Property-Based Test Case Generators for Free*

Emanuele De Angelis[1], Fabio Fioravanti[1], Adrián Palacios[2], Alberto Pettorossi[3], and Maurizio Proietti[4]

[1] DEC, University "G. d'Annunzio" of Chieti-Pescara, Italy
{emanuele.deangelis,fabio.fioravanti}@unich.it
[2] MiST, DSIC, Polytechnic University of Valencia, Spain, apalacios@dsic.upv.es
[3] University of Roma Tor Vergata, Italy, pettorossi@info.uniroma2.it
[4] CNR-IASI, Roma, Italy, maurizio.proietti@iasi.cnr.it

Abstract. Property-Based Testing requires the programmer to write suitable *generators*, i.e., programs that generate (possibly in a random way) input values for which the program under test should be run. However, the process of writing generators is quite a costly, error-prone activity. In the context of Property-Based Testing of Erlang programs, we propose an approach to relieve the programmer from the task of writing generators. Our approach allows the automatic, efficient generation of input test values that satisfy a given specification. In particular, we have considered the case when the input values are data structures satisfying complex constraints. That generation is performed via the symbolic execution of the specification using constraint logic programming.

1 Introduction

Over the years, Property-Based Testing (PBT), as proposed by Claessen and Hughes [8], has been established as one of the favorite methods for testing software. The idea behind PBT is as follows: instead of supplying specific inputs, i.e., test cases, to a program under test, the developer defines properties to be satisfied by the program inputs and outputs. Then, random inputs are generated and the program is run with those input values, thereby producing output values and checking whether or not the input-output pairs satisfy the desired property. If the output associated with a particular input does not satisfy the property, the counterexample to the property reveals an undesirable behavior. Then, the developer can modify the program under test so that the counterexample is no longer generated. The fact that input values are generated in a random way plays a key role in the PBT techniques, because randomicity enables the generation of valid inputs which originally could have escaped the attention of the developer.

QuickCheck [8] is the first tool that implemented Property-Based Testing and it works for the functional language Haskell. Then, a similar approach has been followed for various programming languages, and among many others, let us mention: (i) Erlang [2,27], (ii) Java [22,39], (iii) Scala [33], and (iv) Prolog [1].

* This work has been partially supported by the EU (FEDER) and the *Spanish Ministerio de Ciencia, Innovación y Universidades*/AEI, grant TIN2016-76843-C4-1-R and by the *Generalitat Valenciana*, grant PROMETEO-II/2015/013 (SmartLogic).

In this paper we will focus on the dynamically typed functional programming language Erlang and, in particular, we will refer to the PropEr framework [27,30]. Typically, in PropEr the set of valid input data is defined through: (i) a type specification, and (ii) a filter specification (that is, a constraint), which should be satisfied by the valid inputs. When working with user-defined types and filters, the developer must provide a *generator*, that is, a program that constructs input data of the given type satisfying the given filter. PropEr supports writing generators by providing a mechanism for turning type specifications into data generators, and also providing primitives for constraining data size and assigning frequencies to guide data generation. However, the task of writing a generator that satisfies all constraints defined by a filter is left to the developer. Unfortunately, writing and maintaining generators is a time-consuming and error-prone activity. In particular, hand-written generators may result in the generation of sets of non-valid inputs or, even worse, sets of inputs which are too restricted.

In this paper we explore an approach that relieves the developer from the task of writing data generators of valid inputs. We assume that the data generation task is specified by providing: (i) a *data type specification*, using the Erlang language for that purpose, and (ii) a *filter specification* provided by any boolean-valued Erlang function. We have constructed a symbolic interpreter, written in the *Constraint Logic Programming* (CLP) language [20], which takes the data type and the filter specification, and automatically generates data of the given type satisfying the given filter. Our interpreter is *symbolic*, in the sense that it is able to run Erlang programs (in particular, the filter functions) on symbolic values, represented as CLP terms with possibly variable occurrences.

The symbolic interpreter works by exploiting various computational mechanisms which are specific to CLP, such as: (i) *unification*, instead of matching, which enables the use of predicate definitions for generating terms satisfying those predicates, (ii) *constraint solving*, which allows the symbolic computation of sets of data satisfying given constraints, and (iii) *coroutining* between the process of generating the data structures and the process of applying the filter. By using the above mechanisms we realize an efficient, automated data generation process following a *constrain-and-generate* computation pattern, which first generates data structure skeletons with constraints on its elements, and then generates random concrete values satisfying those constraints. Finally, these concrete data are translated back into inputs for the Erlang program under test.

The paper is structured as follows. In Sect. 2, we recall some basic notions on the Erlang and CLP programming languages. In Sect. 3, we present the framework for Property-Based Testing based on PropEr [27]. In Sect. 4, we show how from any given data type definition, written in the type language of Erlang, we derive a CLP generator for such data type. In Sect. 5, we describe our CLP interpreter for a sequential fragment of Erlang. In Sect. 6, we show the use of coroutining and, in Sect. 7, we present some experimental results obtained by the ProSyT tool, which implements our PBT technique. Finally, in Sect. 8, we compare our approach with related work in constraint-based testing.

2 Preliminaries

In this section we present the basic notions of the Erlang and CLP languages.

The Erlang language. Erlang is a concurrent, higher-order, functional programming language with dynamic, strong typing [36]. Its concurrency is based on the Actor model [19] and it allows asynchronous communications among processes. These features make the Erlang language suitable for distributed, fault-tolerant, and soft real-time applications. An Erlang program is a sequence of function definitions of the form: $f(X_1, \ldots, X_n)$ -> e, where f is the function name, $X_1, \ldots, X_n$ are variables, and e is an Erlang expression, whose syntax is shown in the box below, together with that of values and patterns. For reasons of simplicity, we have considered a subset of Core Erlang, that is, the intermediate language to which Erlang programs are translated by the Erlang/OTP compiler language [12]. This subset, in particular, does not include letrec expressions, nor commands for raising or catching exceptions, nor primitives for supporting concurrent computations.

$$
\begin{array}{rrcl}
\mathsf{Values} \ni & v & ::= & l \mid c(v_1, \ldots, v_n) \mid \mathsf{fun}(X_1, \ldots, X_n) \texttt{->} e \\
\mathsf{Patterns} \ni & p & ::= & p' \ \mathsf{when} \ g \\
& p' & ::= & l \mid X \mid c(X_1, \ldots, X_n) \\
\mathsf{Expressions} \ni & e & ::= & l \mid X \mid f \mid c(e_1, \ldots, e_n) \mid e_0(e_1, \ldots, e_n) \mid \mathsf{let}\ X = e_1\ \mathsf{in}\ e \\
& & & \mid \mathsf{case}\ e\ \mathsf{of}\ (p_1 \texttt{->} e_1); \ldots; (p_n \texttt{->} e_n)\ \mathsf{end} \mid \mathsf{fun}(X_1, \ldots, X_n) \texttt{->} e
\end{array}
$$

In these syntactic definitions: (i) by '$\mathsf{Values} \ni v$' we mean that v (possibly with subscripts) is a meta-variable ranging over Values, and analogously for Patterns and Expressions, (ii) l ranges over literals (such as integers, floats, atoms, and the empty list []), (iii) c is either the list constructor $[\_|\_]$ or the tuple constructor $\{\_, \ldots, \_\}$, (iv) X (possibly with subscripts) ranges over variables, (v) $\mathsf{fun}(X_1, \ldots, X_n)$ -> e denotes an anonymous function (we stipulate that the free variables in the expression e belong to $\{X_1, \ldots, X_n\}$), (vi) g ranges over *guards*, that is, boolean expressions (such as comparisons of terms using =<, ==, etc.) (vii) f ranges over function names.

