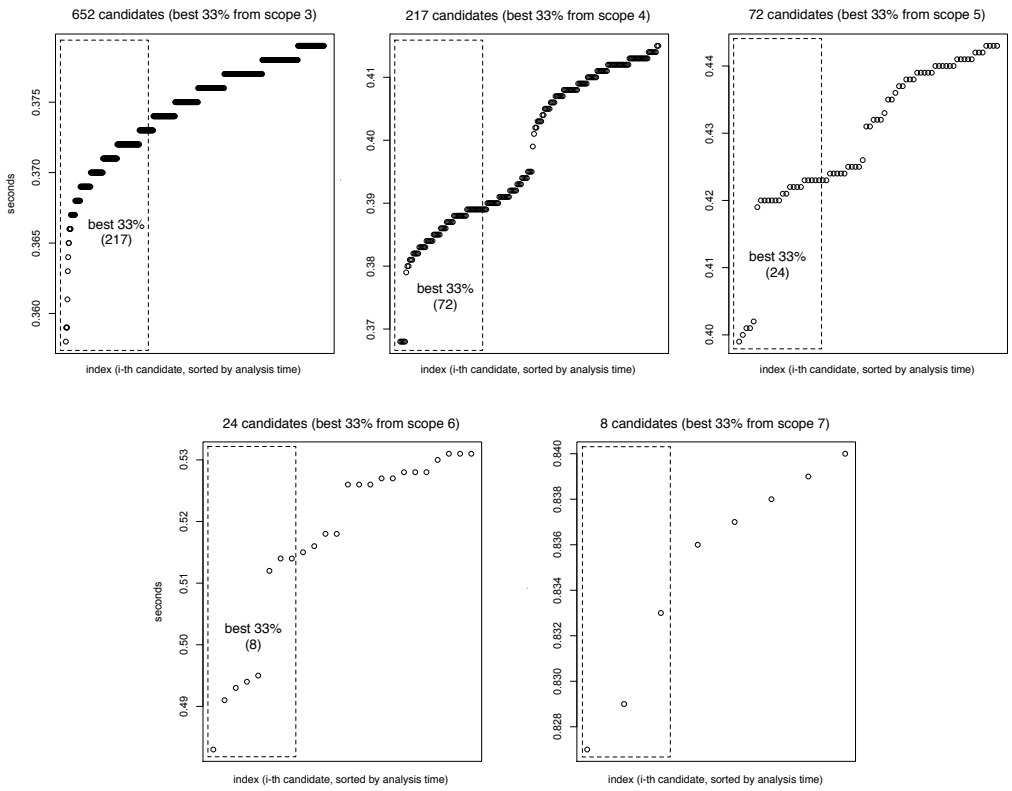

Figure 14. An illustration of the transcoping process for red-black trees.

put test generation. The AVL data structure, despite being somewhat simpler, has also been extensively studied and is found in most benchmarks related to test generation. Both data structures were selected because of the complexity of their invariants, made out of 6 and 5 different portions (being more complex, and therefore more difficult, than other data structures commonly used as benchmarks for this kind of analysis). These complex invariants lead to a high number of alternative configurations both for non-hybrid and hybrid invariants. This poses a more subtle problem when having to select the right order in which to solve them (in the fully imperative case), and also serves as a stress test for the transcoping process presented in this paper. In particular, for simpler invariants (such as, for instance, that of singly linked lists: merely acyclicity and the size matching the number of nodes), the different alternative configurations are just a few, making the problem of correctly selecting the right order for generation a much easier one.

In our comparison with existing techniques, the structure of our invariants may be biased towards our analysis technique. To avoid this problem and make the comparison fair, we took these specifications from case studies of other tools. The red-black tree invariant is taken directly from the Korat distribution, and hence has been manually tailored by experienced users (the developers of the tool) to exploit the generation technique at its best. The AVL tree invariant is based on a specification that is part of the Roops benchmark[1].

