Generation of test data structures using Constraint Logic Programming*

Valerio Senni[1] and Fabio Fioravanti[2]

[1] DISP, University of Rome Tor Vergata, Rome, Italy
`senni@disp.uniroma2.it`
[2] Dipartimento di Scienze, University 'G. D'Annunzio', Pescara, Italy
`fioravanti@sci.unich.it`

Abstract. The goal of Bounded-Exhaustive Testing (BET) is the automatic generation of *all* the test cases satisfying a given invariant, within a given bound. When the input has a complex structure, the development of correct and efficient generators becomes a very challenging task. In this paper we use Constraint Logic Programming (CLP) to systematically develop generators of structurally complex test data.

Similarly to *filtering*-based test generation, we follow a declarative approach which allows us to separate the issue of (i) defining the test structure and invariant, from that of (ii) generating admissible test input instances. This separation helps improve the correctness of the developed test case generators. However, in contrast with filtering approaches, we rely on a symbolic representation and we take advantage of efficient search strategies provided by CLP systems for generating test instances.

Through some experiments on examples taken from the literature on BET, we show that CLP, by combining the use of constraints and recursion, allows one to write intuitive and easily understandable test generators. We also show that these generators can be much more efficient than those built using ad-hoc filtering-based test generation tools like Korat.

1 Introduction

The identification of test cases, which is a central task in the testing process, is very often carried out as a manual activity. As a consequence, it is error-prone, it has limited applicability, and can be very expensive (around 50% of the cost of software development). Formal and automated techniques have thus received interest from the testing community because they can be used to develop test suites in a more systematic and affordable way, by enforcing correctness and allowing flexible integration with the considered code coverage criteria.

In this paper we focus on the *bounded-exhaustive testing* [7,26] approach (BET), whose goal is to test a program on *all* the input instances satisfying a given invariant, up to a given bound on their size. The motivation underlying the BET approach is based on the observation that defects, if any, are likely to appear already in small-sized instances of the inputs.

* This paper has been accepted to appear in the proceedings of the conference Tests & Proofs 2012, *Lecture Notes in Computer Science* 7305, Springer, 2012 (in press).

Automated test input generators should be (i) *correct*, that is, they should generate only test input instances which satisfy the considered invariant, and (ii) *efficient*, when generating test input candidates and filtering out those which are not admissible, so that they can be applied to large and realistic domains.

Modern software often manipulates input data with complex structure (like trees and graphs) and satisfying non-trivial invariants (like sorting, coloring, depth balancing). The correct and efficient generation of structurally complex inputs is a challenging task because the number of test input candidates can grow very fast, but only a few inputs, which satisfy the desired invariants, are to be selected as admissible. The generation of large and complex test objects is also required by some recent application domains, such as XML documents generation, considered in [18], where an RSS feed parser is tested for HTML injection vulnerabilities, and in [4], where they are used for testing Web Services.

In this paper we propose a framework based on Constraint Logic Programming (CLP) for the systematic development of generators of large sets of structurally complex test data.

Similarly to filtering-based techniques, we adopt a declarative approach which allows us to separate the issue of (i) defining the test input structure and invariants, from that of (ii) generating admissible test input instances. This separation helps improve the correctness of the developed test case generators, because it lets testing engineers write *what* to generate, in a very intuitive and easily understandable way. Efficient CLP search strategies are then used for specifying *how* test instances should be generated.

Although heavy optimizations require in-depth knowledge of CLP techniques, we will show that excellent results can be already obtained by following some simple programming guidelines. In particular, we show that test generators should be written following the so-called *constrain-and-generate* approach, according to which the structural and invariant constraints should be computed first, postponing as much as possible the actual generation of test instances. This allows the CLP computation engine to prune the search space at the symbolic level, avoiding useless executions of the expensive instantiation phase.

We experiment with some examples taken from the literature on BET, and we show that CLP, by combining the use of constraints and recursion, allows us to write test generators which are simpler and more efficient than those built using the ad-hoc test generation tool Korat [29]. However, our focus is not on deriving CLP generators from Korat ones. Rather, we assume that those generators have been derived from a given, abstract, model and we evaluate their efficiency.

Our evaluation shows that modern CLP systems can be used effectively as a core component to construct fast and correct test generators and for more complex test suite development frameworks.

In Sec. 2, we briefly recall the Korat approach and illustrate, as a case-study, a Red-Black Trees generator. In Sec. 3, we introduce our CLP-based approach and we illustrate its expressiveness by providing a clean and strongly declarative definition of a Red-Black Trees generator. We also show how to use some optimization techniques known in the Logic Programming community to obtain even faster generators. Finally, in Sec. 4, we carry out an extensive comparison between our CLP-based approach and that of Korat.

2 The Korat Approach

We now illustrate the Korat approach for writing automated test generators. Korat [29] is a tool for bounded-exhaustive testing of Java programs, which is specifically tailored for the construction of structurally complex test inputs. It allows the generation of complex data structures by providing primitives to populate an object domain, to initialize objects, and to set links among them.

Given a data structure definition, Korat requires (1) a finitization method, which defines the bounds of the search space, and (2) a method `repOK()`, which specifies the data structure invariant. Korat performs a systematic search of the program input space, avoiding the full exploration of failing regions and the generation of isomorphic structures (i.e. equal modulo Java objects identity). Details of the optimizations used in the search can be found in [5,25,29].

We now illustrate how the Korat approach works by applying it for writing a test input generator for Red-Black Trees.

Example 1. A Red-Black Tree [8] is a binary search tree where each node has two labels: a *color*, which is either red or black, and an integer, called *key* (for the purpose of test generation, node values are abstracted away in the definition of the data structure). Therefore, it satisfies the following type equation:

```
Color ::= 0 | 1
Key   ::= ... | -1 | 0 | 1 | ...
Tree  ::= e | Color x Key x Tree x Tree
```

where `0` and `1` denote *red* and *black*, respectively, and `e` denotes the emtpy tree. A Red-Black Tree must also satisfy the following three invariants:

($\mathbf{I}_1$) every path from the root to a leaf has the same number of black nodes,
($\mathbf{I}_2$) no red node has a red child, and
($\mathbf{I}_3$) for every node n, all the nodes in the left (respectively, right) subtree of n, if any, have keys which are smaller (respectively, bigger) than the key labeling n.

Since Red-Black Trees enjoy a weak form of balancing, operations such as inserting, deleting, and finding values are more efficient, in the worst-case, than in ordinary binary search trees.

We consider the Red-Black Tree generator implementation taken from the Korat repository.[3] The `RedBlackTree` class, shown in Fig. 1, uses an internal class `Node` defining the generic node of the Red-Black Tree data structure. The `Node` class has integer attributes `key`, `color` and `value`, and attributes `left`, `right` and `parent` of type `Node`.