The evaluation of an expression is performed in the *call-by-value* regime and returns a value. Variables are bound to values via the usual *pattern matching* mechanism. In Erlang each variable is bound to a value only once (this feature is known as *single assignment*). During the evaluation of a function call, the patterns of the `case-of` expression are considered, one after the other, in left-to-right order. The first pattern for which the pattern matching succeeds with a true guard, determines the corresponding expression to be evaluated. If there is no matching pattern with a true guard, a `match_fail` run-time error occurs.

The running example: *a faulty insertion program.* Below we show an Erlang function which is intended to insert an integer `I` in a list `L` of integers sorted in ascending order, thereby producing a new, extended sorted list. That function has an error as we have indicated in the line `ERR`. In what follows we will show how to automatically generate input values for detecting that error.

```
insert(I,L) -> case L of
 [] -> [I];
 [X|Xs] when I=<X -> [X,I|Xs];    % ERR: [X,I|Xs] should be [I,X|Xs]
 [X|Xs] -> [X] ++ insert(I,Xs)    % '++' denotes list concatenation
end.
```

Constraint Logic Programming. By CLP(X) we denote the CLP language based on constraints in the domain X, where X is: either (i) FD (the domain of integer numbers belonging to a finite interval), or (ii) R (the domain of floating point numbers), or (iii) B (the domain of boolean values) [20].

A *constraint* is a quantifier free, first-order formula whose variables range over the domain X. A *user-defined predicate* is a predicate symbol not present in the constraint language. An *atom* is an atomic formula $\mathtt{p}(\mathtt{t}_1, ..., \mathtt{t}_k)$, where $\mathtt{p}$ is a user-defined predicate and $\mathtt{t}_1, ..., \mathtt{t}_k$ are first-order terms constructed out of constants, variables, and function symbols. A CLP(X) program is a set of *clauses* of the form either `A.` or `A :- c, A1,..., An.`, where `A, A1,..., An` are atoms and `c` is a constraint on the domain X. A *query* is written as `?- c, A1,..., An.` A term, or an atom, is said to be *ground* if it contains no variables. As an example, below we list a CLP(FD) program for computing the `factorial` function ('`#>`' and '`#=`' denote the greater-than and equality relations, respectively):

```
factorial(0,1).
factorial(N,FN) :- N #> 0, M #= N-1, FN #= N*FM, factorial(M,FM).
```

For the operational semantics of CLP(X), we assume that, in the normal execution mode, constraints and atoms in a query are selected from left to right. In Sect. 6 we will see how the selection order is altered by using *coroutining* constructs (in particular, by using `when` declarations). When a constraint is selected, it is added to the *constraint store*, which is the conjunction of all constraints derived so far, thereby deriving a new constraint store. Then, the satisfiability of the new store is checked. The search for a clause whose head is unifiable with a given atom is done by following the textual top-down order of the program and, as usual for Prolog systems, the search tree is visited in a depth-first manner.

3 A Framework for PBT of Erlang Programs

In this section we introduce the fragment of the PropEr framework developed by Papadakis and Sagonas [27], which we use to specify PBT tasks. PropEr relies on a set of predefined functions for specifying the properties of interest for the Erlang programs. We consider the following PropEr functions.

- `?FORALL(Xs, XsGen, Prop)`, which is the main function used for property specification. `Xs` is either a variable, or a list of variables, or a tuple of variables. `XsGen` is a *generator* that produces a value for `Xs`. `Prop` is a boolean expression specifying a property that we want to check for the program under test. We assume that `Xs` includes all the free variables occurring in `Prop`. For instance, `?FORALL(X, integer(), mult1(X) >= X)` (i) uses the predefined generator `integer()`, which generates an integer, and (ii) specifies the property `mult1(X) >= X` for the function `mult1(X) -> X*(X+1)`.

- `?LET(Xs, XsGen, InExpr)`, which allows the definition of a *dependent generator*. `Xs` and `XsGen` are like in the `?FORALL` function, and `InExpr` is an expression whose free variables occur in `Xs`. `?LET(Xs, XsGen, InExpr)` generates a value obtained by evaluating `InExpr` using the value of `Xs` produced by `XsGen`. For instance, `?LET(X, integer(), 2*X)` generates an even integer.
- `?SUCHTHAT(Xs, XsGen, Filter)`, which allows the definition of a generator of values satisfying a given *filter* expression. `Xs` and `XsGen` are like in the `?FORALL` function, and `Filter` is a boolean expression whose free variables occur in `Xs`. `?SUCHTHAT(Xs, XsGen, Filter)` generates a value, which is a value of `Xs` produced by `XsGen`, for which the `Filter` expression holds `true`. For instance, `?SUCHTHAT(L, list(integer()), L=/=[])` generates non-empty lists of integers.

In PropEr a generator is specified by using: (i) type expressions, (ii) `?LET` functions, and (iii) `?SUCHTHAT` functions. We consider generators of first-order values only. However, higher-order functions may occur in `Prop`, `InExpr`, and `Filter`.

A *type expression* (whose semantics is a set of first-order values) is defined by using either the following PropEr *predefined types*:

- `any()`: all first-order Erlang values;
- `integer(L,H)`: the integers between `L` and `H` (these bounds can be omitted);
- `float(L,H)`: the floats between `L` and `H` (these bounds can be omitted);
- `atom()`: all Erlang atoms;
- `boolean()`: the boolean values `true` and `false`;

or PropEr *user-defined types*, which are defined by using type parameters and recursion, as usual. For instance, the type of binary trees with elements of a parameter type `T` can be defined as follows:

```
-type tree(T) :: 'leaf' | {'node',tree(T),T,tree(T)}.
```

Compound type expressions can be defined by the following *type constructors*:

- $\{\mathtt{T}_1, \ldots, \mathtt{T}_N\}$: the tuples of `N` elements of types $\mathtt{T}_1, \ldots, \mathtt{T}_N$, respectively;
- `list(T)`: the lists with elements of type `T`;
- $[\mathtt{T}_1, \ldots, \mathtt{T}_N]$: the lists of `N` elements of types $\mathtt{T}_1, \ldots, \mathtt{T}_N$, respectively;
- `union`$([\mathtt{T}_1, \ldots, \mathtt{T}_N])$: all elements x such that x is of type either $\mathtt{T}_1$ or $\ldots$ or $\mathtt{T}_N$;
- `exactly(lt)`: the singleton consisting of the literal `lt`.

Types can be used for specifying a *contract* [5] for an Erlang function `Func` by writing a declaration of the form:

```
-spec Func(ArgType1, ..., ArgTypeN) -> RetType.
```

A property is specified by declaring a nullary function (whose name, by convention, starts with `prop_`) of the form:

```
prop_name() -> ?FORALL(Xs, XsGen, Prop)
```

Here is an example of a *property specification*, which we will use for testing the `insert` function presented in Sect. 2.

