Efficient Generation of Test Data Structures using Constraint Logic Programming and Program Transformation

Fabio Fioravanti[1], Maurizio Proietti[2], and Valerio Senni[3]⋆

[1] Dipartimento di Scienze, University ‘G. D’Annunzio’, Pescara, Italy
fioravanti@sci.unich.it

[2] IASI-CNR, Roma, Italy
maurizio.proietti@iasi.cnr.it

[3] IMT Institute for Advanced Studies, Lucca, Italy
valerio.senni@imtlucca.it

Abstract. The goal of Bounded-Exhaustive Testing (BET) is the automatic generation of *all* test cases satisfying a given invariant, within a given size bound. When the test cases have 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 structures.

We follow a declarative approach which allows us to separate the issue of (i) defining the test data structure in terms of its properties, from that of (ii) efficiently generating data structure instances. This separation helps establish the correctness of the developed test case generators. We rely on a symbolic representation and we take advantage of efficient search strategies provided by CLP systems for generating test instances.

Through a running example taken from the literature on BET, we illustrate our test generation framework and we show that CLP allows us to develop easily understandable and efficient test generators.

Additionally, we propose a program transformation technique whose goal is to make the evaluation of these CLP-based generators much more efficient and we demonstrate its effectiveness on a number of complex test data structures.

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 it 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 cost-effective way, as well as guaranteeing the correctness of test case generators.

⋆ Address for correspondence: Valerio Senni, IMT Institute for Advanced Studies, Piazza San Ponziano 6, 55100 Lucca, Italy, valerio.senni@imtlucca.it

In this paper we focus on the *bounded-exhaustive testing* [9] approach (BET), whose goal is to test a program on *all* input data 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 present, are likely to appear already in small-sized instances of the inputs.

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.

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 structures. We adopt a declarative approach which allows us to separate the issue of (i) defining the test data structure in terms of its properties, from that of (ii) efficiently generating all data structure instances that comply with the given definition. This separation helps establish the correctness of the developed test case generators, because it lets testing engineers specify *what* to generate, in a very modular and easily understandable way. General purpose CLP search strategies are then used for realizing *how* test instances are generated. However, declarativeness is often deemed to be paid in terms of inefficiency. We show that this is not the case for CLP-based test generation, and that very good results can already be 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, computing the structural and invariant constraints first, allowing to prune the search space at the symbolic level, and postponing as much as possible the actual (expensive) generation of test instances. Our experimental evaluation in [35] demonstrates the effectiveness of the CLP-based approach, for the construction of efficient and correct test generators.

Despite the good performance of declarative CLP-based test case generators, the problem of generating large sets of complex data structures remains very challenging, due to the inherent combinatorial nature of the problem. Therefore, to obtain significantly more efficient CLP-based generators, we propose in this paper an optimization technique based on program transformation [7,13,14,36].

This optimization technique is inspired by the simple consideration that left-to-right scheduling of constraints and atoms implemented by standard CLP systems enforces the full generation of a data structure before the evaluation of a filter can start, which can be very inefficient. A possible solution is to adopt a more efficient, problem specific, scheduling strategy, as it happens in dynamic scheduling via *delay declarations* [31], at the cost of a strong overhead on the evaluation mechanism.

In this paper we follow a different approach: we apply program transformation to a given test case generator and we obtain a transformed program such that (i) it computes the same test cases as the original generator, and (ii) under the default left-to-right scheduling it behaves like the original generator does when using a more efficient scheduling strategy, thus avoiding any evaluation overhead. The underlying idea is that filters should be applied as soon as a data structure has been partially generated, hence determining an early pruning of the search space.

In Section 2, we formalize our CLP-based test generation approach and we illustrate its expressiveness by providing a clean and declarative definition of a Red-Black tree generator. In Section 3 we present our optimization technique based on CLP program transformation, for obtaining faster generators. Finally, in Section 4, we carry on an evaluation of the proposed CLP-based approach by comparing its performance with that of a state-of-the-art Java-based tool called Korat, and by showing the improvements obtained through our optimization technique on a number of CLP-based test generators.

2 The CLP-based Approach

Among several applications, CLP has been shown to be well suited for encoding and solving combinatorial problems [28]. In this section we illustrate how to formulate test generation problems as CLP programs and queries, and how to exploit the evaluation mechanism of CLP to solve them efficiently. We start by recalling the CLP framework, with a special attention to its operational semantics. For missing details we refer the reader to [28].

2.1 Preliminaries

We consider a typed first order language [26] with two types: $\mathbb{D}$ denoting the domain of the constraints, and $\mathbb{T}$ denoting finite trees of elements in $\mathbb{D}$. Let $\Sigma_{\mathcal{C}} = \langle \mathcal{V}_{\mathcal{C}}, \mathcal{F}_{\mathcal{C}}, \Pi_{\mathcal{C}} \rangle$ be a logic language signature, where $\mathcal{V}_{\mathcal{C}}$ is a denumerable set of variables, $\mathcal{F}_{\mathcal{C}}$ is a denumerable set of function symbols, and $\Pi_{\mathcal{C}}$ is a finite set of predicate symbols. An *atomic constraint* is an atomic formula over $\Sigma_{\mathcal{C}}$. A *constraint* is a finite conjunction of atomic constraints. Terms and constraints are typed according to the following rules: (1) variables in $\mathcal{V}_{\mathcal{C}}$ are typed $\mathbb{D}$, (2) function symbols of arity $k \geq 0$ in $\mathcal{F}_{\mathcal{C}}$ are typed $\mathbb{D}^k \rightarrow \mathbb{D}$, and (3) predicate symbols of arity $k \geq 0$ in $\Pi_{\mathcal{C}}$ are typed $\mathbb{D}^k$. The constraint interpretation $\mathcal{D}$ is a mathematical structure with domain $\mathbb{D}$, and provides a fixed interpretation for $\Sigma_{\mathcal{C}}$. A constraint solver for $\mathcal{D}$ provides a function $solv(c)$ that maps a constraint c to either *true*, *false*, or *unknown* and such that: (i) if $solv(c) = true$ then $\mathcal{D} \models \exists c$ (that is, c is satisfiable) and (ii) if $solv(c) = false$ then $\mathcal{D} \models \neg \exists c$ (that is, c is unsatisfiable). The solver is *complete* if, for any given constraint, it returns either *true* or *false*. One can adopt an incomplete solver for efficiency reasons. Such a solver, for example, can be used to eliminate a number of *false* constraints, while the discovery of *true* constraints among a restricted set is performed by using a complete solver at a later stage.