The finitization method `finRedBlackTree`, shown in Fig. 2, is used to define the search space for generating the test candidates. It accepts the following arguments:

[3] `https://korat.svn.sourceforge.net/`

```
public class RedBlackTree {
    private Node root = null;
    private int size = 0;
    private static final int RED = 0;
    private static final int BLACK = 1;

    public static class Node  {
        int key, value;
        Node left = null, right = null, parent;
        int color = BLACK;
    }

    METHODS...
}
```

Fig. 1: Red-Black Trees java class

```
public static IFinitization finRedBlackTree
    (int numEntries, int minSize, int maxSize, int numKeys) {

        IFinitization f          = FinitizationFactory.create(RedBlackTree.class);
        IClassDomain entryDomain = f.createClassDomain(Node.class, numEntries);
        IObjSet entries          = f.createObjSet(Node.class, true);
        entries.addClassDomain(entryDomain);

        IIntSet sizes = f.createIntSet(minSize, maxSize);
        IIntSet keys = f.createIntSet(-1, numKeys - 1);
        IIntSet values = f.createIntSet(0);
        IIntSet colors = f.createIntSet(0, 1);

        f.set("root",        entries);   f.set("size",       sizes);
        f.set("Node.left",   entries);   f.set("Node.color", colors);
        f.set("Node.right",  entries);   f.set("Node.key",   keys);
        f.set("Node.parent", entries);   f.set("Node.value", values);

        return f;
    }
```

Fig. 2: Red-Black Trees finitization method

numEntries, denoting the number of objects of class Node which can be used for building the Red-Black Tree, minSize and maxSize, denoting the minimum and maximum number of nodes of the tree (maxSize is expected to be not bigger than numEntries), and numKeys, denoting the upper bound for keys values (with lower bound 0). The methods createClassDomain, createObjSet, and addClassDomain populate the object domain, while the calls to the method createIntSet populate the integer domains for tree sizes and node keys, values, and colors, respectively. Finally, the method set is used to map class attributes to the appropriate domains (nodes or integers), which will be used by Korat during the candidate instantiation phase. Notice that, though color and value are declared as integers, color can only take values in $\{0, 1\}$ and value is a constant, (so, in practice, values are abstracted away).

The method repOK(), which can be found in Appendix A, is an imperative specification of the Red-Black Tree data structure invariants $\mathbf{I}_1$, $\mathbf{I}_2$, and $\mathbf{I}_3$. It is used by Korat to filter, among the many candidate trees generated, only those that satisfy the Red-Black Trees invariants.

3 The CLP-based Approach

Logic Programming [24] is a declarative programming paradigm based on a computational interpretation of resolution-based first-order theorem proving. Sets of formulas can be regarded as programs and proof search can be used as a general-purpose problem solving mechanism.

Constraint Logic Programming (briefly, CLP) [20] extends Logic Programming with constraints, which are managed by fast, domain-specific, constraint solvers. During the proof search process, constraints are collected in a store which is required to be consistent at each step, and the problem solving process amounts to reducing the initial problem to a satisfiable set of constraints. Among several other applications, CLP has shown to be well suited for encoding and solving combinatorial problems [11].

Let us now briefly recall the CLP framework and its operational semantics, for more details we refer the reader to [20]. Let Σ be a logic language signature $\Sigma = \langle \mathcal{F}, \mathcal{V}, \Pi \cup \Pi_C \rangle$, where $\mathcal{F}$ is a finite set of function and constant symbols, $\mathcal{V}$ is a denumerable collection of variables, $\Pi \cup \Pi_C$ is a finite set of predicate symbols, where Π and Π_C are disjoint sets. *Atoms* are of the form $p(s_1, \ldots, s_n)$ where p is a predicate symbol in Π and s_i's are $(\mathcal{F}, \mathcal{V})$-terms. A *constraint* is a first-order formula over $\mathcal{F}$, $\mathcal{V}$, and Π_C, (typically, a conjunction such as `X#>3,X+Y#<0`). In logic programming notation, a comma denotes a conjunction and the symbol `:-` denotes the implication $\leftarrow$. Strings denote variables if they start with a capital letter and constants, otherwise. Comments are started by `%`. When variables need not be named, they are replaced by `_`. A CLP *program* P over Σ is a finite set of *clauses* of the form:

$H \ :- \ c,\ A_1, \ldots, A_m.$

where c is a constraint, and H and A_i's are atoms.

A CLP system computes the answers to a user query (called *goals*) of the form $c, A_1, \ldots, A_k$ against a program P, where c is a constraint and $A_1, \ldots, A_k$ is a finite conjunction of atoms. Given a program P, an *answer* to a goal $c, A_1, \ldots, A_k$ is a substitution ϑ such that $\forall(A_1, \ldots, A_k)\vartheta$ is a logical consequence of P and $c\vartheta$ is satisfiable in the constraints theory. We denote by $[\![G]\!]_P$ the set of all the answers to the goal G against the program P. We will feel free to omit the subscript P whenever the intended program is clear from the context. An answer ϑ is *ground* whenever $(c, A_1, \ldots, A_k)\vartheta$ contains no variables.

The programming paradigm of CLP, sometimes referred to as constrain-and-generate [27], is structured mainly in two phases: first constraints are added to the constraint store and checked for consistency (*constrain*), then the solver instantiates variables to produce actual values that satisfy the constraints in the store (*generate*). Since in the constrain phase the constraint store is checked for consistency at each modification, several unsatisfiable cases are rejected at the symbolic level. When the problem is satisfiable, the search for a satisfying substitution is committed to a dedicated solver. This behavior is quite different from the generate-and-test approach of Korat and other filter-based techniques [14,22,29]. In particular, we focus on Constraint Logic Programming over Finite Domains (CLP(FD) [27]) where constraints are linear arithmetic equalities and inequalities on variables ranging over finite integer domains. Note that CLP(FD) can handle more general constraints, however, for the examples considered in this paper, arithmetic equalities and inequalities are sufficiently expressive.

The FD comparison predicates we assume available in Π_C are `#=`, `#>`, `#>=`, `#<`, `#=<`. The function signature $\mathcal{F}$ extends the set $\{\texttt{+},\texttt{-},\texttt{*}\}\cup\mathbb{Z}$ and the set $\{\,\texttt{[]},\,\texttt{[\_|\_]}\,\}$ of list constructors. Predicates and function symbols in FD have the standard interpretation over $\mathbb{Z}$. We assume to have the built-in predicates: `fd_domain(Vs,Min,Max)`, that constrains all the variables in the list `Vs` to range over $[\texttt{Min},\ldots,\texttt{Max}]\subset\mathbb{Z}$, and `fd_labeling(Vs)`, that forces each variable in `Vs` to assume a concrete value among those allowed by the current constraint store (an additional `Settings` argument is contemplated, for configuring the search process).

We propose the following instantiation of the constrain-and-generate paradigm for the implementation of efficient (filter-based) test case generators:

```
gen_structure(Struct,P1,...,Pk) :-
    % Preamble                                (constrain)
      ...definition of the variables in Vars and their domain,
    % Symbolic Definition                     (constrain)
      structure(Struct,P1,...,Pk,Vars),       % data structure shape
      ...filters,                             % invariants
    % Instantiation                           (generate)
      fd_labeling(Vars).
```

The semantics of this predicate is that, for any given value of the parameters `P1`,...,`Pk` we build a structure `Struct` that satisfies the desired invariants.

The `Preamble` contains the definition of the set `Vars` of the required variables and their domains. The `Symbolic Definition` phase contains: (i) a call to a predicate `structure` which defines, by structural induction, the data structure shape (e.g. list, tree, graph) using variables in `Vars` as placeholders for values, and (ii) a `filters` part which contains a conjunction of predicates that assert constraints among the variables in `Vars`. Finally, the `Instantiation` phase invokes the built-in labeling mechanism, possibly using problem-specific settings to configure the search strategy. The solver tries to minimize backtracking on assignments and each assignment triggers a deterministic propagation towards related variables, which reduces their domain and the set of future choices.

Concerning point (i) of the `Symbolic Definition` phase above, in this paper we consider simple tree-like structures, where Logic Programming can show the advantage of the built-in unification mechanism. Graph-like structures are a bit more involved to deal with and one can rely on a classical incidence/adjacency-matrix representation or rely on more sophisticated decompositions [31].