[5] More detailed information about types and contract specifications can be found at http://erlang.org/doc/reference_manual/typespec.html.

```
prop_ordered_insert() ->                                % property_spec
  ?FORALL({E,L}, {integer(),ne_ordered_list()}, ordered(insert(E,L))).
ne_ordered_list() ->                                    % generator_1
  ?SUCHTHAT(L, non_empty(list(integer())), ordered(L)).
non_empty(T) -> ?SUCHTHAT(L, T, L=/=[]).                % generator_2
ordered(L) -> case L of                                 % filter
  [A,B|T] -> A =< B andalso ordered([B|T]);
  _ -> true
end.
```

In order to run the `prop_ordered_insert()` function, PropEr needs an *ad-hoc* implementation of the function `ne_ordered_list()` that generates a *non-empty ordered* list. If such a function is not provided by the user, PropEr executes the `ne_ordered_list()` generator in a very inefficient way by randomly generating non-empty lists of integers until it produces a list which is ordered [27, Sect. 4.2].

The main contribution of this paper is a technique that relieves the programmer from implementing generator functions and, instead, it derives efficient generators directly from their specifications. By doing so, we mitigate the problem of ensuring that the implementation of the generator is indeed correct, and we also avoid, in most cases, the inefficiency of a generate-and-test behavior by a suitable interleaving (via coroutining) of the process of data structure generation with the process of checking the constraint satisfaction (that is, filter evaluation). The implementation of our technique consists of the following six components.

1. A *translator from PropEr to CLP*, which translates the property specification, together with the definitions of Erlang/PropEr types and functions that are used, to a CLP representation.
2. A *type-based generator*, which implements a CLP predicate `typeof(X,T)` that generates datum `X` of any given (predefined or user-defined) type `T`. `typeof` queries can be run in a symbolic way, thereby computing for `X` a term containing variables, possibly subject to constraints.
3. A *CLP interpreter* for filter functions, that is, functions occurring in filter expressions. The interpreter handles the subset of the Core Erlang language presented in Sect. 2. In particular, it defines a predicate `eval(In,Env,Out)` such that, for an Erlang expression `In` whose variables are bound to values in an environment `Env`, `eval` computes, according to the semantics of Erlang, an output expression `Out`. The evaluation of `eval` is performed in a *symbolic* way, as the values in the bindings in `Env` may contain CLP variables, possibly subject to constraints. Thus, by running a query consisting of the conjunction of a `typeof` atom and an `eval` atom, we get as answer a term whose ground instances are values of the desired type, satisfying a given filter.
4. A *value generator*, which takes as input a term produced by running the type-based generator (Component 2) and then the interpreter (Component 3). The value generator can also be run immediately after the type-based generator, if no filter is present. Term variables, if any, may be subject to constraints. Concrete instances of the term (i.e., ground terms) satisfying these constraints are generated by choosing values (in a deterministic or random way) from the domains of the variables.

5. A *translator from CLP to Erlang*, which translates the values produced by the value generator (Component 4) to Erlang values.
6. A *property evaluator*, which evaluates, using the Erlang system, the boolean Erlang expression `Prop` whose inputs are the values produced by the translator (Component 5). Then the property evaluator checks whether or not one of the following three cases occurs: (i) `Prop` holds, (ii) `Prop` does not hold, or (iii) the evaluation of `Prop` crashes, that is, produces a runtime error.

The above six components have been implemented in a fully automatic tool, called ProSyT[6] (Property-Based Symbolic Testing).

4 Type-Based Value Generation

Type-based generation (Component 2 of our ProSyT tool) is achieved through the implementation of the `typeof` predicate. Given a type `T`, the predicate `typeof(X,T)` holds iff `X` is a CLP term encoding an Erlang value of type `T`. If `T` is a predefined type, `typeof` invokes a `T`-specific predicate for generating the term `X`. For example, for the type `list(A)`, that is, the type of the lists whose elements are of type `A`, `typeof` is implemented by the following clause:

```
typeof(X,list(A)) :- list(X,A).
```

where the binary predicate `list` is defined by the following two clauses:

```
list(nil,T).
list(cons(X,Xs),T) :- typeof(X,T), list(Xs,T).
```

where `nil` and `cons` are the CLP representations of the Erlang empty list and list constructor, respectively. If `T` is a user-defined type, `typeof` invokes the clause:

```
typeof(X,T) :- typedef(T,D), typeof(X,D).
```

where `typedef(T,D)` holds iff `D` is the (CLP representation of the Erlang) definition of type `T`. The clauses for `typedef` are generated during the translation from Erlang to CLP. For example, for the definition of the type `tree(T)` of binary trees, introduced in Sect. 3, we have the following clause:

```
typedef(tree(T), union([
       exactly(lit(atom,leaf)),
       tuple([exactly(lit(atom,node)),tree(T),T,tree(T)]) ])).
```

where: (i) `union([T1,T2])` denotes the union of the two types `T1` and `T2`, (ii) `exactly(E)` denotes a type consisting of the term `E` only, and (iii) `tuple(L)` denotes the type of the tuples $\{t_1, \ldots, t_n\}$ of terms such that t_i has the type specified by the `i`-th element of the list `L` of types.

Apart from the case when the type `T` is `exactly(lit(...))`, the query `?- typeof(X,T)` returns answers of the form `X=t`, where `t` is a non-ground term, whose variables may be subject to constraints. Here follow some examples of use of the `typeof` predicate. If we run the query `?- typeof(X,integer)` we get a single answer of the form `X=lit(int,_1320), _1320 in inf..sup`, where `_1320`

[6] https://fmlab.unich.it/testing/

is a variable that can take any integer value in the interval `inf..sup`, where `inf` and `sup` denote the minimum and the maximum integer, respectively. We can explicitly specify a range for integers. For instance, the answer to the query `?- typeof(X,integer(10,20))` is `X=lit(int,_1402), _1402 in 10..20.`

The query `?- typeof(X,list(integer))` produces a first answer of the form `X=nil`. If we compute an additional answer for that query, then we get `X=cons(lit(int,_1618), nil), _1618 in inf..sup` denoting the `nil` terminated list containing a single integer value. If we continue asking for additional answers, then by the standard Prolog execution mechanism, based on backtracking and depth-first search, we get answers with lists of increasing length.

When dealing with recursively defined data types, we have to care about the size of the generated terms, with the objective of avoiding the possible non-termination of the evaluation of the `typeof` predicate. The size of a term is defined to be the number of list or tuple constructors occurring in it. Thus, for instance, the term `lit(X,integer)` encoding an integer, has size `0`, and the size of a list of integers is equal to its length. The size of any term generated by `typeof` is constrained to be in the interval `min_size..max_size`, where `min_size` and `max_size` are configurable non-negative integers.

As an alternative to the built-in mechanisms for size management, we also provide the predicate `typeof(X,T,S)` which holds if `X` is a term of type `T` and size `S`. By using specific values of `S` or providing constraints on `S`, the user can specify the term size he desires and can control the answer generation process.