CLP($\mathcal{D}$) programs are defined on a signature $\Sigma = \langle \mathcal{V}_{\mathcal{T}} \cup \mathcal{V}_{\mathcal{C}}, \mathcal{F}_{\mathcal{T}} \cup \mathcal{F}_{\mathcal{C}}, \Pi_{\mathcal{U}} \cup \Pi_{\mathcal{C}} \rangle$, where $\mathcal{V}_{\mathcal{T}}$ is a denumerable set of variables, $\mathcal{F}_{\mathcal{T}}$ is a denumerable set of function symbols (i.e., tree constructors), $\Pi_{\mathcal{U}}$ is a finite set of (user-defined) predicate symbols. Terms and atoms are constructed complying with the usual typing rules, including the following ones: (1) variables in $\mathcal{V}_{\mathcal{T}}$ are typed $\mathbb{T}$, (2) function symbols of arity $k \geq 0$ in $\mathcal{F}_{\mathcal{T}}$ are typed $\mathbb{A}^k \rightarrow \mathbb{T}$, where $\mathbb{A} \in \{\mathbb{D}, \mathbb{T}\}$, and (3) predicate symbols of arity $k \geq 0$ in $\Pi_{\mathcal{U}}$ are typed $\mathbb{A}^k$, where $\mathbb{A} \in \{\mathbb{D}, \mathbb{T}\}$. *Atoms* are of the form $p(s_1, \ldots, s_m)$, where p is a predicate symbol in $\Pi_{\mathcal{U}}$ and $s_1, \ldots, s_m$ are terms of appropriate type. A *goal* is a finite conjunction $L_1, \ldots, L_m$ where L_i is either an atomic constraint or an atom. The empty goal is denoted by $\Box$. A CLP *program* P over Σ is a finite set of *clauses* of the form H `:-` G`.` where H is an atom and G is a goal. We have two extra assumptions on programs: (i) symbols in $\mathcal{F}_{\mathcal{C}}$ appear only within the constraints, and (ii) occurrences of $\mathcal{V}_{\mathcal{C}}$-variables in the head of a clause are all *distinct*. These two assumptions can be easily satisfied whenever the theory of constraints includes an equality predicate (which happens in most theories). The reason for these assumptions is to avoid (unsound) unifications between terms of type $\mathbb{D}$. The semantics of a CLP program P is given in terms of its least $\mathcal{D}$-model $M(P)$ [28].

A CLP system computes the answers to a user *query* of the form `:-` G`.` against a program P, where G is a goal. The evaluation of a query is performed by constructing a so-called LD-derivation, which is a (possibly infinite) sequence of *states*. In the following we will omit the prefix LD- and we will simply talk about derivations. A *state* is a pair $\langle G \,|\, c\rangle$, where G is a goal $L_1, \ldots, L_n$ and c is a constraint, called *store*. Given a program P and a state $\langle G \,|\, c\rangle$, a *derivation step* $\langle G \,|\, c\rangle \rightarrow_{\sigma} \langle G' \,|\, c'\rangle$ is performed by selecting the *leftmost* (atomic) conjunct L_1 of G and rewriting the state according to the following two cases: (1) L_1 is an atomic constraint, then σ is the identity substitution, c' is $L_1 \wedge c$ and if $solv(c') = \mathit{false}$ then G' is the empty goal, otherwise G' is $L_2, \ldots, L_n$, and (2) L_1 is a user defined atom, then (2.1) if there exists a clause H `:-` B`.` in P such that σ is the most general unifier of L_1 and H, c' is $c\sigma$ and G' is $(B, L_2, \ldots, L_n)\sigma$, and (2.2) if there is no such clause, c' is *false* and G' is the empty goal. A *derivation* in P starting in the state $\langle G_0 \,|\, c_0\rangle$ is a sequence $\langle G_0 \,|\, c_0\rangle \rightarrow_{\sigma_1} \langle G_1 \,|\, c_1\rangle \rightarrow_{\sigma_2} \langle G_2 \,|\, c_2\rangle \rightarrow_{\sigma_3} \ldots$ of states such that $\langle G_i \,|\, c_i\rangle \rightarrow_{\sigma_{i+1}} \langle G_{i+1} \,|\, c_{i+1}\rangle$, for $i \geq 0$, is a derivation step. If G is a goal, then a derivation in P for G is a derivation in P starting in the state $\langle G \,|\, \mathit{true}\rangle$. A state $\langle \Box \,|\, c\rangle$ is a *failure* state if $solv(c) = \mathit{false}$ and it is a *success* state otherwise. A *finite* derivation $\delta : \langle G \,|\, \mathit{true}\rangle \rightarrow_{\sigma_1} \langle G_1 \,|\, c_1\rangle \rightarrow_{\sigma_2} \ldots \rightarrow_{\sigma_k} \langle G_k \,|\, c_k\rangle$ for the goal G is *failed* if $\langle G_k \,|\, c_k\rangle$ is a failure state and it is *successful* if $\langle G_k \,|\, c_k\rangle$ is a success state. If δ is successful, then $(\sigma_1 \cdots \sigma_k, c_k)$ is an *answer* to the query `:-`G.. Clearly, depending on the choice of the clause used in case (2) of the derivation step, there can be several (possibly infinite) derivations in P for a given goal. Let us define the *derivation tree* in P for the query Q of the form `:-` G`.` as the set of all possible derivations for the goal G. We denote by $Ans_P(Q)$ the set of all answers in P to the query Q.

For reasons of simplicity, in this paper we focus on Constraint Logic Programming over Finite Domains (CLP(FD) [28]) but the test generation framework as well as the optimization technique we propose and the related results do not rely on this assumption and can be applied also if we consider logic programs that combine constraints on finite and infinite domains. Clearly, when considering infinite domains, one needs to rely on a finite (symbolic) presentation or on criteria to select finitely many values out of infinitely many.

In CLP(FD) we have that: (1) $\mathcal{V}_{\mathcal{C}}$ is a set of variables ranging over a finite integer domain, (2) $\mathcal{F}_{\mathcal{C}} = \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{min}, \texttt{max}\} \cup \mathbb{Z}$, where $\mathbb{Z}$ is the set of the integer numbers, and (3) $\Pi_{\mathcal{C}} = \{\texttt{\#=}, \texttt{\#\textbackslash=}, \texttt{\#>}, \texttt{\#>=}, \texttt{\#<}, \texttt{\#=<}\}$. 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 fixed interpretation for $\Sigma_{\mathcal{C}}$ is the structure of the integers $\mathcal{Z}$.

The solver over $\Sigma_{\mathcal{C}}$ of most CLP(FD) systems (such as SICStus [3] and GNU Prolog [2]) is *incomplete*, which means that those systems may produce successful derivations that terminate on a state having an unsatisfiable store. Most systems provide the following additional predicates: `domain(Vs,Min,Max)`[1], that constrains all the variables in the list `Vs` to range over $[\texttt{Min}, \ldots, \texttt{Max}] \subset \mathbb{Z}$, and `labeling(Settings,Vs)`[1], that implements a *complete* solver for FD-constraints and binds the variables in `Vs` to ground values such that the current constraint store is satisfied (the optional `Settings` argument can be used for configuring the search process).

We are interested into characterizing terms manipulated by CLP-based test generators in terms of classes of trees. In particular, we introduce the set Υ of the trees constructed over $(\mathcal{V}_{\mathcal{C}}, \mathcal{F}_{\mathcal{T}} \cup \mathcal{F}_{\mathcal{C}})$ (no variable of type $\mathbb{T}$ allowed) and with at least one symbol in $\mathcal{F}_{\mathcal{T}}$. We also define the set $\overline{\Upsilon}$ of ground trees, that is, the set of those trees in Υ with no occurrences of variables in $\mathcal{V}_{\mathcal{C}}$. Note that $\mathbb{D} \cap \Upsilon = \emptyset$.

A *mode* m_p for a predicate p of arity n is a function $\{1, \ldots, n\} \rightarrow \{\texttt{+}, \texttt{-}, \texttt{?}\}$ such that if position i has type $\mathbb{T}$ then $m_p(i) \in \{\texttt{+}, \texttt{-}\}$, otherwise $m_p(i) = \texttt{?}$. If $m_p(i) = \texttt{+}$ then we call the i-th argument an *input position*, otherwise we call it an *output position*. Constraint predicates have mode `(?,?)`. A program is *moded* if each user-defined predicate has a mode. From now on we will assume all programs are moded. An atom is *correctly input-typed* (resp. *output-typed*) if its input (resp. output) positions are filled in by terms in Υ.

