Multi-Configurable Search Rules in Prolog and Application to Testing*

Daniela Ferreiro[1,2][0009−0002−1072−8989], Jose F. Morales[1,2][0000−0001−9782−8135], Pedro López-García[1,3][0000−0002−1092−2071], and Manuel V. Hermenegildo[1,2][0000−0002−7583−323X]

[1] Universidad Politécnica de Madrid (UPM), Madrid, Spain
[2] IMDEA Software Institute, Madrid, Spain
[3] Spanish Council for Scientific Research, Madrid, Spain
{daniela.ferreiro,josef.morales,pedro.lopez,manuel.hermenegildo}@imdea.org

Abstract. Prolog systems traditionally employ leftmost, depth-first search as their execution strategy. This choice is well-justified for efficiency reasons, generally accepted, and useful in practice. However, it is also well-known that it can lead to incompleteness when evaluating programs over infinite search spaces and may not be ideal for complex search spaces. We revisit the role of search strategies in Prolog programs, and present a new approach, that enables programmable and composable control of search. While advanced search strategies can always be *programmed* in Prolog, we opt instead for an approach that separates the search strategy used from the actual code, so that different strategies can be used on the same set of clauses. We provide constructs for controlling the search strategies that allow adapting the search dynamically. We also illustrate the usefulness of the proposed approach by applying it in the context of testing (constraint) logic programs, showing how composable search parameters enable more controlled and targeted exploration of program behavior.

Keywords: Prolog · Search Rules · Assertion-based Testing · Property-based Testing · (Constraint) Logic Programming.

1 Introduction

Ever since Kowalski's well-known equation *Algorithm = Logic + Control* [22], the advantages of separating the logic and control components of a program have been well established. In Prolog, the *Logic* component is expressed through Horn clauses, while *Control* is handled by the engine, primarily via the search strategy. Standard systems rely on SLD resolution, typically using a fixed top-down, left-to-right strategy that corresponds to depth-first search. While this choice is efficient in terms of memory, requiring storage only for the active branch,

* Partially funded by MICIU projects CEX2024-001471-M María de Maeztu and TED2021-132464B-I00 PRODIGY, as well as by the Tezos foundation. We would also like to thank the reviewers for their very useful and constructive feedback.

it is incomplete in general (i.e., for infinite search spaces). Moreover, it forces programmers to handle control issues such as cycles or left recursion indirectly, typically by rewriting rules or adding operational details, which can compromise the declarative character of the language.

Alternative search strategies, such as breadth-first, iterative deepening, or random search, offer different trade-offs among efficiency, completeness, and memory consumption. However, mainstream Prolog implementations provide little support for selecting or customizing these strategies. Consequently, programmers have limited influence over how the search space is explored, and control decisions remain implicit in the engine rather than expressible by the programmer.

This paper revisits the role of search strategies in the execution of logic programs and explores how to make the control component *configurable in flexible ways* while still supporting the full language and preserving the separation between *logic* and *control* in the spirit of declarative programming. Writing Prolog code from scratch that implements some search in a state space with a particular strategy is not especially difficult, nor is running Prolog predicates with other search rules by using variations of the standard meta-interpreter. However, our goal is to be able to provide the programmer with a mechanism that allows running the predicates of a standard Prolog program with different search strategies with a high degree of flexibility. Moreover, straightforward implementation approaches of alternative search strategies typically limit the use to a subset of Prolog and thus do not support the full expressiveness of the language. Our second goal is thus to maintain compatibility with modules, built-ins, and other libraries and features to a high degree.

We refine the meaning of a *search strategy* by defining it as the composition of: i) A *search engine*, which determines how the SLD-tree is traversed, e.g., depth-first, breadth-first, iterative deepening, random search; and ii) *control parameters*, which decide if/when to switch strategies, how far or how much to explore (e.g., depth, number of solutions, time limit), etc.

Our interest in alternative search rules comes from different angles. First, when teaching (C)LP and Prolog, the ability to switch between depth-first and fair search rules can help students visualize the true potential of the (C)LP paradigm and Prolog to gain a hands-on understanding of concepts such as *termination*, *decidability*, or *the halting problem* (see, e.g., [16]).

Also, Prolog programmers, for example when implementing Artificial Intelligence (AI) applications, can define and experiment with different search strategies to trade off efficiency against completeness. As mentioned before, the standard Prolog depth-first search can, in some cases, become trapped in an infinite branch and fail to produce a solution, even when solutions exist. As just an example, a strategy that begins with breadth-first search and switches to depth-first search once a certain resource-consumption threshold is reached may yield some, though possibly not all, solutions, yet these can still be useful.