The user can also generate terms of *random* size, instead of terms of increasing size, as obtained by standard Prolog execution. For this purpose, we provide configuration options allowing `typeof` to generate data structures whose size is randomly chosen within the interval `min_size..max_size`.

It is also possible to use randomness during the generation of tuples and unions. For instance, every run of the query `?- typeof(X,tree(integer),2)` using standard Prolog execution, produces the same *first* answer, which is a tree consisting of the root and its right child. In order to modify such a behavior, we have introduced the `random_tuple` option that makes `typeof` generate tuples by randomly choosing one of its elements. (Recall that non-empty trees are indeed defined as tuples.) By doing so, the first answer to the above query is the tree consisting of the root and either its right child or its left child.

Similarly, for `union` types, we can introduce randomness through the use of the `random_union` option. For example, suppose that the type `color` has the two values `black` and `white` only. Thus, its translation into CLP is as follows:

```
typedef(color,union([exactly('black'),exactly('white')])).
```

Then, if we run the query `?- typeof(X,color)` using the standard Prolog execution mechanism, we will always obtain `black` as the first answer. However, if we use the `random_union` option we may get either `black` or `white` with equal frequency. More in general, we provide a `weighted_union` type, which allows the association of frequencies with types using non-negative integers, so that elements of types with higher frequencies are generated more often.

Random generation of *ground* terms (Component 4 of ProSyT) is achieved through the use of the `rand_elem(X)` predicate. For example, the clauses used for the generation of (possibly, non-flat) lists of integers are the following ones:

```
rand_elem(nil).
rand_elem(cons(X,L)) :- rand_elem(X), rand_elem(L).
rand_elem(lit(int,V)) :- rand_int(V).
rand_int(V) :- int_inf(V,Inf), int_sup(V,Sup), random_between(Inf,Sup,V).
```

where `rand_int(V)` holds iff `V` is a random integer value in the range `Inf..Sup`, `Inf` and `Sup` being the minimum and maximum values that `V` can take, subject to the constraints that are present in the constraint store. For instance, the query: `?- typeof(X,list(integer(1,10)),3), rand_elem(X).` may return the answer:

```
X = cons(lit(int,6), cons(lit(int,4), cons(lit(int,9), nil))).
```

A similar mechanism is used for generating ground terms containing floats.

Finally, ground CLP terms are translated to Erlang values (Component 5 of ProSyT) by using the `write_elem` predicate. For instance, if we append `write_elem(X)` to the above query, we get the Erlang list `[6,4,9]`.

5 The Interpreter of Filter Functions

The CLP interpreter, which is Component 3 of ProSyT, provides the predicate `eval(In,Env,Out)` that computes the output value `Out` of the input expression `In` in the environment `Env`. The environment `Env`, which maps variables to values, is represented by a list of pairs of the form `('X',V)`, where `'X'` is the CLP constant representing the Erlang variable `X` and `V` is the CLP term representing its value. By means of a symbolic representation of Erlang expressions and values occurring in the environment (by using possibly non-ground CLP terms subject to suitable constraints), the evaluation of any input expression via the interpreter allows the exhaustive exploration of all program computations without explicitly enumerating all the concrete input values.

In the interpreter, a function definition is represented by a CLP term of the form `fun(Pars,Body)`, where `Pars` and `Body` are the formal parameters and the function body, respectively. As an example of how the interpreter is defined, Fig. 1 lists the CLP implementation of the semantic rule for function application, represented by a term of the form `apply(Func,IExps)`, where `Func` is the name of the function to be applied to the actual parameters `IExps`.

The behavior is as follows. First, the interpreter retrieves (at line 1) the definition of the function `Func`. Then, it evaluates (at line 2) the actual parameters `IExps` in the environment `Env`, thereby deriving the list of expressions `AExps`. Then, the interpreter binds (at line 3) the formal parameters `Pars` to

```
eval(apply(Func,IExps),Env,Out) :-
  fundef(Func,fun(Pars,Body)),          % 1
  eval_args(IExps,Env,AExps),           % 2
  zip_binds(Pars,AExps,Binds),          % 3
  constrain_output_exp(Func,Out),       % 4
  eval(Body,Binds,Out).                 % 5
```

Fig. 1. CLP interpreter for applying the function `Func` to the actual parameters `IExps`.

the actual parameters `AExps`, thereby deriving the new environment `Binds`. If a contract for `Func` has been provided (see Sect. 3) and the `--force-spec` option of ProSyT has been used, then (at line `4`) a constraint is added on the CLP variables occurring in the output expression `Out`. For instance, let us suppose that the programmer specifies the following contract for the `listlength` function that computes the length of a list:

```
-spec listlength(list(any())) -> non_neg_integer().
```

where the `non_neg_integer()` type requires the output of `listlength` on lists of any type to be a non-negative integer. Thus, the constraint `M#>=0` is imposed on the CLP variable `M` occurring in the output expression `lit(int,M)` computed by `listlength`. Finally, the interpreter evaluates (at line `5`), the `Body` of the function `Func` in the new environment `Binds`, thereby deriving the output expression `Out`.

Now, let us consider the filter function `ordered_list` of Sect. 3. In order to generate symbolic ordered lists, which will be used for producing the test cases for `insert`, we run the following query:

```
?- typeof(I,non_empty(list(integer))),
   eval(apply(var('ordered',1),[var('L')]),[('L',I)],lit(atom,true)).
```

In the above query `eval` calls `ordered` in an environment where the `'L'` parameter is bound to `I`, and outputs an expression denoting the atom `true`. As a result of query evaluation, `typeof` binds the CLP variable `I` to a nonempty list of integers and `eval` adds constraints on the elements of the list enforcing them to be in ascending order. Among the first answers to the query we obtain:

```
I = cons(lit(int,_54),cons(lit(int,_55),nil)), _55#>=_54 ;
I = cons(lit(int,_51),cons(lit(int,_52),cons(lit(int,_53),nil))),
    _52#>=_51, _53#>=_52
```

Then, by running the predicates `rand_elem` and `write_elem`, for each non-ground list whose elements are in ascending order, we can (randomly) generate one or more ordered Erlang lists, without backtracking on the generation of lists whose elements are not ordered. The following command runs ProSyT on the file `ord_insert_bug.erl` that includes the bugged `insert` function and the `prop_ordered_insert()` property specification.

```
$ ./prosyt.sh ord_insert_bug.erl prop_ordered_insert
```

By default, ProSyT runs 100 tests by generating non-ground ordered lists of increasing length, which are then instantiated by choosing integers from the interval `-1000..1000`. The 100 tests produce as output a string of 100 characters such as (we show an initial part of that string only):

```
x.x.xxxxxxx...xxxx.xxxxxxx.xxxxx.xxxx..
```

Each character represents the result of performing a test case: (i) the character '`.`' means that the desired property `prop_ordered_insert` holds, and (ii) the character '`x`' means that it does not hold, hence revealing a bug.

The generation of the ordered lists for the 100 test cases takes 42ms (user time) on an Intel® Core™ i7-8550U CPU with 16GB running Ubuntu 18.04.2.

6 Coroutining the Type-Based Generator and the Filter Interpreter

The process of symbolic test case generation described in the previous section has a good performance when the filter does not specify constraints on the *skeleton* of the data structure, but only on its *elements*. For instance, in the case of ordered lists, the filter `ordered(L)` does not enforce any constraint on the length of the symbolic list `L` generated by the type-based generator, but only on its elements.