Another threat to the validity of our results is the selection of the other tools for comparison. Instead of Korat, other tools such as SPF or UDITA could have been chosen for comparison; however, these do not offer alternative mechanisms for generation from imperative invariants. Moreover, Korat has been recognized as the most efficient amongst a set of similar tools for bounded exhaustive generation [29]. In the context of SAT-based bounded exhaustive generation, there are fewer tools to compare with. Essentially, we can either compare with TestEra, or with the tool introduced in [11]. The latter is our choice because it is more efficient due to the use of tight bounds, as shown in [11]. Therefore, from the point of view of efficiency, the comparison is fairer than if it had been done against TestEra.

An (internal) threat to the validity of the results is the correctness of our tool. Although we did not prove our implementation to be correct, we compared the number of structures obtained for the case studies in all cases. We consistently obtained exactly the same number of structures, for all analyzed hybrid invariants and all scopes in our experi-

[1] http://code.google.com/p/roops/

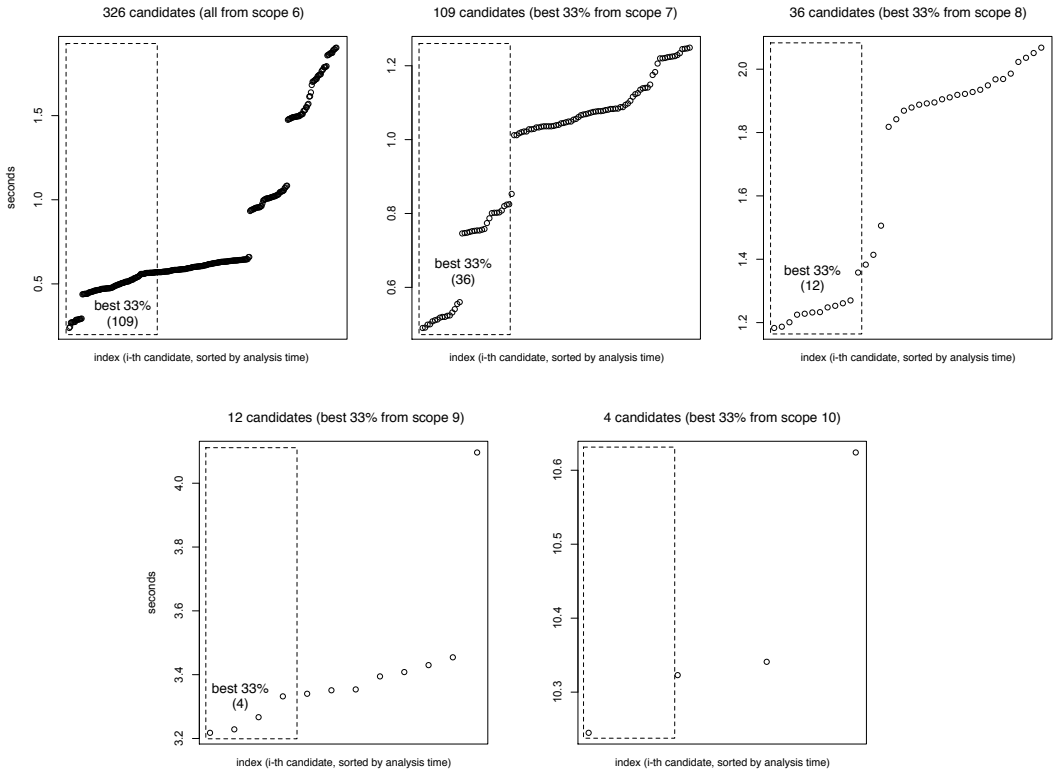

Figure 15. An illustration of the transcoping process for AVL trees.

ments, which also coincide with the corresponding number of structures both using standard Korat and TestEra. While this does not constitute proof of correctness, it is concrete evidence that we are not excessively pruning the search (we do not discard valid cases), nor solving simpler constraints (we do not generate, in any of the cases, invalid structures that other tools and techniques would discard).