We now introduce the notion of well-modedness, which is a slight modification of that in [4], where modes distinguish ground and non-ground arguments. To simplify the notation, in the following we write $p(\mathbf{i}, \mathbf{o})$ to indicate that $\mathbf{i}$ is the sequence of the input arguments of p, and $\mathbf{o}$ is the sequence of the output arguments of p. Moreover, we indicate by $\mathit{Vars}_{\mathcal{T}}(t)$ the set of variables of type $\mathbb{T}$ occurring in the term t.

Definition 1 (Well-modedness). *A clause C is* well-moded *if the clause obtained from C by dropping all constraints is of the form*

$p_0(\mathbf{t}_0, \mathbf{s}_{n+1})$ `:-` $p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$. *and, for $i \in \{1, \ldots, n+1\}$,*
$\mathit{Vars}_{\mathcal{T}}(\mathbf{s}_i) \subseteq \bigcup_{j=0}^{i-1} \mathit{Vars}_{\mathcal{T}}(\mathbf{t}_j)$.

A goal G (query `:-` *G.) is well-moded if the clause q* `:-` *G. is well-moded. A program is well-moded if all of its clauses are well-moded.*

Note that, if a goal is well-moded, then its leftmost atom is correctly input-typed. Similarly to [4] for the case of (pure) logic programs, we can now state a result of preservation of well-modedness and correct input/output typing of goals.

Proposition 1. *Let P be a well-moded program and* `:-` *G. a well-moded query. For every state $\langle G' \mid c\rangle$ in a derivation δ for* `:-` *G. in P: (1) the goal G' is well-moded and correctly input-typed, and (2) if δ is successful and (ϑ, c) is the answer computed by δ, then $G\vartheta$ is correctly output-typed.*

[1] Here we adopt the SICStus Prolog syntax [3].

2.2 CLP(FD)-based test case generation

Our method for developing test case generators takes advantage of the CLP(FD) programming paradigm, sometimes referred to as constrain-and-generate [28]. It follows a symbolic approach and it is structured mainly in two phases, as follows.

In a first phase we construct a so-called *shape* of the desired data structure (a term in Υ), capturing some basic structural properties (such as being a binary tree or a list). We also leave several parts of the data structure unspecified, by using variables in $\mathcal{V}_\mathcal{C}$ as place-holders. Then, we add to those $\mathcal{V}_\mathcal{C}$ variables constraints deriving from the invariants that define the data structure and we obtain a so-called *constrained shape*. In this phase constraints are checked for consistency using the built-in (incomplete) solver (*constrain* phase). The constrained shapes found inconsistent are rejected at this early stage. Since one of these shapes may represent a large number of instances, we have a great advantage in evaluating them at this symbolic stage.

In a second phase, we invoke a complete solver to find the constrained shapes that can effectively be instantiated to a concrete data structure (we call these *feasible* structures) and we compute all the correct instantiations (i.e., terms in $\overline{\Upsilon}$), which we call the set of the *test cases* (*generate* phase).

An additional feature of our approach is the declarativeness of test case generators, that allows us to define a desired structure through several steps. Firstly, by providing the definition of the search domain specified by a so-called *generator*, which constructs a set of shapes. Then, by providing the definition (over that domain) of one or more *filters*, encoding the invariants these shapes must satisfy.

We define a template (τ), based on the constrain-and-generate paradigm, for developing (filter-based) test case generators. In the following, for the sake of conciseness, whenever possible we use a shorter notation for modes and types: we use m^h to denote a sequence of $h \geq 0$ inputs of a given mode m, for $m \in \{\texttt{+}, \texttt{-}, \texttt{?}\}$, and t^h to denote the Cartesian product of h copies of type $t \in \{\mathbb{D}, \mathbb{T}\}$.

The template is divided into three parts: (1) the `Preamble` contains the definition of the lists of $\mathcal{V}_\mathcal{C}$-variables and their domains, (2) the `Symbolic Definition` contains: (i) a call to a generator which defines the data structure shapes (e.g. list, tree, graph), and (ii) a sequence of calls to filters defining the invariants in terms of constraints among the $\mathcal{V}_\mathcal{C}$-variables, and (3) the `Instantiation` contains the calls to the complete solver to instantiate the $\mathcal{V}_\mathcal{C}$-variables.

```
      tc(T,P_1,...,P_h) :-
          % Preamble                        (constrain)
            init(P_1,...,P_h,V_1,...,V_k),  % definition of domains
                                            % of the FD-variables
          % Symbolic Definition             (constrain)
(τ)         g(T,P_1,...,P_h,V_1,...,V_k),   % shape
            inv_1(T,P_1,...,P_h), ...,      % invariants
            inv_n(T,P_1,...,P_h),
          % Instantiation                   (generate)
            labeling(V_1),...,
            labeling(V_k).
```

where we assume that: (1) the predicate tc has type $\mathbb{T} \times \mathbb{D}^h$ and mode $(\texttt{-},\texttt{?}^h)$, (2) the predicate init has type $\mathbb{D}^h \times \mathbb{T}^k$ and mode $(\texttt{?}^h,\texttt{-}^k)$, (3) the predicate g has type $\mathbb{T} \times \mathbb{D}^h \times \mathbb{T}^k$ and mode $(\texttt{-},\texttt{?}^h,\texttt{+}^k)$, (4) the predicates $\texttt{inv}_1, \ldots, \texttt{inv}_n$ have type $\mathbb{T} \times \mathbb{D}^h$ and mode $(\texttt{+},\texttt{?}^h)$, and (5) the predicate labeling has type $\mathbb{T}$ and mode (+). The init predicate has the role of constructing the finite lists $\texttt{V}_1$,...,$\texttt{V}_k$ of $\mathcal{V}_\mathcal{C}$-variables and fixing their domain, depending on the values of the parameters $\texttt{P}_1$,...,$\texttt{P}_h$. Note that any instance of the template clause satisfying the given types and modes is well-moded.

Besides typing and moding, we expect the predicate g to be a generator and the predicates $\texttt{inv}_1, \ldots, \texttt{inv}_n$ to be filters, according to the following definitions.

Definition 2 (Generator). *Given a well-moded program P, a* generator *is a user-defined predicate* g *of type $\mathbb{T} \times \mathbb{D}^h \times \mathbb{T}^k$ and mode $(\texttt{-},\texttt{?}^h,\texttt{+}^k)$ such that, for any well-moded query Q of the form* :- g(T,p1,...,ph,v1,...,vk).*, if* $\texttt{p1},\ldots,\texttt{ph} \in \mathbb{Z}$ *then $Ans_P(Q)$ is finite.*

Note that, in Def. 2, for any answer (σ, c) to the well-moded query Q, the term $\texttt{T}\sigma$ is a tree, that is $\texttt{T}\sigma \in \varUpsilon$. Therefore, for any given values $\texttt{p1},\ldots,\texttt{ph} \in \mathbb{Z}$ of its parameters, a generator defines a finite set of shapes, that is, a finite search domain.

Definition 3 (Filter). *Given a well-moded program P, a* filter *is a user-defined predicate* inv *of type $\mathbb{T} \times \mathbb{D}^h$ and mode $(\texttt{+},\texttt{?}^h)$ such that, for any well-moded query Q of the form* :- inv(t,p1,...,ph).*, $Ans_P(Q)$ is finite.*

Note that, in Def. 3, by the well-modedness assumption, t is in $\varUpsilon$ and any answer in $Ans_P(Q)$ is of the form (ε, c), where ε is the identity substitution. For any given input shape t, a filter computes a finite set of constrained shapes.