Now, let us consider the following filter function `avl`, which checks whether or not a given binary tree is an AVL tree, that is, a *binary search tree* that satisfies the constraint of being *height-balanced* [10].

```
avl(T) -> case T of
   leaf -> true;
   {node,L,V,R} ->
     B = height(L)-height(R) andalso B >= -1 andalso B =< 1 andalso  % 1
     ltt(L,V)  andalso  gtt(R,V)  andalso                            % 2
     avl(L)  andalso  avl(R);
    _ -> false
 end.
```

The recursive clause of the `case-of` checks whether or not any tree {`node,L,V,R`} rooted in `V` (the value of the node) is height-balanced (line `1`), all the values in the left subtree `L` are smaller than `V`, and all the values in the right subtree `R` are larger than `V` (line `2`). `avl` uses the following utility functions: (i) `height(T)`, which returns the height of the tree `T`, (ii) `ltt(T,V)`, and (iii) `gtt(T,V)`, which return `true` if the value of each node n in the tree `T` is less than, or greater than `V`, respectively. In order to generate AVL trees, we run the following query:

```
?- typeof(X,tree(integer)),
   eval(apply(var('avl',1),[var('T')]),[('T',X)],lit(atom,true)),
   rand_elem(X).
```

However, unlike the case of ordered lists, among the answers to the query `typeof(X,tree(integer))` just a few instances of `X` turn out to be AVL trees. Hence, `eval` repeatedly fails until `typeof` generates a binary tree satisfying the constraints specified by the filter. As an example, for trees of size 10, `eval` finds 10 AVL trees out of 9000 trees generated by `typeof`.

In order to make the generation process more efficient, we use the *coroutining* mechanism to implement a data-driven cooperation [23], thereby interleaving the execution of the type-based generator `typeof` and that of the interpreter `eval`. Coroutining is obtained by interchanging the order of the `typeof` and `eval` atoms in the query, so that the `eval` call is selected before the `typeof` call. However, the execution of `eval` is *suspended* on inputs of the filter function that are not instantiated enough to decide which clauses of a `case-of` expression can be used to proceed in the function evaluation. The execution of `eval` is then *resumed* whenever the input to the filter function gets further instantiated by the `typeof` execution. By doing so, during the generation of complex data structures, `typeof` must comply with the constraints enforced by `eval`. This mechanism

can dramatically improve efficiency, because the unsatisfiability of the given constraints may be detected before the entire data structure is generated.

Coroutining is implemented by using the `when(Cond,Goal)` primitive provided by SWI-Prolog [35], which suspends the execution of `Goal` until `Cond` becomes true. In particular, `when` declarations are used in the rule of the interpreter shown below, which defines the operational semantics of `case-of` expressions.

```
eval(case(CExps,Cls),Env,Exp) :-
  eval(CExps,Env,EExps),
  suspend_on(Env,EExps,Cls,Cond),
  when(Cond,( match(Env,EExps,Cls,MEnv,Cl), eval(Cl,MEnv,Exp) )).
```

The evaluation of expressions of the form '`case CExps of Cls end.`', encoded as `case(CExps,Cls)`, in the environment `Env` behaves as follows. The expressions `CExps` are evaluated in the environment `Env`, thereby getting the expressions `EExps` to be matched against one of the patterns of the clauses `Cls`. Then, `suspend_on(Env,EExps,Cls,Cond)` generates a condition `Cond` of the form `( nonvar(`V_1`), ..., nonvar(`V_n`) )`, where $V_1, \ldots, V_n$ are the CLP variables occurring in `EExps` that would get bound to either a list or a tuple while matching the expressions `EExps` against the patterns of the clauses `Cls`. Such a condition forces the suspension of the evaluation of the goal occurring as a second argument of the `when` primitive until all of these variables get bound to a non-variable term. If the evaluation of the `case-of` binds all the variables to terms which are neither lists nor tuples, then `suspend_on` produces a `Cond` that holds `true`. Thus, when the goal of the `when` primitive is executed: (i) `match(Env,EExps,Cls,MEnv,Cl)` selects a clause `Cl` from `Cls` whose pattern matches `EExps`, hence producing the environment `MEnv` that extends `Env` with the new bindings derived by matching, and (ii) `eval(Cl,MEnv,Exp)` evaluates `Cl` in `MEnv` and produces the output expression `Exp`. Now, if we run the following query:

```
?- eval(apply(var('avl',1),[var('T')]),[('T',X)],lit(atom,true)),
   typeof(X,tree(integer)),
   rand_elem(X).
```

the CLP variable `X`, shared between `typeof` and `eval`, forces the type-based generator and the filter to cooperate in the generation of AVL trees. Indeed, as soon as the `typeof` (partially) instantiates `X` to a binary tree, the evaluation of the filter function adds constraints on the skeleton of `X` (corresponding to the properties at lines `1` and `2` of the definition of the `avl` function). The advantage of this approach is that the constraints on `X` restrict the possible ways in which its left and right subtrees can be further expanded by recursive calls of `typeof`. As an example, suppose we want to test the following function `avl_insert` that inserts the integer element `E` into the AVL tree `T`:

```
avl_insert(E,T) -> case T of
 {node,L,V,R} when E < V -> re_balance(E,{node,avl_insert(E,L),V,R});
 {node,L,V,R} when E > V ->
   re_balance(E,{node,L,V,avl_insert(E,R)});
 {node,L,V,R} when E == V -> {node,L,V,R};
 leaf -> {node,leaf,E,leaf}
end.
```

This function uses the following utility functions shown below: (i) `re_balance`, which given an integer element `E` and a binary search tree `T`, performs suitable rotations on `T` so as to make it height-balanced, and (ii) `right_rotation`, and (iii) `left_rotation`, which perform a right rotation, and a left rotation on `T`, respectively. The definition of `re_balance` has two errors: (1) at line `ERR_1`, where '`<`' should be '`>`', and (2) at line `ERR_2`, where '`>`' should be '`<`'.

```
re_balance(E,T) ->
 {node,L,V,R} = T,
 case height(L) - height(R) of
   2 -> {node,_,LV,_} = L,              % Left unbalanced tree
   if E < LV -> right_rotation(T);
      E > LV -> right_rotation({node,left_rotation(L),V,R})
   end;
  -2 -> {node,_,RV,_} = R,              % Right unbalanced tree
   if E < RV -> left_rotation(T);                                  % ERR_1
      E > RV -> left_rotation({node,L,V,right_rotation(R)})    % ERR_2
   end;
   _ -> T
 end.

right_rotation({node,{node,LL,LV,LR},V,R}) ->
  {node,LL,LV,{node,LR,V,R}}.

left_rotation({node,L,V,{node,RL,RV,RR}}) ->
  {node,{node,L,V,RL},RV,RR}.
```

The following test case specification states that after inserting an integer element `E` in an AVL tree, we get again an AVL tree:

```
avl() -> ?SUCHTHAT(T, tree(integer()), avl(T)).
prop_insert() ->
   ?FORALL({E,T}, {integer(),avl()}, avl(avl_insert(E,T))).
```

The following command runs ProSyT on the file `avl_insert_bug.erl` that includes the above bugged `avl_insert` function and the test case specification.

```
$ ./prosyt.sh avl_insert_bug.erl prop_insert --force-spec\
  --min-size 3 --max-size 20 --inf -10000 --sup 10000 --tests 200
```

In this command we have used the following options:
(i) `--min-size` and `--max-size` specify the values of `min_size` and `max_size`, respectively, determining the size of the data structure (see Sect. 3), (ii) `--inf` and `--sup` specify the bounds of the interval where integer elements are taken from (see Sect. 3), and (iii) `--tests N` specifies the number of tests to be run.