As an attempt to ensure the reliability of our results, we have rerun our prototype tool several times (always on dedicated machines), consistently obtaining results nearly identical to those reported in this article.

7. Related Work

Our work is concerned with test input generation. In this regard, it is related to many approaches aimed at tackling the same problem, including random generation approaches, and more closely, generation based on constraint solving. Since in this article we target bounded exhaustive test generation (although the techniques presented in it can be generalized to other automated test generation approaches), our approach is particularly related to other tools dealing with this kind of testing. Korat [2], FAJITA [1], TestEra [21], and UDITA [14] are tools targeting this domain. No other tool for automated test generation (bounded exhaustive or any other kind) supports hybrid input specifications. Moreover, most tools only support one generation approach, with no automated aid for tailoring specifications for improving generation. UDITA offers more flexibility, by allowing different kinds of generation approaches (e.g., based on input factories, or on structure generation and filtering), and even allows one to combine these approaches. However, like the other tools mentioned, it does not tailor specifications automatically. This task is largely manual, and is based on the experience of users with different tools. To the best of our knowledge, this is the first test generation approach that automatically manipulates input specifications to improve generation.

The SPIN [15] model checker introduced the idea of combining an imperative language with a model checker's input language. SPIN allows writing models in its input language Promela and supports insertion of C code into the models. In contrast with Alloy, which provides a declarative logic backed by SAT technology, SPIN provides traditional stateful model checking with a focus on checking temporal logic properties. Moreover, SPIN does not support the use of different solvers to solve separate kinds of constraints (which is our focus in this article).

Chang and Jackson [4] add support for temporal logic to Alloy by translating the ensuing formulas into a BDD-based representation, thereby enhancing Alloy with some operators from temporal logic, albeit not with imperative constructs.

In the context of Alloy's integration with imperative code, the Alloy tool-set has served as an execution engine for (partial) Java programs using their Alloy specifications [25, 28, 34]. These projects focus on the problem of *executing* specifications using Alloy's SAT-based backend to update states of imperative programs, and not the problem of how to efficiently solve logical constraints written in a mixed notation using a combination of solvers, as done in our work.

Similar in spirit to these projects is more recent work by Koksal et al. which presented Kaplan [22], an extension to the Scala programming language to support constraint programming. The purpose of our approach is different from Kaplan since we focus on writing and solving logical constraints using a hybrid approach, whereas Kaplan provides a general purpose integration of constraint programming into a stateful language. Moreover, we show how to integrate a constraint solver for first-order logic with a solver for Java predicates. In contrast, Kaplan uses the SMT solver Z3 [6] as the underlying enabling technology for constraint programming.

Uzuncaova's doctoral work [30, 31] introduced incremental solving for Alloy models, where a solution to one formula is fed as a partial solution to efficiently solve another formula, and applied incremental solving in the context of test input generation for software product lines.

More recently, Ganov et al. [12] introduced *annotations* for Alloy models to guide solving of Alloy constraints using different dedicated solvers, including an integer constraint solver and a string constraint solver. However, the focus of that work is on constraints written purely in the declarative language Alloy. In contrast, our work is on hybrid invariants, which are written partly in Alloy and partly in Java; moreover, we perform incremental solving using solvers designed for constraints in different programming paradigms.

Khalek's doctoral work [18, 19] designed the JABAL framework for writing and solving constraints in a *mixed* declarative and imperative paradigm, which applies a solver for declarative constraints and a solver for imperative constraints together, and lays out the initial groundwork. This paper introduces a new technique for formulation and solving of *hybrid invariants*, which provides a tight integration of declarative constraint solving and imperative constraint solving and applies them in synergy, where information that assists in imperative solving is first computed during declarative solving and then utilized to make imperative solving much more efficient. Moreover, we optimize our technique using a novel approach, namely transcoping [27], which was not previously developed for hybrid invariants.