Definition 4 (Test case Generator). *A test case generator is a well-moded program P defining a predicate* tc *by a clause which is an instance of the template (τ) satisfying mode and type requirements, and such that the predicate for* g *is a generator and the predicates for $\texttt{inv}_1, \ldots, \texttt{inv}_n$ are filters.*

Finally, we define the set of the test cases as follows.

Definition 5 (Test Cases). *Given a test generator program P defining* tc*, the set of the test cases is the set $T = Ans_P(Q)$, for any well-moded query Q of the form* :- tc(T,p1,...,ph). *and parameters* $\texttt{p1},\ldots,\texttt{ph} \in \mathbb{Z}$.

The test cases are elements of $\overline{\varUpsilon}$, as stated by the following proposition.

Proposition 2. *Let $T = Ans_P(Q)$ be a set of test cases, for a given well-moded query Q of the form* :- tc(T,p1,...,ph). *and parameters* $\texttt{p1},\ldots,\texttt{ph} \in \mathbb{Z}$*. An answer in T is of the form $(\sigma, true)$ and $\texttt{T}\sigma \in \overline{\varUpsilon}$.*

For any answer in T, the ground term $\texttt{T}\sigma$ represents the test case/data structure of interest.

In order to better evaluate our approach and the advantage of the symbolic evaluation, we define some sets of terms, constructed during the evaluation of test case generators. We indicate by $S \subseteq \Upsilon$ the set of shapes, generated by the predicate `g` and by C_S the set of the constrained shapes, produced after the execution of the filters. We denote by F the set of the feasible shapes, that is, the set of the symbolic shapes selected using a complete solver. Finally, we denote by $D \subseteq \overline{\Upsilon}$ the set of all possible (domain consistent) instantiations of the shapes in S. The set D, ideally, represents the search space we would have to consider if we were not using a symbolic approach. In order to compute those sets, one can instrument appropriately the test generator. We have $F \subseteq T \subseteq D$ but no knowledge about the relation of T with the other sets.

Note that, from Definitions 2 and 3 it directly follows that the sets of the shapes, of the constrained shapes, and of the feasible shapes are finite. However, if no further assumption is made on the predicate `init` and on the domains of the variables in the lists $\mathtt{V}_1$,...,$\mathtt{V}_k$, we have no guarantee on the finiteness of the set of test cases T.

2.3 Red-Black Trees

In this section we discuss a concrete example, in order to show our approach in practice. We consider a classical data structure, called Red-Black tree.

Example 1. A Red-Black tree [10] is a binary search tree where each node has two labels: a *color*, 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 equations:

```
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:

(I$_1$) every path from the root to a leaf has the same number of black nodes,
(I$_2$) no red node has a red child, and
(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.

The CLP specification of a Red-Black tree generator is parameterized by the minimum and maximum tree size (defined as the number of its nodes), and by the maximum value for the keys. The following clause defines the test case generator:

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

Given the ground (non-negative) input integers minSize, maxSize, and numKeys, the set Ans_P(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 predicate varlist(N,L), is used for constructing a list L of N fresh $\mathcal{V}_C$-variables.

The first line of the Preamble fixes the domain of the variable S denoting the tree size. 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 ,Cs     ,Cs ) :- S#=0.
2. lbt(t(C,K,L,R),S,[K1|Ks],NKs,[C1|Cs],NCs) :-
             S#>=1, SL#>=0, SR#>=0, S#=SL+SR+1, C#=C1, K#=K1,
             lbt(L,SL,Ks,TKs,Cs,TCs), lbt(R,SR,TKs,NKs,TCs,NCs).
```

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 it is split into a pair of non-negative integers SL and SR denoting the size of the left and right subtrees, respectively, such that S = SL + SR + 1. The left and right subtrees are then constructed recursively. The remaining two arguments contain the key and color variables, that are placed in each node.

The predicate pi (for *path invariant*) encodes the invariant $(\mathbf{I}_1)$:

```
3. pi(e         ,C) :- C#=0.
4. 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 d is the number of black nodes on every root-to-leaf path in t. We say that d is the value of the *black-nodes counter* of t. 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 $(\mathbf{I}_2)$:

```
5. ci(e         ).
6. ci(t(C,_,L,R)) :- root_col(L,C), root_col(R,C), ci(L), ci(R).

7. root_col(e         ,_).
8. root_col(t(C,_,_,_),D) :- C+D#>0.
```

In clause 6 the color invariant is enforced by testing the color of the roots of the left and right subtrees. For each branch, the constraint C+D#>0 enforces either the father or the child to be black (i.e., equal to 1).

Finally, the predicate ordered defines the invariant $(\mathbf{I}_3)$:

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

We compared our declarative CLP-based test generator for Red-Black trees with a generator written using Korat [30], a tool for bounded-exhaustive testing of Java programs, which is specifically tailored for the construction of structurally complex test inputs. Korat allows the generation of complex data structures by providing primitives to populate an object domain, to initialize objects, and to set links among them. Korat performs a systematic search of the program input space, avoiding the full exploration of failing regions and the generation of isomorphic structures.

Following [27], we consider the 'canonical set' $\mathit{Ans}_P(\mathtt{rbtree}(\mathtt{T}, s, s, s))$, for $s \in \mathbb{Z}$, which is the set of all Red-Black trees of `s` nodes and keys ranging in $\{0, \ldots, s-1\}$. The results of the comparison are summarized in Table 1 of Section 4 and show that the CLP-based Red-Black tree generator is much more efficient than the Korat one.

3 Transformational Optimization

In this section we present a technique based on *program transformation* [7,36] for improving the efficiency of test case generator programs.

The left-to-right scheduling of constraints and atoms implemented by most CLP systems can be a source of inefficiency, because it enforces the full generation of a data structure shape before the evaluation of a filter can start. This difficulty can be mitigated by providing a more intelligent scheduling strategy, so that filters are applied as soon as a data structure has been partially generated, hence determining early failure and backtracking when a data structure shape does not satisfy the constraints defined by the filters. A lot of research in the area of logic programming has been devoted to devise techniques for allowing the user to specify sophisticated selection strategies, such as *delay (or wait) declarations*, which determine a dynamic scheduling of the atoms to be evaluated based on their instantiation patterns [31]. However, the increased intricacies of the implementation of the dynamic scheduling mechanism may reduce the efficiency gains due to more intelligent scheduling.

An alternative approach that we follow here is based on a program transformation that modifies a given test case generator in such a way that the transformed program computes the same test cases and behaves under left-to-right scheduling like the original program behaves under a selection rule which interleaves generators and filters. Hence, program transformation avoids the overhead due to the execution of sophisticated dynamic scheduling mechanisms.

We will follow the rule-based approach to transformation [7], where a program transformation is achieved by applying semantics preserving *transformation rules* (see [13,14,36] for the case of logic and constraint logic programs) guided by suitable strategies which have the goal of improving program efficiency (see [33,34] for strategies related to the one presented here).

The transformation rules we will use here are the following, taken from [13,14,36]: *definition*, *unfolding*, *folding*, *goal replacement*, *constraint replacement*, and *clause deletion*. These rules are applied according to the *Filter Promotion* (FP) strategy outlined in Figure 1, where: (1) the *Dfn* function applies the definition rule, (2) the *Unf* function applies the unfolding rule, (3) the *Rrg* function applies the goal replacement and clause deletion rules, and finally, (4) the *Fld* function applies the folding rule. Below we provide more details about the FP strategy by means of the Red-Black tree example.

Fig. 1: The Filter Promotion Strategy