Here is an initial part of the string of characters we get:

```
..x...x...cx..x.xxc....x..c.x..x..x.c..
```

The generation of the 200 test cases takes 550ms (user time). Several '`x`' characters are generated, corresponding to runs of `avl_insert` that do not return an AVL tree, and hence reveal bugs. Moreover, the '`c`' characters in the output string correspond to crashes of the execution, due to the fact that the `right_rotation` or `left_rotation` functions threw a `match_fail` exception when applied to a tree on which those rotations cannot be performed.

7 Experimental evaluation

In this section we present the experimental evaluation we have performed for assessing the effectiveness and the efficiency of the test case generation process we have presented in this paper and we have implemented in ProSyT.

Benchmark suite. The suite consists of 10 Erlang programs: (1) `ord_insert`, whose input is an integer and an ordered list of integers (see Sect. 2); (2) `up_down_seq`, whose input is a list of integers of the form: $[w_1, \ldots, w_m, z_1, \ldots, z_m]$, with $w_1 \leq \ldots \leq w_m$ and $z_1 \geq \ldots \geq z_m$; (3) `n_up_seqs`, whose input is a list of ordered lists of integers of increasing length; (4) `delete`, whose input is a triple $\langle w, u, v \rangle$ of lists of integers such that w is the ordered permutation of the list obtained by concatenating the ordered lists u and v; (5) `stack`, whose input is a pair $\langle s, n \rangle$, where s is a stack encoded as a list of integers, and n is the length of that list; (6) `matrix_mult`, whose input is a pair of matrices encoded as a pair of lists of lists, (7) `det_tri_matrix`, whose input is a lower triangular matrix encoded as a list of lists of increasing length of the form: $[[v_{11}], [v_{21}, v_{22}], \ldots, [v_{n1}, \ldots, v_{nn}]]$, (8) `balanced_tree`, whose input is a height-balanced binary tree [10]; (9) `binomial_tree_heap`, whose input is a binomial tree satisfying the minimum-heap property [10]; (10) `avl_insert`, whose input is an AVL tree (see Sect. 6). The benchmark suite is available online as part of the ProSyT tool (the suffixes `_bug.erl` and `_ok.erl` denotes the buggy and correct versions of the programs, respectively).

Experimental processes. We have implemented the following two experimental processes. (*i*) *Generate-and-Test*, which runs PropEr for randomly generating a value of the given data type, and then tests whether or not that value satisfies the given filter; this process uses the predefined generator for lists and a simple user-defined generator for trees. (*ii*) *Constrain-and-Generate*, which runs ProSyT by coroutining the generation of the skeleton of a data structure and the evaluation of the filter expression (see Sect. 6), and then randomly instantiating that skeleton.

Technical resources. The experimental evaluation has been performed on a machine equipped with an Intel® Core™ i7-8550U CPU @ 1.80GHz × 8 processor and 16GB of memory running Ubuntu 18.04.2 LTS. The timeout limit for each run of the test cases generation process has been set to 300 seconds.

Results. We have run PropEr and ProSyT for generating up to 100,000 test cases whose size is in the interval $[10, 100]$, and we made the random generator for integer and real values to take values in the interval $[-10000, 10000]$. ProSyT has been configured so that the random instantiation phase can produce at most 1500 concrete test cases for every data structure skeleton found. The results of the experimental evaluation are summarized in Table 1.

The experiments show that the *Constrain-and-Generate* process used by ProSyT performs much better than the *Generate-and-Test* process used by PropEr. Indeed, *Generate-and-Test* is able to find valid test cases only when the filter is very simple. Actually, in some examples, PropEr generates test cases with very small size only (less than the minimum specified size limit of 10).

Program	PropEr		ProSyT	
	Time	*N*	*Time*	*N*
1. `ord_insert`	300.00	0	300.00	67,083
2. `up_down_seq`	300.00	0	300.00	22,500
3. `n_up_seqs`	300.00	0	300.00	24,000
4. `delete`	300.00	0	9.21	100,000
5. `stack`	143.71	100,000	19.57	100,000
6. `matrix_mult`	300.00	0	300.00	76,810
7. `det_tri_matrix`	300.00	304	32.28	13,500
8. `balanced_tree`	300.00	121	21.54	100,000
9. `binomial_tree_heap`	300.00	0	43.45	4,500
10. `avl_insert`	300.00	0	300.00	23,034

Table 1. Column *Time* reports the seconds needed to generate N ($\leq$ 100,000) test cases of size in the interval $[10, 100]$ within the time limit of 300 seconds.

In particular, for the `ord_insert` program, PropEr generates ordered lists of length at most 8, while ProSyT is able to generate lists of length up to 53. Also for the programs `det_tri_matrix` and `balanced_tree`, the size of the largest data structure generated by PropEr (a 5×5 matrix and a 15 node balanced tree) is much smaller than the largest data structure generated by ProSyT (a 12×12 matrix and a 22 node balanced tree). Finally, note that for the programs `det_tri_matrix` and `binomial_tree_heap`, ProSyT halted before the time limit of 300 seconds because no more skeletons exist within the given size interval.

8 Related Work

Automated test generation has been suggested by many authors as a means of reducing the cost of testing and improving its quality [32]. Property-Based Testing, and in particular the QuickCheck approach [8], is one of the most well-established methods for automating test case generation (see also the references cited in the Introduction).

PropEr [27,30] is a popular Property-Based Testing tool for Erlang programs that follows the QuickCheck approach. PropEr was proposed as an open-source alternative to Quviq QuickCheck [2], a proprietary, closed-source tool for Property-Based Testing of Erlang programs. In addition, PropEr was designed to be integrated with the Erlang type specification language.

However, a critical point of PropEr (and of other PBT frameworks) is that the user bears most of the burden of writing correct, efficient generators of test data. Essentially, PropEr only provides an automated way for converting type specifications into generators of *free* data structures, but very limited support is given to automatically generate data structures subject to *constraints*, such as the sorted lists and AVL-trees we have considered in this paper. In this respect, the main contribution of our work is a technique for the automated generation of test data from PropEr specifications. Indeed, our approach, based on a CLP

interpreter for (a subset of) Erlang, allows the automated generation of test data in an efficient way. Test data are generated by interleaving, via coroutining, the data structure generation, the filtering of those data structures based on constraint solving, and the random instantiation of variables. This mechanism, implemented in the ProSyT tool, has demonstrated good efficiency on some non-trivial examples.