Let `gen_structure` be a generator predicate and let `p1`,...,`pk` be concrete values for the parameters, the set $\mathcal{T}$ of *test cases* induced by the generator is $\mathcal{T} = \{\texttt{Struct}\vartheta \mid \vartheta \in [\![\texttt{gen\_structure(Struct,p1,...,pk)}]\!]\}$. Note that, due to the final `Instantiation` phase, we have that $\mathcal{T}$ contains only ground test cases.

3.1 Red-Black Trees

Let us now illustrate the CLP-based specification of a Red-Black Tree generator. This generator is parameterized, as for Korat, by the maximum and minimum tree size (defined as the number of its nodes), and by the maximum value for the keys. Since we do not generate nodes beforehand, but on demand, we do not need an extra parameter for counting the number of nodes, as Korat does. The following clause defines the predicate `rbtree`:

```
rbtree(T,MinSize,MaxSize,NumKeys) :-
    % Preamble
      MinSize#=<S, S#=<MaxSize, fd_labeling([S]),
      varlist(S,Keys), varlist(S,Colors), Max#=NumKeys-1,
      fd_domain(Keys,0,Max), fd_domain(Colors,0,1),
    % Symbolic Definition
      lbt(T,S,Keys,[]),                               % data structure shape
      pi(T,_), ci(T,Colors,[]), ordered(T,0,NumKeys), % filters
    % Instantiation
      fd_labeling(Keys), fd_labeling(Colors).
```

where the predicate `varlist(N,L)`, is used for constructing a list `L` of `N` fresh variables. Given the ground (non-negative) input integers `minSize`, `maxSize`, and `numKeys`, the set ⟦`rbtree(T,minSize,maxSize,numKeys)`⟧ contains all the red-black trees of size ranging in {`minSize`,...,`maxSize`}, with keys ranging in {`0`,...,`numKeys-1`}.

The first line of the Preamble chooses a tree size value `S`. Then, two lists of (distinct) variables are defined, `Keys` and `Colors`, with the corresponding domains, {`0`,...,`NumKeys-1`} and {`0`,`1`}, respectively. These variables are placed along the tree structure in the `Symbolic Definition` part by the predicate `lbt`, which defines (2-)labeled binary trees by structural induction:

```
1. lbt(         e,S,    Ks, Ks) :- S#=0.
2. lbt(t(_,K,L,R),S,[K|Ks],NKs) :- S#>=1, SL#>=0, SR#>=0,
                                   NS#=S-1, fdsum(NS,SL,SR),
                                   lbt(L,SL,Ks,TKs), lbt(R,SR,TKs,NKs).
```

The first argument is either the constant `e` denoting the empty tree or a term `t(C,K,L,R)` denoting a (non-empty) tree with left subtree `L`, right subtree `R`, and whose root node is labeled with color `C` and key `K`. The second argument is the size of the tree (the number of nodes) which, in clause `2`, is at least `1` and is non-deterministically split by the `fdsum` predicate into a pair of non-negative integers `SL` and `SR` denoting the size of the left and right subtrees, respectively, such that $\mathtt{S} = \mathtt{SL} + \mathtt{SR} + 1$. The left and right subtrees are then constructed recursively. Note that the variables in `Keys` are placed in distinct nodes.

The predicate `pi` (for *path invariant*) encodes the invariant $(\mathbf{I}_1)$ of Sec. 1 and is defined as follows:

```
 9. pi(e         ,C) :- C#=0.
10. pi(t(C,_,L,R),D) :- ND#>=0, D#=ND+C, pi(L,ND), pi(R,ND).
```

The semantics of `pi` is the following: for a given tree `t`, `pi(t,d)` holds if and only if on all root-to-leaf paths in `t` there are exactly `d` black nodes. In this case, we say that `d` is the value of the *black-nodes counter* of `t`. Note that if a tree is empty then its black-nodes counter is `0`, otherwise, the black-nodes counter is computed by adding the 'color' of the root (i.e. `0`, if red, and `1`, if black) to the black-nodes counter of (both) its subtrees (that must have the same value).

The predicate `ci` (for *color invariant*) encodes the invariant $(\mathbf{I}_2)$ of Sec. 1 and is defined as follows:

```
4. ci(         e,    Cs, Cs).
5. ci(t(C,_,L,R),[C|Cs],NCs) :- C#=1,                  % root is black
                                ci(L,Cs,TCs), ci(R,TCs,NCs).
6. ci(t(C,_,L,R),[C|Cs],NCs) :- C#=0,                  % root is red
                                not_redroot(L), not_redroot(R),
                                ci(L,Cs,TCs), ci(R,TCs,NCs).
7. not_redroot(         e).
8. not_redroot(t(C,_,_,_)) :- C#=1.                    % root is black
```

Note that the color variables are placed in distinct nodes. In clause 6 the color invariant is enforced by testing the color of the roots of the left and right subtrees.

Finally, the predicate `ordered` defines the invariant $(\mathbf{I}_3)$ in Sec. 2, concerning the ordering of the keys, and is defined as follows:

```
11. ordered(          e,  _,  _).
12. ordered(t(_,N,L,R),Min,Max) :- Min #=< N, N #< Max, M #= N+1,
                                   ordered(L,Min,N), ordered(R,M,Max).
```

There is a simple correspondence between the CLP(FD) generator and the Korat generator. The predicate `lbt` is essentially a definition of the tree data type, as given in Sec. 1. This allows us to start from *trees* rather than from *graphs*, which is a significant advantage over Korat, which must perform the acyclicity check. The filter predicates `ci`, `pi`, and `ordered` are very similar to the Korat code for `repOk()`, which can be retrieved in the Korat repository. Note that `repOk()` executes three tests: (i) acyclicity, using the `repOkAcyclic()` procedure, (ii) invariants $(\mathbf{I}_1)$ and $(\mathbf{I}_2)$ using the `repOkColors()` procedure, and (iii) ordering invariant $(\mathbf{I}_3)$, using the `repOkKeysAndValues()` procedure. The `repOkColors()` procedure returns `true` iff the goal `pi(T,_),ci(T,Colors,[])` succeeds and `repOkKeysAndValues()` returns `true` iff the goal `ordered(T,0,Max)` succeeds.

We compared the efficiency of the CLP-based Red-Black Trees generator with the Korat-based one. Following the approach of [25], we consider the 'canonical set' ⟦`rbtree(T,s,s,s)`⟧, which is the set of all red-black trees of `s` nodes and keys ranging in {`0`,. . . ,`s-1`}.

The results of this comparison are summarized in Fig. 3 (columns 1-4,9) and show that the CLP-based Red-Black Trees generator, which has been run using two different CLP systems, SICStus[4] and GNU Prolog[5], is very efficient with respect to the Korat generator. Further details and discussion about the experimental evaluation can be found in Sec. 4.

3.2 Optimizations

In this section we discuss some optimization techniques available in the field of logic programming that can be used for building even more efficient test case generators. Simple optimizations can be done by using information coming from groundness analysis [19] and determinacy checking [23], that allow us to determine when a predicate has *at most* one answer. In order to improve determinism, one can put such predicates

[4] `http://www.sics.se/sicstus/`

[5] `http://www.gprolog.org/`

Fig. 3: Comparison of Red-Black Trees generators. The table reports the size of the red-black trees (column 1), the number of computed red-black trees (2), the time, in seconds, for generating all the structures running the original CLP generator of Sec. 3 using GNU Prolog and SICStus (3-4), the partially evaluated CLP generator of Sec. 3.2 (5-6), the synchronized CLP generator of Sec 3.2 (7-8), and Korat (9). Zero means less than 10 ms and (-) means more than 200 seconds.