8. Conclusions and Future Work

We introduced HyTeK, a novel technique for test input generation that builds test suites from *hybrid* input specifications. In order to deal with these hybrid invariants, our technique combines Korat, an efficient mechanism for producing test inputs from imperative invariants, with SAT solving to process declarative invariants. Hybrid invariants are more flexible and general than fully imperative or fully declarative invariants, allowing software engineers to design specifications that better fit their specification preferences or better reflect the nature of the problem being modeled, thus rendering the process of specification less error prone. Moreover, the availability of parts of the specification in different paradigms enabled us to benefit from optimizations of one context in the other one, leading to more efficient generation of test inputs than doing so from fully imperative or fully declarative invariants. The resulting hybrid technique is a sophisticated combination of known techniques for test generation (from imperative and declarative specifications), whose associated profit is better overall than the sum of the parts that it incorporates.

We also presented a technique for automatically tailoring hybrid input specifications to improve test input generation. This technique automatically explores alternative orderings of the specification components on the imperative side, and when part of the invariant is provided both declaratively and imperatively, it decides the most convenient setting (imperative or declarative) in which each part is to be solved.

We assessed the approach presented in this article on two relevant and interesting data structures, whose invariants are regarded as among the most difficult for automated analysis. Our evaluation showed that, for these case studies, our technique performs substantially better than previous state-of-the-art approaches in declarative constraint solving as well as imperative constraint solving. The transcoping mechanism proved to be an effective means to discover the most convenient hybrid input specification, provided there is room to consider alternatives.

There are various lines for future work. First, notice that in our current approach, the interaction between the declarative and imperative constraint solving approaches is given essentially in one direction: information from the declarative side is exploited by the imperative side of the generation. We are exploring ways of profiting from information gathered while solving the imperative side, to be exploited by the declarative constraint solving. Also, our experimental evaluation has mainly focused on whether generating tests from hybrid specifications can be more efficient than doing so from fully imperative or fully declarative specifications, and whether transcoping is able to effectively identify the best hybrid specifications, from the point of view of efficiency in generation. We have not considered other aspects of our techniques in our evaluation, such as the most appropriate way of performing transcoping (e.g., scopes to start

the analysis, percentage of candidates to promote to larger scopes, etc.), or the profit that tight bounds contribute to the analyses. Further experiments with other data structures, as well as with appropriate metrics to identify the contribution of tight bounds, are necessary, and are part of our current and future work. We also plan to evaluate the effectiveness of our techniques in other contexts besides that of complex heap-allocated data structures.

Acknowledgments

This work was partially supported by ITBACyT 8, ANPCyT PICT 2010-1690 and 2012-1298, and by the MEALS project (EU FP7 MEALS - 295261). Khalek's work was done at the University of Texas at Austin. Khalek's and Khurshid's work was funded in part by the National Science Foundation (NSF Grant Nos. CCF-0845628 and CNS-1239498). We thank Elena Morin for proofreading this article.

References

[1] P. Abad, N. Aguirre, V. S. 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 ICST 2013.

[2] C. Boyapati, S. Khurshid and D. Marinov, *Korat: Automated Testing based on Java Predicates*, in ISSTA 2002.

[3] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. Rustan M. Leino and E. Poll, *An overview of JML tools and applications*, in STTT 7(3), Springer, 2005.

[4] F. S.-H. Chang and D. Jackson. Symbolic model checking of declarative relational models. In *ICSE 2006*.

[5] K. Claessen and J. Hughes, *QuickCheck: a lightweight tool for random testing of Haskell programs*, in ICFP 2000.

[6] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS 2008*.

[7] G. Dennis, F. Chang, D. Jackson. *Verification of Code with SAT*, in ISSTA 2006, ACM, 2006.

[8] B. Dutertre and L. D. Moura. The Yices SMT solver. Technical report, 2006.

[9] N. Een and N. Sorensson. An Extensible SAT-solver. In *SAT 2003*.