The work closest to ours is the one implemented by the FocalTest tool [6]. FocalTest is a PBT tool designed to generate test data for programs and properties written in the functional language Focalize. Its main feature is a translation of Focalize programs into CLP(FD) programs extended with the *constraint combinators* `apply` and `match`, which encode function application and pattern matching, respectively. `apply` and `match` are implemented by using *freeze* and *wake-up* mechanisms based on the instantiation of logical variables, and in particular, the evaluation of `apply` and `match` is waken up when their arguments are bound to non-variable terms.

A difference between FocalTest and ProSyT comes from the fact that, Focalize is a statically typed language and Erlang is a dynamically typed language. Static type information is used by FocalTest for instantiating variables, while in ProSyT type-based instantiation is performed through the `typeof` data structure generator. Static typing is also exploited in the proof of correctness of FocalTest, whose operational semantics has been formalized in Coq [5]. In contrast, we handle Erlang's dynamic typing discipline by writing an interpreter of (a subset of) the language, which also models failure due to runtime typing errors.

The development of the CLP interpreter for Erlang and, more in general, for the PropEr framework, is indeed the most significant and distinctive feature of our approach. From a methodological point of view, a direct implementation of the operational semantics in a rule-based language like CLP, requires a limited effort for the proof of correctness with respect to a formal semantics (we did not deal with this issue in the present paper, but we tested our interpreter on several examples). From a practical point of view, the use of the interpreter avoids the need of extending CLP with special purpose constraint operators like `apply` and `match`. Moreover, our interpreter-based approach lends itself to possible optimizations for improving the efficiency of test case generation, such as *partial evaluation* [21], for automatically deriving specialized test case generators.

The freeze and wake-up mechanisms used by FocalTest are quite related to the coroutining mechanism implemented by ProSyT, which, however, is realized by the interpreter, rather than by the constraint solving strategy.

Other differences between FocalTest and ProSyT concern numerical variables and random instantiation. FocalTest handles integer numerical variables using the CLP(FD) constraint solver and randomly instantiates those variables using a strategy called *random iterative domain splitting*. ProSyT handles integer and float numerical variables using CLP(FD) and CLP(R), respectively, for solving constraints on those variables, and their random instantiation is done by using CLP(FD) and CLP(R) built-ins. ProSyT is also able to perform random gener-

ation of data structures, by using a randomized version of the predicate `typeof` (see Sect. 4), possibly specifying a distribution for the values of a given type.

The idea of interleaving, via coroutining, the generation of a data structure with the test of consistency of the constraints that the data structure should satisfy, is related to the lazy evaluation strategy used by *Lazy SmallCheck* [31], a PBT tool for Haskell. Lazy SmallCheck checks properties for partially defined values and lazily evaluates parallel conjunction to enable early pruning of the set of candidate test data. Lazy SmallCheck does not use symbolic constraint solving, and exhaustively generates all values up to a given bound.

Besides functional programming languages, PBT has also been applied to Prolog [1]. Similarly to ProSyT, the PrologCheck tool [1] implements randomized test data generation for Prolog. However, when the test data specification contains constraints, PrologCheck follows a *generate-and-test* approach, and no mechanism is provided by the tool for coroutining data generation and constraint solving (unless this is coded directly by the programmer).

The use of constraint-based reasoning techniques for test-case generation is a well-established approach [11,16,18,25], which has been followed for the implementation of several tools in various contexts. Among them, we would like to mention: GATeL [26], a test sequence generator for the synchronous dataflow language LUSTRE, AUTOFOCUS [29], a model-based test sequence generator for reactive systems, JML-TT [3], a tool for model-based test case generation from specifications written in the Java Modeling Language (JML), Euclide [17], a tool for testing safety-critical C programs, and finally, tools for *concolic testing*, such as PathCrawler [37], CUTE [34], and DART [15], which combine concrete execution with constraint-based path representations of C programs.

Our work is also related to approaches and tools proposed in the context of languages for specifying and testing meta-logic properties of formal systems. In particular, αCheck [7] follows an approach very much inspired by PBT for performing bounded model-checking of formal systems specified in αProlog, which is a Horn clause language based on nominal logic. Related concepts are at the basis of QuickChick, a PBT tool for the Coq proof assistant [28]. Lampropoulos et al. [24] also address the problem of deriving correct generators for data satisfying suitable inductive invariants on top of QuickChick. In that work, the mechanism for data generation makes use of the narrowing technique, which similarly to our resolution-based approach, builds upon the unification algorithm.

Declarative approaches for test data generation have been proposed in the context of *bounded-exhaustive testing* (BET) [9], whose goal is to test a program on all input data satisfying a given invariant, up to a fixed bound on their size. One of the most well-known declarative frameworks for BET is Korat [4], which is a tool for testing Java programs. Given a Java predicate specifying a data structure, subject to a given invariant and a size bound on its input, Korat uses backtracking to systematically explore the bounded input space of the predicate by applying a generate-and-test strategy. JMLAutoTest [38] implements a technique, based on statistical methods, for avoiding the generation of many useless test cases by exploiting JML specifications.

A different domain-specific language for BET of Java programs is UDITA [14]. It provides non-deterministic choice operators and an interface for generating linked structures. UDITA improves efficiency with respect to the generate-and-test approach by applying the *delayed choice* principle, that is, postponing the instantiation of a variable until it is first accessed.

It has been shown that CLP-based approaches, which exploit built-in unification and special purpose constraint solving algorithms, can be very competitive with respect to domain-specific tools for BET [13].

9 Conclusions

We have presented a technique for automated test case generation from *test case specifications*. We have considered the Erlang functional programming language and a test case specification language based on the PropEr framework [27,30].

In this paper we have shown how we can relieve the programmer from writing generators of test data by using constraint logic programming (CLP). However, even if our approach to automated PBT is based on CLP, the programmer is not required to deal with any concept related to logic programming, and Prolog code is fully transparent to the programmer. Indeed, we provide both (i) a translator from PropEr and Erlang specifications and programs to CLP, and (ii) a translator of the generated test data from CLP syntax to Erlang syntax.

At present, the ProSyT tool, which implements our PBT technique, does not provide any *shrinking* mechanism to try to generate an input of *minimal* size in case the program under test does not satisfy the property of interest. However, we think that this mechanism can efficiently be realized by using the primitives for controlling term size provided by our tool, together with Prolog default search strategy based on backtracking.

Finally, we would like to notice that, even if we developed our technique in the context of PBT of Erlang programs, the approach we followed is to a large extent independent of the specific programming language, as it is based on writing a CLP interpreter of the programming language under consideration. Thus, as future work, we plan to apply a similar scheme to other programming languages by providing suitable CLP interpreters.

Acknowledgements

We would like to thank the anonymous reviewers for their very helpful and constructive comments.

E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti are members of the INdAM Research group GNCS. E. De Angelis, F. Fioravanti, and A. Pettorossi are research associates at CNR-IASI, Rome, Italy.

A. Palacios was partially supported by the EU (FEDER) and the Spanish *Ayudas para contratos predoctorales para la formación de doctores* and *Ayudas a la movilidad predoctoral para la realización de estancias breves en centros de I+D* (MICINN) under FPI grants BES-2014-069749 and EEBB-I-17-12101.