Size	Trees	Time						
		Original		Partially evaluated		Synchronized		Korat
		GNU	SICStus	GNU	SICStus	GNU	SICStus	Korat
6	20	0	0.01	0	0	0	0	0.47
7	35	0.01	0.06	0	0.03	0	0.01	0.63
8	64	0.02	0.20	0.01	0.07	0	0.03	1.49
9	122	0.08	0.71	0.04	0.28	0	0.06	4.51
10	260	0.29	2.60	0.16	0.98	0.01	0.13	21.14
11	586	1.07	9.52	0.58	3.55	0.03	0.26	116.17
12	1296	3.98	35.19	2.17	13.00	0.06	0.54	-
13	2708	14.85	131.13	8.15	47.90	0.12	1.17	-
14	5400	55.77	-	30.73	177.63	0.26	2.49	-
15	10468	-	-	115.59	-	0.55	5.35	-
20	279264	-	-	-	-	25.90	-	-

in the `Preamble` so that they are evaluated only once (we applied this optimization to the Heaparrays example, discussed in Sec. 4) and whose code can be found in Appendix B).

A more sophisticated technique is Program Transformation, that is a semantics-preserving program rewriting technique which is applicable, among other languages, also to Constraint Logic Programming [12]. It can be used to optimize and synthesize programs [13] and it is based on rewriting *rules*, whose application is directed by *strategies*. Here we use Program Transformation to optimize our Red-Black Trees generator. The transformations are performed by using our transformation tool MAP [2].

Step 1. We perform a *partial evaluation* [21] of `lbt`, `pi`, `ci`, and `ordered` w.r.t. their first argument and the term domain `T ::= e | t(_,_,T,T)`, by using the *unfolding* rule (which unrolls predicate calls by using their definitions). For reasons of space, we show the effect of this transformation only for predicate `pi`:

```
pi(                                  e,0).
pi(t(C,_,             e,            e),D) :- D#=C.
pi(t(C,_,             e,t(Cr,Lr,Kr,Rr)),D) :- D#=C, pi(t(Cr,Kr,Lr,Rr),0).
pi(t(C,_,t(Cl,Ll,Kl,Rl),            e),D) :- D#=C, pi(t(Cl,Kl,Ll,Rl),0).
pi(t(C,_,t(Cl,Ll,Kl,Rl),t(Cr,Lr,Kr,Rr)),D) :- ND#>=0, D#=ND+C,
                                              pi(t(Cl,Kl,Ll,Rl),ND),
                                              pi(t(Cr,Kr,Lr,Rr),ND).
```

(the rest of the clauses can be found in Appendix C). Performance improvement is due to the instantiation of the heads of the clauses which reduces unification successes and prunes the search tree, at the cost of an increase in the number of clauses. The performance of the partially evaluated generator w.r.t. that of the original one is shown in Fig. 3. ■

The execution of a conjunction of atoms that share a variable ranging over a term domain (such as `T ::= e | t(_,_,T,T)` above) may be inefficient and even non-terminating, under the standard depth-first CLP evaluation strategy. In the Red-Black Trees generator the predicates `lbt`, `pi`, `ci`, and `ordered` share the tree variable `T`. Each tree is traversed four times. Indeed, there are productive interactions between those predicates that we do not take advantage of. Namely, the check of the path invariant of pi may produce earlier failure for a given distribution of the nodes of T. Similarly for the choice of nodes colors in ci and the lengths of the paths computed in T.

Step 2. We apply a program transformation strategy that optimizes programs to avoid *multiple traversals* of a data structure [30]. This strategy replaces a conjunction of atoms G by a new atom that performs a synchronized execution on the shared variables, and produces the same results. The effect is to obtain (i) a single traversal of terms in the shared domain and (ii) a better interaction among the constraints of each predicate in G to promote as early as possible failures/successes. We take advantage of the previous partial evaluation step by applying the current transformation to the partially evaluated versions of lbt, pi, ci, and ordered. We do not have enough space here to discuss the details of this transformation, but we give a sketch of it. In goal G=lbt(T,S,Keys,[]),pi(T,_),ci(T,Colors,[]),ordered(T,0,NumKeys) (occurring in the definition of rbtree) the atoms share the variable T. We introduce a new predicate sync (which stands for *synchronized*) defined by the following clause:

```
sync(T,S,Keys,NewKeys,Colors,NewColors,Min,Max,D) :-
         lbt(T,S,Keys,NewKeys),                                   (†)
         pi(T,D), ci(T,Colors,NewColors), ordered(T,Min,Max).
```

where the goal G is abstracted to one containing only variables. Then we derive a recursive definition of sync in two steps. First, we perform a partial evaluation, by *unfolding*, and we obtain (after rearrangement of the constraints to the left of the other atoms) the following new definition:

```
sync(                            e,A,    B,B,    C,C,_,_,0) :- A#=0.
sync(t(A,B,         e,          e),C,[B|D],D,[A|E],E,F,G,H) :- C#=1, H#=A, F#=<B, B#<G.
sync(t(A,B,         e,t(C,D,E,F)),G,[B|H],I,[A|J],K,L,M,N) :-
      G#>=2, O#=G-1, A+C#>0, N#=A, L#=<B, B#<M, P#=B+1,
      lbt(t(C,D,E,F),O,H,I), pi(t(C,D,E,F),O), ci(t(C,D,E,F),J,K), ordered(t(C,D,E,F),P,M).
sync(t(A,t(B,C,D,E),F,         e),G,[F|H],I,[A|J],K,L,M,N) :-
      G#>=2, O#=G-1, A+B#>0, N#=A, L#=<F, F#<M,
      lbt(t(B,C,D,E),O,H,I), pi(t(B,C,D,E),O), ci(t(B,C,D,E),J,K), ordered(t(B,C,D,E),L,F).
sync(t(A,F,t(B,C,D,E),t(G,H,I,J)),K,[F|L],M,[A|N],O,P,Q,R) :-
      K#>=3, S#=K-1, T#>0, U#>0, A+B#>0, A+G#>0, V#>=0,
      R#=V+A, P#=<F, F#<Q, W#=F+1, fdsum(S,T,U),
      lbt(t(B,C,D,E),T,L,X), pi(t(B,C,D,E),V), ci(t(B,C,D,E),N,Y), ordered(t(B,C,D,E),P,F),
      lbt(t(G,H,I,J),U,X,M), pi(t(G,H,I,J),V), ci(t(G,H,I,J),Y,O), ordered(t(G,H,I,J),W,Q).
```