```
Input: Clause $\gamma$ in the test case generator $P$;
Output: A set $T_\gamma$ of clauses and a test case generator $\mathit{TransfP} = (P - \{\gamma\}) \cup T_\gamma$ such that,
for all queries $Q$ of the form :- tc(T,p1,...,ph)., where T is a variable and p1,...,ph $\in \mathbb{Z}$,
$\mathit{Ans}_P(Q) = \mathit{Ans}_{\mathit{TransfP}}(Q)$.
$T_\gamma := \{\gamma\}$;   $\mathit{Defs} := \emptyset$;
while  $T_\gamma$ contains a clause with multiple occurrences of a relevant variable in the body  do
   Define:      $\mathit{Transf} := \mathit{Dfn}(T_\gamma)$;   $\mathit{Defs} := \mathit{Dfn}(T_\gamma) \cup \mathit{Defs}$;
   Unfold:      $\mathit{Transf} := \mathit{Unf}(\mathit{Transf}, P)$;
   Rearrange:   $\mathit{Transf} := \mathit{Rrg}(\mathit{Transf})$;
   Fold:        $T_\gamma := \mathit{Fld}(T_\gamma \cup \mathit{Transf}, \mathit{Defs})$;
end-while
```

The FP strategy takes as input a test case generation program P and a clause $\gamma \in P$ which is an instance of the template (τ) defined in Section 2.

In order to apply the FP strategy and guarantee its termination by applying the results of [33], we now introduce the notion of *marking* on the input program P. Intuitively, the marking identifies the slice of P that manipulates the data structures to be generated. We assume that some of the predicates occurring in P are marked as *relevant*. An atom with relevant predicate is said to be a *relevant atom*. Each relevant predicate has exactly one argument position (without loss of generality, the first one) which is marked as *relevant*. The argument occurring in a relevant position is said to be a *relevant argument*. Each variable of type $\mathbb{T}$ occurring in a relevant argument is said to be a *relevant variable*.

Definition 6 (Linear Marked Program). *A test case generator P is said to be a* linear marked program *if its marking satisfies the following conditions. (i) The first argument position of* `g`*,* `inv`$_1, \ldots,$ `inv`$_n$ *is relevant. (ii) Only predicates in $\Pi_{\mathcal{U}}$ and terms of type $\mathbb{T}$ are marked as relevant. (iii) For each clause C in $P - \{\gamma\}$ whose head is relevant we have that: (iii.1) every variable* occurs at most once *in the relevant argument t_0 of the head of C, (iii.2) every relevant argument occurring in the body of C is a* proper subterm *of t_0, and (iii.3) distinct relevant arguments occurring in the body of C share no variables. (iv) Every non-relevant argument has mode in* $\{$`+`$,$`?`$\}$.

Thus, in a linear marked program, both the generator and the filters are defined by structural induction on the data structure we want to generate.

Note that clause γ is not required to fulfill Condition (iii.3) of the above definition, and indeed the atoms `g(T,P1,...,Ph,V1,...,Vk)`, `inv`$_1$`(T,P1,...,Ph,V1,...,Vk)`, ..., `inv`$_n$`(T,P1,...,Ph,V1,...,Vk)`, share the relevant variable `T`.

The output of the Filter Promotion strategy is a set T_γ of clauses and a transformed program *TransfP* which is equivalent to P with respect to queries of the form

:- tc(T,p1,...,ph). From the operational point of view, *TransfP* achieves filter promotion by performing consistency tests defined by the filters as soon as a partial instantiation of T is generated.

3.1 Filter Promotion Applied to the Red-Black Tree Generator

We illustrate the transformation strategy on the Red-Black tree program (RBT) discussed in Section 2. Let γ be the clause defining the predicate rbtree. Program RBT, where the predicates lbt, pi, ci, and ordered (along with their first arguments) are marked as relevant, is a linear marked program.

Define. The Filter Promotion strategy starts off by applying the *Dfn* function, which introduces a new predicate sync (which stands for *synchronized*) defined by the following clause:

```
  sync(T,P1,...,Ph,V1,...,Vk) :-
        g(T,P1,...,Ph,V1,...,Vk),                                   (†)
        inv_1(T,P1,...,Ph,V1,...,Vk), ..., inv_n(T,P1,...,Ph,V1,...,Vk).
```

whose body consists of precisely those atoms in the body of γ that share the relevant variable T. The predicate sync has mode $(-,?^h,+^k)$. In the Red-Black tree example, the *Dfn* function introduces the following new predicate definition:

```
1.  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).
```

Transf and *Defs* are both initialized to the set consisting of clause 1. Then the strategy proceeds by looking for a recursive definition of sync.

Unfold. The unfolding rule consists of a symbolic evaluation step: an atom A in the body of a clause is selected according to a given *selection function*, and A is replaced by the bodies of the clauses in program P whose heads unify with A. The function *Unf* applies the unfolding rule once or more times. For the first step of unfolding, *Unf* selects the leftmost relevant atom in the body of each clause in *Transf*. In the Red-Black tree example *Unf* selects the atom lbt(T,S,Keys,NewKeys) in the body of clause 1, thus deriving the following pair of clauses (new variable names have been automatically generated by our prototype transformation system MAP [1]):

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

The first unfolding step causes a partial instantiation of T. Further unfolding steps are performed on the filter predicates to add constraints to the partially generated structure, guided by a selection function defined as follows. A relevant atom $p(t,\ldots)$ in the body of a clause is *selectable* if for every head in $P-\{\gamma\}$ unifiable with $p(t,\ldots)$ via a most general unifier ϑ, the term $t\vartheta$ is a variant of t. Then, every clause is unfolded with respect to a selectable atom, until no such atom exists. In the Red-Black tree example, from clauses 2 and 3 we derive:

```
4. sync(e,A,B,B,C,C,D,E,F) :- A#=0, D#=0.
5. sync(t(A,B,C,D),E,[F|G],H,[I|J],K,L,M,N) :-
        E#>=1, O#>=0, P#>=0, E#=O+P+1, A#=I, B#=F,
        lbt(C,O,G,Q,J,R), lbt(D,P,Q,H,R,K), S#>=0, L#=S+A, pi(C,S), pi(D,S),
        not_redroot(C,A), not_redroot(D,A), ci(C), ci(D), M#=<B, B#<N, T#=B+1,
        ordered(C,M,B), ordered(D,T,N).
```

Finally, all clauses are unfolded by selecting the atoms that unify with the heads of constrained facts only. In the Red-Black tree example, by selecting the `not_redroot` atoms, from clause 5 we derive:

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

At this point of the execution of the strategy *Transf* consists of clauses 7–10.

Rearrange. By unfolding we have partially constructed a structure together with some constraints it must satisfy. This partial generation does not determine a significant efficiency improvement on its own, but it provides opportunities for optimizations performed by the *Rrg* function, as listed below.

- Promotion of consistency checks on constraints by moving them to the left of atoms.

- Simplification of constraints by projecting out existential variables whenever possible.

- Deletion of clauses whose body contains an unsatisfiable constraint.

- Deletion of subsumed clauses (H `:-` $c, B.$ is subsumed by H `:-` $d.$ if c entails d).

- Simplification of clauses by exploiting properties of user-defined predicates.

Finally, the *Rrg* function also reorders the atoms so as to derive clauses of the form

H `:-` $c, B_1, \ldots, B_m$.

where, for $i = 1, \ldots, m$, B_i is a conjunction of consecutive atoms, called *block*, such that two distinct atoms occur in B_i iff they share a relevant variable (in particular, each non-relevant atom occurs in a block made out of that atom only). In our example we derive the following clauses:

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

where clauses 12 and 13 contain one block each, while clause 14 contains two non-singleton blocks:

```
    lbt(t(C,D,E,F),S,L,U), pi(t(C,D,E,F),V),
      ci(t(C,D,E,F),O,W), ordered(t(C,D,E,F),Q,B)
```

and

```
    lbt(t(G,H,I,J),T,U,M), pi(t(G,H,I,J),V),
      ci(t(G,H,I,J),W,P), ordered(t(G,H,I,J),X,R)
```

At the end of the *Rearrange* step, *Transf* consists of clauses 4, 11, 12, 13, and 14.

Fold. The *Fld* function applies the folding rule once or more times to the clauses in $T_\gamma \cup$ *Transf*, with the goal of deriving a recursive definition of `sync`. Indeed, *Fld* replaces each block B_i which is an instance of the body of clause (†) by the corresponding instance of the head (`sync(T,S,Keys,NewKeys,Colors,NewColors,Min,Max,D)` in our example). Blocks without relevant atoms are not folded. In the Red-Black tree example, by folding the clause defining the predicate `rbtree` and also clauses 12, 13, and 14, we get:

```
15. rbtree(A,B,C,D) :- B#=<E, E#=<C, labeling([],[E]), varlist(E,F),
         varlist(E,G), H#=D-1, domain(F,0,H), domain(G,0,1),
         sync(A,E,F,[],G,[],I,0,D), labeling([],F), labeling([],G).
4.  sync(e,A,B,B,C,C,D,E,F) :- A#=0, D#=0.
11. sync(t(A,B,e,e),C,[D|E],E,[F|G],G,H,I,J) :-
        A#=F, B#=D, C#=1, H#=A, I#=<B, B#<J.
16. sync(t(A,B,e,t(C,D,E,F)),G,[H|I],J,[K|L],M,N,O,P) :-
        G#>=1, Q#>=0, G#=Q+1, A#=K, B#=H, R#=0, N#=A, S#=0, C+A#>0,
        O#=<B, B#<P, T#=B+1, sync(t(C,D,E,F),Q,I,J,L,M,S,T,P).
```

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

This program implements filter promotion in the sense that at each recursive call the predicate `sync` generates a portion of the tree structure and immediately tests whether the constraints on the finite domain variables occurring in this partial structure are consistent. In other words, `sync` interleaves structure generation and invariant checking, possibly determining an early failure in the case where the partially generated structure is not consistent with the invariants defining Red-Black trees.

3.2 Termination and Correctness of the FP Strategy

Termination of the FP Strategy. Due to Condition (iii.2) of Def. 6 the function *Unf* terminates. Other selection strategies guarantee the termination of *Unf*. For a general treatment of the problem of controlling unfolding in the related field of *Partial Deduction* we refer to the survey [25]. Since also the other functions used in the FP strategy obviously terminate, in order to establish the termination of the strategy one should prove that a finite number of iterations is performed.

In general, it may happen that some block with multiple relevant atoms cannot be folded because it is not an instance of the body of (†). In this case, after the *Fold* step, in T_γ there exists a clause whose body contains multiple occurrences of a relevant variable, and hence the transformation strategy performs other iterations of the *while* loop. At a generic iteration, given a set T_γ of clauses, the *Dfn* function computes a set of new predicate definitions as follows. Let B be a non-singleton block in the body of a clause in T_γ. The *Dfn* function computes a new clause of the form

$newp(X_1, \ldots, X_m)$:- B'.

where *newp* is a new predicate symbol, $\{X_1, \ldots, X_m\} = \mathit{Vars}(B')$, and there exists a *most general* substitution ϑ with the following properties: (i) $B = B'\vartheta$, and (ii) for all relevant variables X_i, X_j in B', if $X_i\vartheta = X_j\vartheta$ then $X_i = X_j$.

Then, for each new clause introduced by *Dfn* the FP strategy applies *Unfold*, *Rearrange*, and *Fold*. To guarantee the termination of the strategy we need to enforce that finitely many new definitions are introduced. Since the class of linear marked programs taken as input by the strategy is a subclass of the one considered in Section 4 of [33], the finiteness of the set of new definitions, and hence the termination of the strategy is a consequence of Theorem 11 of that paper.

All programs considered in Section 4 are linear marked, and thus the FP strategy could be applied in a fully automatic way. If the input of the strategy is not a linear marked program, then termination is not guaranteed. However, by using the *generalization* technique presented in [33], we can define a transformation strategy that always terminates, possibly paying the price of deriving a suboptimal program.

Correctness of the FP Strategy. It is fairly simple to verify that each rule application preserves well-modedness. Indeed, this is straightforward for definition (provided that we suitably define the mode for the new predicate), unfolding, reordering of constraints, clause deletion, and folding. For the reordering of atoms during the *Rearrange* step, the preservation of well-modedness is ensured by condition (iv) of Definition 6.

By construction there is no clause in T_γ whose body contains a multiple occurrence of a relevant variable. In particular, these multiple occurrences can be eliminated by folding, which replaces a block with multiple occurrences of a relevant variable by an atom with a single occurrence.

The FP strategy applies the transformation rules by fulfilling the restrictions considered in [13,14,36], thus ensuring that the least $\mathcal{D}$-model of $P \cup \mathit{Defs}$, where *Defs* is the set of new definitions introduced by the strategy, is equal to the least $\mathcal{D}$-model of *TransfP*. Now, let us consider a query Q of the form `:- tc(T,p1,...,ph).`, where `T` is a variable and `p1,...,ph` are ground integers. Both P and *TransfP* are well-moded, and hence all answers to Q in P or *TransfP* are of the form (σ, true), where $\mathtt{T}\sigma$ is a ground term (see Proposition 2). By the soundness and completeness of LD-resolution, we get that $\mathit{Ans}_P(Q) = \mathit{Ans}_{\mathit{TransfP}}(Q)$, and also that for any well-moded query Q' of the form `:-sync(T,p1,...,ph,v1,...,vk).`, if $\mathtt{p1},\ldots,\mathtt{ph} \in \mathbb{Z}$ then $\mathit{Ans}_{\mathit{TransfP}}(Q')$ is finite.

Thus, we have the following result.

Theorem 1 (Termination and Correctness of Filter Promotion). *For any clause γ and linear marked program P the FP strategy terminates. Let TransfP be the output of the strategy. Then:*

(i) TransfP is a test case generator (in particular, well-moded),

(ii) no clause in TransfP has a multiple occurrence of a relevant variable in its body,

(iii) for all queries Q of the form `:- tc(T,p1,...,ph).`*, where* `T` *is a variable and* $\mathtt{p1},\ldots,\mathtt{ph} \in \mathbb{Z}$*, $\mathit{Ans}_P(Q) = \mathit{Ans}_{\mathit{TransfP}}(Q)$.*

4 Experimental Evaluation

In this section we discuss the results obtained by performing an experimental evaluation of our CLP-based method for test generation, considering: (1) the original declarative programs and (2) their optimized versions obtained by Filter Promotion.

During the first set of experiments, we assessed the performance of the declarative CLP-based test generator for Red-Black trees presented in Section 2. Then, we compared the results with those obtained by running Korat, a state-of-the-art BET tool for Java on the same problem instances. The experimental data obtained by running the declarative CLP-based test generator and Korat are reported in Table 1.

In our experiments, the performance of CLP-based test generators is always one or two orders of magnitude better than the performance of the corresponding Korat generators. This difference could partly be ascribed to the fact that Korat builds tree data structures generating elements from a domain of graphs and filtering out the

Table 1: 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, needed for generating all the structures running the original CLP generator of Sec. 2 using the GNU and SICStus Prolog systems (3-4), the size of the domain and the number of shapes, constrained shapes and feasible shapes (5-8). The last two columns (9-10) report the time needed by the Korat generator and the number of explored structures. (-) means not computed within one hour.

		CLP-based						Korat	
Size	Trees	Time		Counts				Time	Counts
		GNU	SICStus	$\lvert D\rvert$	$\lvert S\rvert$	$\lvert C_S\rvert$	$\lvert F\rvert$		Explored
9	122	0.06	0.24	10^{15}	4862	54	46	3.59	1510006
10	260	0.20	0.89	10^{17}	16796	92	76	15.79	7530712
11	586	0.71	3.17	10^{19}	58786	218	190	89.24	39089158
12	1296	2.76	12.09	10^{22}	208012	512	456	469.66	205512574
13	2708	10.12	45.13	10^{24}	742900	1004	904	3080.21	1084433242
14	5400	38.21	172.82	10^{27}	2674440	1960	1696	-	-