References

1. C. Amaral, M. Florido, and V. S. Costa. PrologCheck – Property-Based Testing in Prolog. In M. Codish and E. Sumii, editors, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4–6, 2014*, Lecture Notes in Computer Science 8475, pages 1–17. Springer, 2014.
2. Th. Arts, J. Hughes, J. Johansson, and U. T. Wiger. Testing telecoms software with Quviq QuickCheck. In M. Feeley and Ph. W. Trinder, editors, *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, September 16, 2006*, pages 2–10. ACM, 2006.
3. F. Bouquet, F. Dadeau, and B. Legeard. Automated boundary test generation from JML specifications. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods*, Lecture Notes in Computer Science 4085, pages 428–443. Springer Berlin Heidelberg, 2006.
4. C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 123–133, New York, NY, USA, 2002. ACM.
5. M. Carlier, C. Dubois, and A. Gotlieb. A first step in the design of a formally verified constraint-based testing tool: FocalTest. In A.D. Brucker and J. Julliand, editors, *Tests and Proofs - 6th International Conference, TAP 2012, Prague, Czech Republic, May 31–June 1, 2012. Proceedings*, Lecture Notes in Computer Science 7305, pages 35–50. Springer, 2012.
6. M. Carlier, C. Dubois, and A. Gotlieb. FocalTest: A Constraint Programming Approach for Property-Based Testing. In J. Cordeiro, M. Virvou, and B. Shishkov, editors, *Software and Data Technologies - 5th International Conference, ICSOFT 2010, Athens, Greece, July 22–24, 2010. Revised Selected Papers*, Communications in Computer and Information Science 170, pages 140–155. Springer, 2013.
7. J. Cheney and A. Momigliano. αCheck: A mechanized metatheory model checker. *Theory and Practice of Logic Programming*, 17(3):311–352, 2017.
8. K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In M. Odersky and Ph. Wadler, editors, *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18–21, 2000*, pages 268–279. ACM, 2000.
9. D. Coppit, W. Le, K. J. Sullivan, S. Khurshid, and J. Yang. Software assurance by bounded exhaustive testing. *IEEE Transactions on Software Engineering*, 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. J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J. Woodcock and P. G. Larsen, editors, *FME '93: Industrial-Strength Formal Methods, Proceedings of the 1st International Symposium of Formal Methods Europe, Odense, Denmark, April 19–23, 1993*, Lecture Notes in Computer Science 670, pages 268–284. Springer, 1993.
12. R. Carlsson et al. Core Erlang 1.0.3. language specification. Technical Report, Department of Information Technology, Uppsala University, Uppsala, Sweden, `www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf`, 2004.
13. F. Fioravanti, M. Proietti, and V. Senni. Efficient generation of test data structures using constraint logic programming and program transformation. *Journal of Logic and Computation*, 25(6):1263–1283, 2015.

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, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, May 2–8 2010, Cape Town, South Africa*, pages 225–234. ACM, 2010.
15. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In V. Sarkar and M. W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12–15, 2005*, pages 213–223. ACM, 2005.
16. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Test case generation for object-oriented imperative languages in CLP. *Theory and Practice of Logic Programming*, 10(4–6):659–674, 2010.
17. A. Gotlieb. Euclide: A Constraint-Based Testing Framework for Critical C Programs. In *2nd International Conference on Software Testing, Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1–4, 2009*, pages 151–160. IEEE Computer Society, 2009.
18. A. Gotlieb, B. Botella, and M. Rueher. A CLP framework for computing structural test data. In Lloyd J. et al., editor, *Computational Logic - CL 2000*, Lecture Notes in Computer Science 1861, pages 399–413. Springer, Berlin, Heidelberg, 2000.
19. C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI '73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.
20. J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
21. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
22. junit-quickcheck: Property-based testing, JUnit-style `https://github.com/pholser/junit-quickcheck`.
23. R. A. Kowalski. *Logic for Problem Solving*. North Holland, 1979.
24. L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce. Generating good generators for inductive relations. *Proceedings of the ACM on Programming Languages*, 2:45:1–45:30, 2017.
25. B. Marre. Toward automatic test data set selection using algebraic specifications and logic programming. In K. Furukawa, editor, *Logic Programming, Proceedings of the 8th International Conference, Paris, France, June 24–28, 1991*, pages 202–219. MIT Press, 1991.
26. B. Marre and A. Arnould. Test sequences generation from LUSTRE descriptions: GATeL. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering, ASE 2000, Grenoble, France, September 11–15, 2000*, page 229. IEEE Computer Society, 2000.
27. M. Papadakis and K. Sagonas. A PropEr Integration of Types and Function Specifications with Property-Based Testing. In K. Rikitake and E. Stenman, editors, *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang, Tokyo, Japan, September 23, 2011*, pages 39–50. ACM, 2011.
28. Z. Paraskevopoulou, C. Hritcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational Property-Based Testing. In Ch. Urban and X. Zhang, editors, *Interactive Theorem Proving - Proceedings of the 6th International Conference, 2015, Nanjing, China, August 24–27, 2015*, Lecture Notes in Computer Science 9236, pages 325–343. Springer, 2015.

29. A. Pretschner and H. Lötzbeyer. Model based testing with constraint logic programming: First results and challenges. In *Proceedings of the 2nd ICSE Workshop on Automated Program Analysis, Testing and Verification (WAPATV)*, pages 1–9, 2001.
30. PropEr: Property-Based Testing for Erlang. `http://proper.softlab.ntua.gr/`
31. C. Runciman, M. Naylor, and F. Lindblad. SmallCheck and Lazy SmallCheck: Automatic exhaustive testing for small values. In A. Gill, editor, *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, pages 37–48. ACM, 2008.
32. J. M. Rushby. Automated test generation and verified software. In B. Meyer and J. Woodcock, editors, *Verified Software: Theories, Tools, Experiments, 1st IFIP TC2/WG2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10–13, 2005, Revised Selected Papers and Discussions*, Lecture Notes in Computer Science 4171, pages 161–172. Springer, 2008.
33. ScalaCheck: Property-Based Testing for Scala. `http://www.scalacheck.org/`
34. K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In M. Wermelinger and H. C. Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5–9, 2005*, pages 263–272. ACM, 2005.
35. The SWI Prolog Logic Programming System. `http://www.swi-prolog.org/`
36. R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang (2nd Ed.)*. J. Armstrong, editor. Prentice Hall International Ltd., Hertfordshire, UK, 1996.
37. N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In M. Dal Cin, M. Kaâniche, and A. Pataricza, editors, *Proceedings of the 5th European Dependable Computing Conference, EDCC-5, Budapest, Hungary, April 20–22, 2005*, Lecture Notes in Computer Science 3463, pages 281–292. Springer, 2005.
38. G. Xu and Z. Yang. JMLAutoTest: A novel automated testing framework based on JML and JUnit. In A. Petrenko and A. Ulrich, editors, *Formal Approaches to Software Testing. FATES 2003*, Lecture Notes in Computer Science 2931, pages 70–85. Springer Berlin Heidelberg, 2004.
39. K. Yatoh, K. Sakamoto, F. Ishikawa, and S. Honiden. ArbitCheck: A Highly Automated Property-Based Testing Tool for Java. In *Proceedings of the 7th IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2014, March 31–April 4, 2014, Cleveland, Ohio, USA*, pages 405–412. IEEE Computer Society, 2014.