Next, by *folding*, we replace goals that match the body of the clause (†) by the corresponding instances of the head and we obtain the following clauses:

```
1. sync(                            e,A,    B,B,    C,C,_,_,0) :- A#=0.
2. sync(t(A,B,         e,          e),C,[B|D],D,[A|E],E,F,G,H) :- C#=1, H#=A, F#=<B, B#<G.
3. sync(t(A,B,         e,t(C,D,E,F)),G,[B|H],I,[A|J],K,L,M,N) :-
         G#>=2, O#=G-1, A+C#>0, N#=A, L#=<B, B#<M, P#=B+1,
         sync(t(C,D,E,F),O,H,I,J,K,P,M,O).                       % replacement
4. sync(t(A,t(B,C,D,E),F,         e),G,[F|H],I,[A|J],K,L,M,N) :-
         G#>=2, O#=G-1, A+B#>0, N#=A, L#=<F, F#<M,
         sync(t(B,C,D,E),O,H,I,J,K,L,F,O).                       % replacement
5. sync(t(A,F,t(B,C,D,E),t(G,H,I,J)),K,[F|L],M,[A|N],O,P,Q,R) :-
         K#>=3, S#=K-1, T#>0, U#>0, A+B#>0, A+G#>0, V#>=0,
         R#=V+A, P#=<F, F#<Q, W#=F+1, fdsum(S,T,U),
         sync(t(B,C,D,E),T,L,X,N,Y,P,F,V),                       % replacement
         sync(t(G,H,I,J),U,X,M,Y,O,W,Q,V).                       % replacement
```

The definition of the predicate sync is now made of the clauses $\{1, 2, 3, 4, 5\}$. The final step of this transformation consists in a further application of *folding* which replaces the goal G in the definition of rbtree (which is an instance of the body of clause (†)) by the goal sync(T,S,Keys,[],Colors,[],0,NumKeys,_). The performance improvements of the synchronized generator w.r.t. the partially evaluated generator and the original one are shown in Fig. 3. ■

By correctness of the transformation rules and of the transformation strategy, we are ensured that the two generators are equivalent, in terms of their least Herbrand models [24] and, thus, define the same set of test cases.

4 Experimental Evaluation

In order to measure the effectiveness of our technique, we retrieved the code of several test case generators from the Korat repository and we encoded the corresponding CLP(FD)-based generators (their code can be found in Appendix B).

We considered generators for: (i) sorted lists of integers, (ii) an array-based representation of the heap data structure, (iii) integer-labeled search trees, and (iv) an array-based representation of disjoint partitions of a set. For disjoint sets, we found it difficult to reverse-engineer the exact specification from the Java code in the Korat repository and, thus, we decided to recode the Korat version from scratch w.r.t. an abstract model (the code can be found in Appendix D). The new Korat generator has performances not far from the original one but selects a restricted class of structures.

In all our experiments, we found that the performance of CLP-based test generators is always much better than the performance of the corresponding Korat generators. We should stress here some points discussed in Sec. 3. Korat builds data structures starting from a domain of graphs. In logic programming, on the contrary, terms are first-class objects, and graphs are represented through terms (see standard textbooks encodings). This is an advantage of logic programming, which allows us to choose the most adequate and simple primitive data structure. In contrast, Korat generates trees through the computationally expensive process of generating directed graphs and filtering the acyclic ones.

All the experiments were performed on an Intel Core 2 Duo E7300 2.66 GHz under the Linux operating system, using GNU Prolog 1.3.0 and SICStus Prolog 3.12.8, and the timings were collected using built-in CLP and Java statistics predicates.

In particular, the CLP timings were measured by exploring the whole search space through a call of the form gen_structure(Struct,p1,...,pk),fail for the parameter values of interest. The idea is to exploit the CLP backtracking mechanism to explore each success of gen_structure, while trying to succeed. For every success of the subgoal gen_structure(Struct,p1,...,pk), a concrete structure is generated. Since we are not interested in keeping all the structures in memory (which could be unmanageable), each structure is deleted as soon as it has been constructed. Due to the presence of the fail built-in, the whole goal fails, and, thus, the CLP system backtracks and tries to find another solution to gen_structure(Struct,p1,...,pk). The computation terminates after all the structures have been generated. At this stage, we are not yet addressing the issue of converting the CLP structures to proper Java objects. This issue will be briefly discussed in Sec. 5.

We selected two different CLP(FD) systems for running our CLP-based test generators: SICStus, for its diffusion and industrial strength, and GNUProlog, for its efficient compilation. We found that, in our experiments, GNUProlog outperforms SICStus, due to its efficient compilation. However, we chose to keep also the SICStus timings, because they revealed to be much more stable w.r.t. different encodings we experimented with (such as moving term comparison constraints from the head to the body). Therefore, SICStus seems to be more reliable in a setting where the user is not aware of the inner evaluation mechanism and cannot take advantage of it, while being still efficient.

In Fig. 4 we show the tables containing the timings for GNU Prolog, SICStus Prolog, and Korat, with a timeout of 200 seconds. The memory consumption of the CLP generators is negligible and grows very slowly on the size of the structures (as in Korat) so we did not report it.

Fig. 4: Generators performance evaluation.

Size	Sorted Lists	Time		
		GNU	SICStus	Korat
8	6435	0.00	0.01	0.61
9	24310	0.00	0.05	1.08
10	92378	0.02	0.17	1.83
11	352716	0.09	0.65	6.37
12	1352078	0.36	2.51	24.95
13	5200300	1.40	9.63	125.73
14	20058300	5.40	37.35	-
15	77558760	21.16	143.79	-
16	300540195	82.22	-	

Size	Heaparrays	Time		
		GNU	SICStus	Korat
6	13139	0.00	0.01	0.30
7	117562	0.03	0.15	0.86
8	1005075	0.17	1.27	3.41
9	10391382	1.66	12.65	34.103
10	111511015	17.57	134.26	-

Size	Search Trees	Time		
		GNU	SICStus	Korat
7	429	0.01	0.03	0.87
8	1430	0.02	0.11	4.43
9	4862	0.08	0.43	33.99
10	16796	0.28	1.67	-
11	58786	1.11	6.53	-
12	208012	4.43	25.42	-
13	742900	17.68	100.09	-
14	2674440	70.75	-	-

Size	Disjoint Sets	Time		
		GNU	SICStus	Korat
6	203	0.00	0.00	1.60
7	877	0.00	0.01	37.10
8	4140	0.01	0.06	-
9	21147	0.10	0.28	-
10	115975	0.61	1.58	-
11	678570	3.90	9.83	-
12	4213597	26.63	65.29	-
13	27644437	189.42	-	-

The results show that the CLP(FD)-based approach outperforms Korat in all the examples we considered. In some examples the CLP(FD)-based approach allowed us to explore a much larger input domain.

We did not explore different tunings of the CLP(FD)-solver, other than the default ones, which revealed to be already satisfactory. However, more complex problems (involving, for example, conditions based on minimization) may benefit of the many built-in predicates implementing more sophisticated solution search algorithms [27].

These promising results allow us to draw first conclusions on the validity of our CLP(FD)-based approach. The CLP(FD) encoding of generators requires no more ingenuity that the Korat encoding. On the contrary, we claim that correctness is a natural outcome of this approach and the programmer confidence in the developed generators is greatly increased. Furthermore, while being more declarative, this approach is also much more efficient and deserves to be further investigated for better integration into real-world testing frameworks.

5 Related Work and Conclusions

Constraint-based techniques have been widely used in the field of test case generation, since pioneering work in [10]. Early use of CLP for test generation can be found in the tool ATGen [28], developed for testing Spark ADA programs.

Several approaches using constraints are white-box and aim at the automatic extraction of CLP test generators from program source code, according to given coverage criteria. Moreover, most of them are not directly concerned with the efficiency of bounded-exhaustive generation of complex inputs.

In particular, in [9], white-box testing of an imperative language with pointers and heap is performed by symbolic execution of a small-step operational semantics in CLP, guided by coverage criteria. This approach can generate pointer-based data structures, at the expense of defining ad-hoc constraint solvers for the considered structures (mainly lists). Further work on white-box testing has been done in [6,17] for the generation of heap-allocated data structures, following a fixed coverage criteria for the choice of the test cases. The work in [16] presents a technique for white-box testing of object-oriented programming languages, which is more general and language independent than previous ones. Indeed, test case generators are obtained by partial evaluation of a language interpreter w.r.t. a given program.