[10] M. Frias, J. Galeotti, C. Pombo, and N. Aguirre. DynAlloy: Upgrading Alloy with actions. In *ICSE 2005*.

[11] J.P. Galeotti, N. Rosner, C. López Pombo, M. Frias, *TACO: Efficient SAT-Based Bounded Verification Using Symmetry Breaking and Tight Bounds*. IEEE TSE 39(9): 1283-1307 (2013).

[12] S. Ganov, S. Khurshid, D. E. Perry. Annotations for Alloy: Automated Incremental Analysis Using Domain Specific Solvers. In *ICFEM 2012*.

[13] J. Geldenhuys, N. Aguirre, M. F. Frias and W. Visser, *Bounded Lazy Initialization*, in NFM 2013, LNCS, Springer, 2013.

[14] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak and D. Marinov, *Test generation through programming in UDITA*, in ICSE 2010, Cape Town, South Africa.

[15] G. J. Holzmann. The model checker SPIN. *IEEE TSE 1997*.

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

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

[18] S. A. Khalek. *Systematic Testing Using Test Summaries: Effective and Efficient Testing of Relational Applications*. Ph.D. Thesis. University of Texas at Austin, 2011.

[19] S. A. Khalek, V. P. Narayanan, and S. Khurshid. Mixed constraints for test input generation – An initial exploration. In *ASE 2011 (Short paper)*.

[20] S. A. Khalek, G. Yang, L. Zhang, D. Marinov and S. Khurshid, *TestEra: A Tool for Testing Java Programs using Alloy Specifications*, in ASE 2011, IEEE, 2011.

[21] S. Khurshid and D. Marinov, *TestEra: Specification-Based Testing of Java Programs Using SAT*, Automated Software Engineering 11(4), Springer, 2004.

[22] A. S. Koksal, V. Kuncak, and P. Suter. Constraints as Control. In *POPL 2012*.

[23] B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*, Addison-Wesley, 2000.

[24] A. Milicevic, S. Misailovic, D. Marinov and S. Khurshid, *Korat: A Tool for Generating Structurally Complex Test Inputs*, in ICSE 2007, IEEE Press, 2007.

[25] A. Milicevic, D. Rayside, K. Yessenov, and D. Jackson. Unifying execution of imperative and declarative code. In *ICSE 2011*.

[26] C. Pacheco, S. K. Lahiri, M. D. Ernst and T. Ball, *Feedback-Directed Random Test Generation*, in ICSE 2007, IEEE, 2007.

[27] N. Rosner, C. G. López Pombo, N. Aguirre, A. Jaoua, A. Mili and M. F. Frias, *Parallel Bounded Verification of Alloy Models by TranScoping*, in VSTTE 2014, LNCS, Springer, 2013.

[28] H. Samimi, E. D. Aung, and T. D. Millstein. Falling back on executable specifications. In *ECOOP 2010*.

[29] J. Siddiqui and S. Khurshid, *An Empirical Study of Structural Constraint Solving Techniques*, in ICFEM 2009, LNCS, Springer, 2009.

[30] E. Uzuncaova. *Efficient specification-based testing using incremental techniques*. PhD thesis, UT@Austin, 2008.

[31] E. Uzuncaova and S. Khurshid. Constraint prioritization for efficient analysis of declarative models. In *FM 2008*.

[32] W. Visser, C. Păsăreanu and S. Khurshid, *Test input generation with java PathFinder*, in ISSTA 2004, ACM, 2004.

[33] T. Xie, D. Marinov and D. Notkin, *Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests*, in ASE 2004, IEEE, 2004.

[34] R. N. Zaeem and S. Khurshid. Contract-based data structure repair using Alloy. In *ECOOP 2010*.

[35] The HyTeK distribution can be downloaded from http://bonnie.exp.dc.uba.ar/hytek-oopsla.tar.gz

[36] http://alloy.mit.edu/