acyclic ones. In CLP, on the contrary, trees can be represented in a straightforward manner by using terms, which are natively managed by existing Prolog systems.

Columns 7 and 8 of Table 1 show that the number of feasible shapes is never much smaller than the number of constrained shapes. This means that, for the considered examples, most of the constrained shapes do actually lead to a test case, and thus the consistency checks performed by the Prolog systems during the search for a solution are effective.

Moreover, we notice that the number of constrained shapes is considerably smaller, and grows more slowly, than the number of shapes (actually they grow exponentially with a base of 2 and 4, respectively). In cases like this, where the ratio between constrained shapes and shapes is small, the Filter Promotion strategy, is potentially able to improve the performance in a substantial way. We will elaborate more on this aspect in the rest of this section.

In the second set of the examples, we applied our Filter Promotion strategy to some CLP-based test generators for tree-like complex data structures and we compared the declarative and the optimized ones on problem instances of different size, focusing on their time performance and the number of shapes generated. The results are reported in Table 2.

In order to assess the effectiveness of our Filter Promotion strategy we have comparatively analyzed the declarative CLP-based generators and the corresponding optimized ones for the following tree-like data structures: AVL-trees (balanced binary search trees, where the length of the root-to-leaf paths differ by at most one), B-trees (n-ary search trees used in filesystems), Red-Black trees (presented in Section 2) and Well-Labeled trees (labeled trees where adjacent nodes have labels differing by at most one). Similarly to Red-Black trees, we considered general, parametric test generators, but we focused the experimental evaluation on canonical structures, where the domain of keys is $\{0, \ldots, n-1\}$ and n is the number of nodes of the tree under consideration. For B-trees, in particular, we focused on canonical B-trees of order 4, also known as 2-3-4 trees.

Table 2: Comparison of declarative and optimized versions of generators for tree-like data structures. The table reports the name of the examples (column 1), the size and the number of computed trees (2-3), the time, in seconds, needed for generating all the trees running the declarative and optimized CLP generators using SICStus Prolog (4-5), the number of shapes and constrained shapes for the declarative version (7-8), and the number of shapes for the optimized version (9). Columns (6) and (10) report the time (and number of shapes, respectively) percentage ratio between the optimized version and the declarative one. Zero in time columns means less than 10 ms, zero in ratio columns means less than 0.01%, (-) means not computed within 400 seconds or undefined.

Example			Time			Shapes			
Name	Size	Trees	T	T_{opt}	$ratio_T(\%)$	$\|S\|$	$\|C_S\|$	$\|S_{opt}\|$	$ratio_S(\%)$
AVL-tree	11	70	4.09	0.09	2.20	58786	70	70	0.12
	12	184	14.55	0.18	1.24	208012	184	184	0.09
	13	476	53.84	0.33	0.61	742900	476	476	0.06
	14	872	194.78	0.64	0.33	2674440	872	872	0.03
	23	174374	-	247.89	-	-	-	174374	-
B-tree	13	8	6.58	0.01	0.15	208012	8	8	0
	14	10	24.46	0.01	0.04	742900	10	10	0
	15	14	86.18	0.03	0.03	2674440	14	14	0
	16	21	324.06	0.04	0.01	9694845	21	21	0
	31	2952	-	398.27	-	-	-	2952	-
RB-tree	11	586	3.17	0.51	16.09	58786	218	218	0.37
	12	1296	12.09	1.13	9.35	208012	512	512	0.25
	13	2708	45.13	2.43	5.38	742900	1004	1004	0.14
	14	5400	172.82	5.36	3.10	2674440	1960	1960	0.07
	19	140612	-	260.16	-	-	-	72606	-
WL-tree	6	41112	0.18	0.19	105.56	42	42	42	100
	7	463548	2.11	2.18	103.32	132	132	132	100
	8	5280270	24.02	23.88	99.42	429	429	429	100
	9	60570250	267.72	270.54	101.05	1430	1430	1430	100
	10	-	-	-	-	-	-	-	-

For all the examples, the number of shapes of the optimized version ($|S_{opt}|$) equals the number of constrained shapes ($|C_S|$). In most cases (with the exception of WL-tree, which we discuss later) this number is at least two orders of magnitude smaller than the number of shapes ($|S|$). When the constraint solver is able to detect that several potential shapes are unfeasible, the optimized generators obtained by applying the Filter Promotion strategy, perform much better than the declarative ones. The reason for such performance improvement is due to the fact that the optimized generators are able to *directly generate* the set of constrained shapes, avoiding the intermediate generation of a large number of shapes which do not lead to actual solution.

The time performance improvement of the optimized version w.r.t. the declarative one ($ratio_T$), is linearly dependent on the improvement over the number of shapes generated ($ratio_S$). This is confirmed by the experiments performed on the WL-tree instances, where the number of constrained shapes equals the number of shapes, and the optimized version of the generator is not better than the declarative one.

The CLP-based approach to bounded exhaustive testing has been shown to be very efficient when using a straightforward, declarative encoding, as already demonstrated by the experiments reported in [35], where the authors considered generators for sorted lists of integers, integer-labeled search trees, and array-based representations of heaps and disjoint set partitions.

The experimental evaluation we have performed in this paper shows that, starting from a natural, declarative encoding, the application of the Filter Promotion strategy is often able to introduce an additional level of improvement. This is particularly important as it allows the test engineer to encode the properties of the considered data structure in a very declarative and modular way, focusing on each property separately from the others, thus increasing his/her confidence in the correctness of the test specification. Then, the Filter Promotion strategy can be applied on the declarative test generator in order to obtain an equivalent, optimized test generator.

4.1 Experimental settings

We selected two different CLP(FD) systems for running our experiments: SICStus, for its widespread availability and industrial strength, and GNUProlog, for its efficient compilation. In our experiments, GNUProlog outperforms SICStus, due to its efficient compilation of FD constraints. 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.

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.

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 [28].

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, SICStus Prolog 3.12.8 and OpenJDK 7. The timings were collected using CLP and Java statistics predicates.

5 Related Work and Conclusions

Constraint-based techniques have been widely used in the field of test case generation, since pioneering work in [12]. Early use of CLP for test generation can be found in the tool ATGen [29], developed for testing Spark ADA programs.

More recent approaches, such as [11], enable white-box (i.e., code-driven) testing of an imperative language with pointers and heap by symbolic execution of a small-step operational semantics in CLP. Further work on testing has been done in [8,22] for the generation of heap-allocated data structures, following a fixed coverage criteria for the choice of the test cases. The work in [21] presents a technique for white-box testing of object-oriented languages, where test case generators are obtained by partial evaluation of a language interpreter with respect to a given program.

In this paper we are motivated by black-box testing and we follow the BET approach. We focus on showing that CLP search strategies, besides being very general, can be efficient and can be further optimized by employing suitable optimization techniques. In the BET setting, our optimization shows several advantages. The approach proposed in this paper shares motivation also with research on model-based test generation and combinatorial testing [23,32,37], where constraints are adopted for increasing declarativeness and efficiency.

The declarative approach has also been adopted by test generation tools such as Korat [30], which has been used in our experimental evaluation, UDITA [20], and TestEra [24]. 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 [20] can be seen as a particular Constraint Programming strategy. Moreover, these tools are language-specific and they are not easily adaptable to other languages.