The declarative approach has also been adopted by test generation tools such as Korat [29], which has been used in our experimental evaluation, UDITA [14], and TestEra [22]. These tools are quite efficient in practice but require careful implementations of clever ad-hoc backtracking mechanisms and search strategies, which are either built-in (like non-deterministic choice) or easily implementable in standard CLP systems. Lazy instantiation strategies in UDITA [14] can be seen as a particular CP solution strategy. Moreover, these tools are language-specific and they are not easily adaptable to other languages. Their integration with homogeneous but more general testing environments such as JPF [32] can be expensive and lead to suboptimal performance w.r.t. their original version [15].

In contrast, the proper interaction between a CLP test generator and, for example, a Java-based testing environment can be achieved either by using a bidirectional Java-Prolog interface, provided by most CLP systems, or by using an intermediate string representation of CLP data structures combined with Java's (de)serialization. In the latter case, for example, one could (i) generate XML encodings of CLP terms, and (ii) use libraries like XStream[6] or Simple[7] for constructing Java objects from XML. The problem of obtaining XML from CLP data structures can be solved, once and for all, by writing a single, universal, translator which constructs XML elements while recursively traversing a generic CLP term. We expect the CLP to Java translation to add negligible overhead. It should be noted, indeed, that such translation would be triggered only for structures which are of actual interest for testing (in contrast with the partial building of the Java objects and their destruction, if unsatisfactory, as in Korat [5]).

[6] http://xstream.codehaus.org

[7] http://simple.sourceforge.net

Efficiency aspects are considered in [33,34] where the process of generating the shape of the data structure is separated from that of generating proper values for data. In our approach, this separation is achieved transparently by following the constrain-and-generate programming approach of CLP, which prescribes the invocation of the instantiation mechanism only after all the constraints have been generated. Many unfeasible structures can, therefore, be eliminated at the symbolic level by constraint consistency checks.

In this paper we focus on showing that CLP can be used as a core component for efficient test case generators of complex input data. In contrast to some of the abovementioned techniques, our method does not start from source code and has been designed for black-box testing. It does not require the development of ad-hoc constraint solvers or search strategies, but it leverages commonly available CLP systems and libraries. However, strategies can easily be customized, if needed, and further LP instruments (some of which discussed in Sec. 3.2) are available to optimize the generators obtained according to our general scheme.

For example, in [7] it is shown that the use of BET for verifying large systems is feasible and provides effective results, but requires significant effort to be tuned and combined with abstraction techniques to reach the generation of useful test sets. For this purpose, our approach could easily benefit from decades of research on program analysis and abstract interpretation of constraint logic programs.

This work can be extended along several directions.

In [3] the Korat engine is modified to reduce the search space, by trying to skip structures which are in the same equivalence class of already considered structures. A similar issue is addressed in [4] for partition-based testing. Although we did not experiment on this subject (because the focus was on building all the structures) we believe that this optimization can be easily integrated in CLP by generating exactly one or a small set of witnesses per equivalence class.

There are several issues that deserve further study. Among these, we plan to explore the relationship between constraint solution strategies and test coverage criteria. Indeed, one may be interested into exploring the set of possible structures according to an ordering, parameterized by a given coverage criteria.

Furthermore, while in this paper we focused on model-based input generation only, we believe that the CLP approach can be successfully applied also for generating test oracles which can be used for verifying the post-conditions of the methods under test, since CLP generators can also be used as *acceptors*.

Regarding the application of program transformation and other optimization techniques, we plan to develop fully automatic optimization techniques tuned for this specific problem and for our CLP-based approach. A further application domain of program transformation is the automated extraction of test generators from (formal) specifications.

In conclusion, we believe that, due to its inherent symbolic execution mechanism, Constraint Logic Programming has a promising application field in test-case generation. CLP provides a highly declarative language and ensures efficiency by using dedicated constraint solvers and heuristics. On the basis of the results presented in this work we claim that correctness and efficiency of generators can take advantage from CLP-based techniques, especially in the case of complex input data.

Acknowledgements. We would like to thank Maurizio Proietti for many stimulating conversations. We would also like to thank the anonymous referees for their constructive comments on a preliminary version of this paper.

References

1. ICST 2009, Second International Conference on Software Testing Verification and Validation, 1-4 April 2009, Denver, Colorado, USA. IEEE Computer Society, 2009.
2. The MAP transformation system. `http://www.iasi.cnr.it/~proietti/system.html`, © 1995–2012.
3. N. Aguirre, V.S. Bengolea, M.F. Frias, and J.P. Galeotti. Incorporating coverage criteria in bounded exhaustive black box test generation of structural inputs. In M. Gogolla and B. Wolff, editors, TAP, volume 6706 of Lecture Notes in Computer Science, pages 15–32. Springer, 2011.
4. C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini. WS-TAXI: A WSDL-based testing tool for web services. In ICST [1], pages 326–335.
5. C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In ISSTA, pages 123–133, 2002.
6. F. Charreteur, B. Botella, and A. Gotlieb. Modelling dynamic memory management in constraint-based testing. Journal of Systems and Software, 82(11):1755–1766, 2009.
7. D. Coppit, J. Yang, S. Khurshid, W. Le, and K.J. Sullivan. Software assurance by bounded exhaustive testing. IEEE Trans. Software Eng., 31(4):328–339, 2005.
8. T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. Introduction to Algorithms (3. ed.). MIT Press, 2009.
9. F. Degrave, T. Schrijvers, and W. Vanhoof. Towards a framework for constraint-based test case generation. In D. De Schreye, editor, LOPSTR, volume 6037 of Lecture Notes in Computer Science, pages 128–142. Springer, 2009.
10. R.A. DeMillo and A.J. Offutt. Constraint-based automatic test data generation. IEEE Trans. Software Eng., 17(9):900–910, 1991.
11. A. Dovier, A. Formisano, and E. Pontelli. An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems. J. Exp. Theor. Artif. Intell., 21(2):79–121, 2009.
12. F. Fioravanti, A. Pettorossi, and M. Proietti. Transformation rules for locally stratified constraint logic programs. In M. Bruynooghe and K.-K. Lau, editors, Program Development in Computational Logic, volume 3049 of Lecture Notes in Computer Science, pages 291–339. Springer, 2004.
13. F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Program transformation for development, verification, and synthesis of programs. Intelligenza Artificiale, 5(1):119–125, 2011.
14. M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. In J. Kramer, J. Bishop, P.T. Devanbu, and S. Uchitel, editors, ICSE (1), pages 225–234. ACM, 2010.
15. M. Gligoric, T. Gvero, S. Lauterburg, D. Marinov, and S. Khurshid. Optimizing generation of object graphs in Java PathFinder. In ICST [1], pages 51–60.
16. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Test case generation for object-oriented imperative languages in CLP. TPLP, 10(4-6):659–674, 2010.

17. A. Gotlieb, B. Botella, and M. Rueher. A CLP framework for computing structural test data. In J.W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. Moniz Pereira, Y. Sagiv, and P.J. Stuckey, editors, Computational Logic, volume 1861 of Lecture Notes in Computer Science, pages 399–413. Springer, 2000.
18. D. Hoffman, H.-I. Wang, M. Chang, and D. Ly-Gagnon. Grammar based testing of HTML injection vulnerabilities in RSS feeds. In Proceedings of the 2009 Testing: Academic and Industrial Conference - Practice and Research Techniques, TAIC-PART '09, pages 105–110, Washington, DC, USA, 2009. IEEE Computer Society.
19. J.M. Howe and A. King. Efficient groundness analysis in prolog. Theory Pract. Log. Program., 3:95–124, January 2003.
20. J. Jaffar and M.J. Maher. Constraint logic programming: A survey. J. Log. Program., 19/20:503–581, 1994.
21. N.D. Jones, C.K. Gomard, and P. Sestoft. Partial evaluation and automatic program generation. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.
22. S.A. Khalek, G. Yang, L. Zhang, D. Marinov, and S. Khurshid. TestEra: A tool for testing Java programs using Alloy specifications. In P. Alexander, C.S. Pasareanu, and J.G. Hosking, editors, ASE, pages 608–611. IEEE, 2011.
23. J. Kriener and A. King. RedAlert: Determinacy Inference for Prolog. Theory and Practice of Logic Programming, 11(4-5):537–553, July 2011.
24. J.W. Lloyd. Foundations of Logic Programming, 2nd Edition. Springer, 1987.
25. D. Marinov. Automatic Testing of Software with Structurally Complex Inputs. PhD thesis, MIT, 2005.
26. D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. Technical report, MIT Computer Science and Artificial Intelligence Laboratory Report MIT -LCS-TR-921, 2003.
27. K. Marriott and P.J. Stuckey. Programming with constraints : an introduction. MIT Press, Cambridge, Mass., 1998.
28. C. Meudec. ATGen: automatic test data generation using constraint logic programming and symbolic execution. Software Testing, Verification and Reliability, 11(2):81–96, 2001.
29. A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid. Korat: A tool for generating structurally complex test inputs. In ICSE, pages 771–774. IEEE Computer Society, 2007.
30. M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. Theor. Comput. Sci., 142(1):89–124, 1995.
31. R.W. Robinson. Counting unlabeled acyclic digraphs. In Charles Little, editor, Combinatorial Mathematics V, volume 622 of Lecture Notes in Mathematics, pages 28–43. Springer Berlin / Heidelberg, 1977. 10.1007/BFb0069178.
32. W. Visser, C.S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In G.S. Avrunin and G. Rothermel, editors, ISSTA, pages 97–107. ACM, 2004.
33. S. Visvanathan and N. Gupta. Generating test data for functions with pointer inputs. In ASE, pages 149–. IEEE Computer Society, 2002.
34. R. Zhao and Q. Li. Automatic test generation for dynamic data structures. In SERA, pages 545–549. IEEE Computer Society, 2007.

A The Korat Red-Black Tree invariant

```
public boolean repOK() {
      if (!repOkAcyclic()) return false;
      if (!repOkColors())  return false;
      return repOkKeysAndValues();
  }
  private boolean repOkAcyclic() {
      ...
  }
  private boolean repOkColors() {
      // RedHasOnlyBlackChildren
      java.util.LinkedList workList = new java.util.LinkedList();
      workList.add(root);
      while (!workList.isEmpty()) {
          Node current = (Node) workList.removeFirst();
          Node cl = current.left;
          Node cr = current.right;
          if (current.color == RED) {
              if (cl != null && cl.color == RED) return debug("failure");
              if (cr != null && cr.color == RED) return debug("failure");
          }
          if (cl != null)  workList.add(cl);
          if (cr != null)  workList.add(cr);
      }

      // PathsFromRootToNILHaveSameNumberOfBlackNodes
      int numberOfBlack = -1;
      workList = new java.util.LinkedList();
      workList.add(new Pair(root, 0));
      while (!workList.isEmpty()) {
          Pair p = (Pair) 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 debug("failure");
          } else {
              workList.add(new Pair(e.left, n));
              workList.add(new Pair(e.right, n));
          }
      }
      return true;
  }
  private boolean repOkKeysAndValues() {
      if (!orderedKeys(root, null, null)) return debug("failure");
      return true;
  }
  private boolean orderedKeys(Node e, Object min, Object max) {
      if (((min != null) && (compare(e.key, min) <= 0))
              || ((max != null) && (compare(e.key, max) >= 0)))
          return false;
      if (e.left != null)
          if (!orderedKeys(e.left, min, e.key))  return false;
      if (e.right != null)
          if (!orderedKeys(e.right, e.key, max)) return false;
      return true;
  }
```

The `repOK()` method specifying the Red-Black Trees invariant, and some auxiliary methods.

B Experiments

```
sortedlist(A,Size,MaxNat) :-
    % Preamble
      L#>=0, Size#>L, fd_labeling([L]),
      varlist(L,A), fd_domain(A,1,MaxNat),
    % Symbolic Definition
      sorted(A),
    % Instantiation
      fd_labeling(A).

sorted([]).
sorted([_]).
sorted([A,B|Rest]) :- B#>=A, sorted([B|Rest]).
```

Size	Lists	Time		
		GNU	SICStus	Korat
8	6435	0.00	0.01	0.61
9	24310	0.00	0.05	1.08
10	92378	0.02	0.17	1.83
11	352716	0.09	0.65	6.37
12	1352078	0.36	2.51	24.95
13	5200300	1.40	9.63	125.73
14	20058300	5.40	37.35	-
15	77558760	21.16	143.79	-
16	300540195	82.22	-	

```
heaparray(A,S,MaxNat) :-
    % Preamble
      L#>=0, S#>=L, fd_labeling([L]), varlist(L,H), fd_domain(H,0,MaxNat), RS#=S-L,
      pad_rest(Rest,RS), append(H,Rest,A),
    % Symbolic Definition
      heap(H),
    % Instantiation
      fd_labeling(H).

heap([]).
heap([R|H]) :- heap_c(H,[R|H]).

heap_c([],_).
heap_c([C],[F|_])        :- F#>=C.
heap_c([LC,RC|Cs],[F|H]) :- F#>=LC, F#>=RC, heap_c(Cs,H).

pad_rest([],_).
pad_rest([-1|Rest],S) :- S#>0, T#=S-1, pad_rest(Rest,T).
```

Size	Heaps	Time		
		GNU	SICStus	Korat
6	13139	0.00	0.01	0.30
7	117562	0.03	0.15	0.86
8	1005075	0.17	1.27	3.41
9	10391382	1.66	12.65	34.103
10	111511015	17.57	134.26	390.96
11	-	232.01	-	-

```
searchtree(T,S,MaxNat) :-
    % Preamble
      varlist(S,Vs), Max #= MaxNat-1,
      fd_domain(Vs,0,Max), all_different(Vs),
    % Symbolic Definition
      bt(T,S,Vs,_),
      ordered(T,0,MaxNat),
    % Instantiation
      fd_labeling(Vs).

bt(t(e,K,e),S,[K|Ks], Ks) :- S#=1.
bt(t(e,K,R),S,[K|Ks],NKs) :- S#>=2, NS#=S-1, bt(R,NS,Ks,NKs).
bt(t(L,K,e),S,[K|Ks],NKs) :- S#>=2, NS#=S-1, bt(L,NS,Ks,NKs).
bt(t(L,K,R),S,[K|Ks],NKs) :- S#>=3, NS#=S-1, SL#>0, SR#>0, fdsum(NS,SL,SR),
                             bt(L,SL, Ks,TKs), bt(R,SR,TKs,NKs).
```