In [9] 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.

Concerning our optimization technique, the idea of improving *generate-and-filter* programs by transformation is not new. For instance, this idea stands behind the *Filter Promotion* transformation strategy proposed both for functional programs [5] and logic programs [34]. *Compiling control* [6] is another related transformation technique which, given a logic program P and a selection rule R, derives a new program that behaves under LD-resolution like P behaves under R. However, the transformation technique we present here is the first one specifically designed for CLP test case generation programs, where we can exploit the peculiar form of the programs and the properties of the constraint-based computation model.

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

The approach of this paper is essentially based on the idea of promoting constraint solving with respect to structural induction. One should be aware, however, that in realistic examples the size of constraints grows very quickly, together with the search space the solver is required to traverse. Our approach currently relies on ingenuity built in the constraint solver but further optimizations are possible. For example, in [15,18], a static analysis technique is used to infer constraints over variables of a CLP-based encoding of an infinite state reactive system. These constraints are very useful to prune the search of model checking algorithms applied to those infinite state systems. Similar techniques can be applied to infer extra constraints on data structure variables, to improve the solver pruning mechanisms and mitigate the explosion of the search space size.

A further optimization to be explored concerns the use of mixed constraint solvers. Indeed, more efficient solvers on different domains can be employed to compute conservative approximations of solutions, to be refined using domain specific solvers.

For example, if we consider data structures containing integer values we may replace the integer (or finite domain) solver by a more efficient solver on real numbers when determining *feasible* structures. This idea has been successfully applied for the specialization of CLP programs [19].

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 developing 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*.

We also plan to study extensions of the correctness results stated in Theorem 1, and look for sufficient conditions guaranteeing that our optimization technique based on program transformation preserves properties of the operational semantics. In particular, we want to guarantee that no left-terminating program (where all derivations starting from a ground query are finite) is transformed into a non-terminating one. Since we are interested in computing all the solutions of test generator programs, we want to avoid to introduce infinite derivations.

Program transformation of CLP programs could also be applied to white-box testing by lifting the techniques which have been developed for the verification of imperative programs on integers [16]. In that work, similarly to what has been done in [21], an interpreter encoding the semantics of the programming language is specialized w.r.t. a CLP representation of the considered program. However, the transformational method could take advantage of more powerful rules, like those for reasoning about programs manipulating arrays and other data structures [17].

In conclusion, we believe that, due to its inherent symbolic execution mechanism, CLP has a promising application field in test case generation, especially in the case of complex input data. Indeed, CLP provides a highly declarative language and ensures efficiency by using dedicated constraint solvers and optimization techniques.

Acknowledgements. We thank the anonymous referees for their useful feedback.

References

1. `http://www.iasi.cnr.it/~proietti/system.html`.
2. `http://www.gprolog.org/`.
3. `http://www.sics.se/sicstus/`.
4. K. R. Apt and E. Marchiori. Reasoning about prolog programs: From modes through types to assertions. *Formal Asp. Comput.*, 6(6A):743–765, 1994.
5. R. S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Toplas*, 6(4):487–504, 1984.
6. M. Bruynooghe, D. De Schreye, and B. Krekels. Compiling control. *Journal of Logic Programming*, 6:135–162, 1989.
7. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
8. 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.
9. 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.
10. T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms (3rd ed.).* MIT Press, 2009.

11. F. Degrave, T. Schrijvers, and W. Vanhoof. Towards a framework for constraint-based test case generation. In *Proc. LOPSTR 2008*, LNCS 6037, pp. 128–142, Springer, 2009.
12. R.A. DeMillo and A.J. Offutt. Constraint-based automatic test data generation. *IEEE Trans. Software Eng.*, 17(9):900–910, 1991.
13. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
14. 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*, LNCS 3049, pages 291–339. Springer, 2004.
15. F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization strategies for the verification of infinite state systems. *Theory and Practice of Logic Programming*, 13(2):175–199, Cambridge, 2013.
16. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying Programs via Iterated Specialization. In *PEPM '13*, pages 43–52. ACM, 2013
17. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verification of Imperative Programs by Constraint Logic Program Transformation. In *SAIRP '13*, pages 186–210. EPTCS 129, 2013
18. F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Improving reachability analysis of infinite state systems by specialization. In G. Delzanno and I. Potapov, editors, *Proc. of Reachability Problems (RP)*, LNCS 6945, pages 165–179, Springer, 2011.
19. F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Using real relaxations during program specialization. In G. Vidal, editor, *Proc. of Logic-based Program Synthesis and Transformation (LOPSTR)*, LNCS 7225, pages 106–122, Springer, 2012.
20. M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. *Proc. ICSE*, pp. 225–234. ACM, 2010.
21. 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.
22. A. Gotlieb, B. Botella, and M. Rueher. A CLP framework for computing structural test data. In *Computational Logic*, LNAI 1861, pages 399–413. Springer, 2000.
23. R. M. Hierons and K. Bogdanov and J. P. Bowen and R. Cleaveland and J. Derrick and J. Dick and M. Gheorghe and M. Harman and K. Kapoor and P. Krause and G. Lüttgen and A. J. Simons and S. A. Vilkomir and M. R. Woodward and Hussein Zedan. Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2):1–76, 2011.
24. S. Khalek, G. Yang, L. Zhang, D. Marinov, and S. Khurshid. TestEra: A tool for testing Java programs using Alloy specifications. In *Proc. ASE*, pp. 608–611. IEEE, 2011.
25. M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4&5):461–515, 2002.
26. J.W. Lloyd. *Foundations of Logic Programming, 2nd Edition.* Springer, 1987.
27. D. Marinov. *Automatic Testing of Software with Structurally Complex Inputs.* PhD thesis, MIT, 2005.
28. K. Marriott and P.J. Stuckey. *Programming with constraints: An introduction.* MIT Press, Cambridge, Mass., 1998.
29. C. Meudec. ATGen: Automatic test data generation using constraint logic programming and symbolic execution. *Software Testing, Verification and Reliability*, 11(2):81–96, 2001.
30. A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid. Korat: A tool for generating structurally complex test inputs. In *Proc. ICSE*, pages 771–774. IEEE Press, 2007.
31. L. Naish. *Negation and Control in Prolog.* LNCS 238. Springer-Verlag, 1986.
32. C. Nie and H. Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2):1–29, 2011.

33. 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.
34. H. Seki and K. Furukawa. Notes on transformation techniques for generate and test logic programs. In *Proc. ILPS, San Francisco, USA*, pages 215–223. IEEE Press, 1987.
35. V. Senni and F. Fioravanti. Generation of test data structures using constraint logic programming. In *Proc. TAP*, LNCS 7305, pages 115–131. Springer, 2012.
36. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In *Proc. ICLP'84*, pages 127–138, Uppsala, Sweden, 1984.
37. M. Utting and A. Pretschner and B. Legeard. A taxonomy of model-based testing approaches. *Softw. Test., Verif. Reliab.*, 22(5):297–312, 2012.