Size	Trees	Time		
		GNU	SICStus	Korat
7	429	0.01	0.03	0.87
8	1430	0.02	0.11	4.43
9	4862	0.08	0.43	33.99
10	16796	0.28	1.67	234.16
11	58786	1.11	6.53	-
12	208012	4.43	25.42	-
13	742900	17.68	100.09	-
14	2674440	70.75	395.36	-

The ordered/2 predicate has been defined in Sec. 3.

```
disjset(DS,S,MaxNat) :-
    % Preamble
      varlist(S,Set), Max #= MaxNat-1,
      fd_domain(Set,0,Max),
      fd_all_different(Set),
    % Symbolic Definition
      disjset(DS,S,Set,_),
      heads(DS,Hs),
      sorted(Hs),
    % Instantiation
      fd_labeling(Set).

disjset([],0,_Set,_Rest).
disjset([P|Ps],S,Set,Rest) :- S1#>0, S1#=<S, S2#=S-S1, fd_labeling([S1]),
    length(P,S1), sorted(P), append(P,Rest,Set), disjset(Ps,S2,Rest,_).

heads([],[]).
heads([[H|_]|Ls],[H|L]) :- heads(Ls,L).
```

Size	Sets	Time		
		GNU	SICStus	Korat
6	203	0.00	0.00	1.60
7	877	0.00	0.01	37.10
8	4140	0.01	0.06	1153.08
9	21147	0.10	0.28	-
10	115975	0.61	1.58	-
11	678570	3.90	9.83	-
12	-	26.63	65.29	-
13	-	189.42	-	-

The sorted/1 predicate has been defined in the sortedlist example above.

```
% Auxiliary predicates
varlist(0,[]).
varlist(N,[_|L]) :- N#>0, M#=N-1, varlist(M,L).

fdsum(Sum,X,Y) :- fd_domain([X],0,Sum), fd_labeling([X]), Y#=Sum-X.
```

C Partially evaluated Red-Black Tree generator

```
% Binary Trees

lbt(                                   e,S,    Ks, Ks) :- S#=0.
lbt(                          t(_,e,K,e),S,[K|Ks], Ks) :- S#=1.
lbt(             t(_,e,K,t(Cr,Lr,Kr,Rr)),S,[K|Ks],NKs) :- S#>=2, SR#=S-1, lbt(t(Cr,Lr,Kr,Rr),SR,Ks,NKs).
lbt(             t(_,t(Cl,Ll,Kl,Rl),K,e),S,[K|Ks],NKs) :- S#>=2, SL#=S-1, lbt(t(Cl,Ll,Kl,Rl),SL,Ks,NKs).
lbt(t(_,t(Cl,Ll,Kl,Rl),K,t(Cr,Lr,Kr,Rr)),S,[K|Ks],NKs) :- S#>=3, N#=S-1, SL#>0, SR#>0, fdsum(N,SL,SR),
                                                          lbt(t(Cl,Ll,Kl,Rl),SL,Ks,TKs),
                                                          lbt(t(Cr,Lr,Kr,Rr),SR,TKs,NKs).

% Color Invariant

ci(                                   e,    Cs, Cs).
ci(                          t(C,e,_,e),[C|Cs], Cs).
ci(             t(C,e,_,t(Cr,Lr,Kr,Rr)),[C|Cs],NCs) :- C+Cr#>0, ci(t(Cr,Lr,Kr,Rr),Cs,NCs).
ci(             t(C,t(Cl,Ll,Kl,Rl),_,e),[C|Cs],NCs) :- C+Cl#>0, ci(t(Cl,Ll,Kl,Rl),Cs,NCs).
ci(t(C,t(Cl,Ll,Kl,Rl),_,t(Cr,Lr,Kr,Rr)),[C|Cs],NCs) :- C+Cl#>0, C+Cr#>0,
                                                       ci(t(Cl,Ll,Kl,Rl),Cs,TCs),
                                                       ci(t(Cr,Lr,Kr,Rr),TCs,NCs).

% Paths Invariant

pi(                                   e,0).
pi(                          t(C,e,_,e),D) :- D#=C.
pi(             t(C,e,_,t(Cr,Lr,Kr,Rr)),D) :- D#=C, pi(t(Cr,Lr,Kr,Rr),0).
pi(             t(C,t(Cl,Ll,Kl,Rl),_,e),D) :- D#=C, pi(t(Cl,Ll,Kl,Rl),0).
pi(t(C,t(Cl,Ll,Kl,Rl),_,t(Cr,Lr,Kr,Rr)),D) :- ND#>=0, D#=ND+C,
                                              pi(t(Cl,Ll,Kl,Rl),ND),
                                              pi(t(Cr,Lr,Kr,Rr),ND).

% Ordering

ordered(                                   e,  _,  _).
ordered(                          t(_,e,K,e),Min,Max) :- Min#=<K, K#<Max.
ordered(                t(_,e,K,t(C,L,Kr,R)),Min,Max) :- Min#=<K, K#<Max, NK#=K+1, ordered(t(C,L,Kr,R),NK,Max).
ordered(                t(_,t(C,L,Kl,R),K,e),Min,Max) :- Min#=<K, K#<Max, ordered(t(C,L,Kl,R),Min,K).
ordered(t(_,t(Cl,Ll,Kl,Rl),K,t(Cr,Lr,Kr,Rr)),Min,Max) :- Min#=<K, K#<Max, NK#=K+1,
                                                         ordered(t(Cl,Ll,Kl,Rl),Min,K),
                                                         ordered(t(Cr,Lr,Kr,Rr),NK,Max).
```

D The Korat code for Disjoint Sets

```
public class DisjSet2 {
    // helper class
    public static class Record {
        public int parent;
        public int rank;

        public Record() {
        }

        public Record(Record rec) {
            parent = rec.parent;
            rank = rec.rank;
        }
    }// end of helper class
    private Record[] elements;
    private int size;

    public boolean allDifferent() {
        int n = size - 1;
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j <= n; j++) {
             // all different
                if (elements[i].rank == elements[j].rank)
                    return false;

                // If two elements have the same parent, then one of them is the head of the list.
                // This condition ensures that we have lists, not trees.
                if ( (elements[i].parent == elements[j].parent)
                && !(elements[i].parent == i || elements[i].parent == j) )
                    return false;
            }
        }
        return true;
    }

    // methods used by Korat
    public boolean repOK() {

        if (elements.length != size)
            return false;

        for (int i = 0; i < size; i++) {
            int parentID = elements[i].parent;
            if (parentID < 0 || parentID >= size)
                return false;

        // records are sorted by rank,  in decreasing order
            if ((i<size-1) && !(elements[i].rank >= elements[i+1].rank))
            return false;

            // elements of the lists are sorted in decreasing order
            if (parentID != i) {
                int parentRank = elements[parentID].rank;
                if (parentRank <= elements[i].rank)
                    return false;
            }
        }

        if (!allDifferent())
            return false;
        return true;
    }

    public static IFinitization finDisjSet2(int size) {
        IFinitization f = FinitizationFactory.create(DisjSet2.class);

        IClassDomain bindingsCD = f.createClassDomain(Record.class, size);
        IObjSet bindings = f.createObjSet(Record.class);
        bindings.addClassDomain(bindingsCD);

        IIntSet lens = f.createIntSet(0, size);
        IArraySet elems = f.createArraySet(Record[].class, lens, bindings, 1);

        f.set("size", f.createIntSet(size, size));
        f.set(Record.class, "parent", f.createIntSet(0, size - 1));
        f.set(Record.class, "rank", f.createIntSet(0, size - 1));
        f.set("elements", elems);

        return f;
    }
}
```