Our final motivation, that we will use as running example throughout the paper, is *test case generation*. Some previous work in this area within LP leverages

assertion preconditions as *test case generators* [23]. Since these preconditions are conjunctions of literals, the corresponding predicates can be used to systematically produce valid inputs. The key innovation lies in executing standard predicates under non-standard search rules, enabling either fully automatic or user-guided generation. By adapting the search strategy, we can control how the input space is explored, improving coverage and avoiding redundant or non-terminating test cases. Purely random generation may be sufficient for some programs, but for others, it can be inefficient or fail to reach deeper regions of the search space. For instance, when predicates involve multiple recursive clauses, random exploration may become trapped in certain branches, spending excessive time generating similar test cases. Taking into account alternative search strategies or combining different ones can provide useful alternatives.

Thus, the multi-configurable search rule framework presented in this paper can be used for various purposes. As already mentioned, we focus here on its application to testing.

The rest of the paper proceeds as follows: Section 2 provides the background on the test framework that we base our work on. Section 3 presents the proposed language of directives for specifying search strategies and explains its interaction with the assertion language of the test framework. It also provides examples that demonstrate how the framework can be extended using our multi-configurable search rules. Section 4 presents a case study on binary trees, including some experimental results. We close by discussing related work in Section 5 and conclusions in Section 6.

2 Some Background on the Testing Framework Used

In this section, we start by providing some background on concepts needed to understand how our multi-configurable search-rule framework is used to perform improved program testing. To this end, we introduce the assertion framework of Ciao Prolog [15, 17, 18], which already has many useful components for our purposes.

2.1 Checking Predicates Against Specifications

We begin by presenting how to determine whether a given predicate satisfies the preconditions and postconditions declared in program assertions in the Ciao model.[4] In the Ciao model, predicates are checked against their specifications through a combination of static and dynamic techniques. *Static checking* of these assertions is performed at compile time by the CiaoPP tool using abstract interpretation [24, 19, 12]. This allows many properties, such as types, modes, determinacy, and non-failure, to be verified automatically before execution. However, because static analysis is undecidable, some properties or parts of assertions

[4] This model was a precursor of *gradual typing*, *hybrid-typing*, and similar approaches [10, 29, 26], sharing many general principles.

may remain unproven. In such cases, these unverified assertions are annotated in the output program with `check` status. The resulting program can then be *instrumented* with run-time checks [30, 31], to ensure safety during execution. This *run-time checking* process proceeds as follows: Given a set of queries Q and a set of assertions A, the run-time checking process executes the program on the queries in Q and determines whether the resulting derivations belong to the error set defined by the assertions. It is not expected that this process can prove an assertion to be *fully checked*, since that would require exploring all possible derivations from all valid queries, often an infinite set. Instead, the goal is to test a representative subset of queries, which, although incomplete, allows detecting many violations of the specification. An enabler here is the fact that properties are written in Prolog, and are thus *runnable*, meaning they can be verified at run-time. For instance, consider the following property:

```
:- prop sorted_int_list/1.
sorted_int_list([]).
sorted_int_list([X]) :- int(X).
sorted_int_list([X,Y|T]) :- int(X), int(Y), X >= Y, sorted_int_list([Y|T]).
```

The query `sorted_int_list(X)` succeeds for `X = []`, `X = [1]`, and `X = [2, 1]`; fails for `X = a` and `X = f(a)`; and instantiates variables for `X = [A,B]` and `X = A`. A predicate `check/1` exists that captures failure or further instantiation (the latter meaning that the argument is not as instantiated as the property requires), and raises an error in any of those cases.[5] This mechanism enables the dynamic verification of a wide range of properties, providing a smooth integration of static and dynamic checking. If the definition of these properties is provided directly in the source language, then such properties are typically already runnable and thus available for run-time checking. However, it is also possible to provide a specialized implementation for run-time checking if desired. For properties that are declared native but are not written in the source language, a run-time test version must be provided.

2.2 Generating Test Cases from Properties in Assertions

An important complement to static and dynamic checking is the ability to *test* a program automatically by generating input data that satisfies its preconditions. This idea builds on earlier work on random testing [14], later adapted to the Ciao assertion model [23, 3]. Given an assertion for a predicate, the objective is to automatically generate goals whose arguments satisfy the assertion's precondition and then execute them to determine whether the corresponding postconditions (and global properties) hold or whether violations can be detected. In this setting, the notion of generating random test values from assertion preconditions arises naturally: since preconditions are typically expressed as conjunctions of property literals, these same property predicates can serve as *generators* of test inputs. The generation process is based on executing the property predicates

[5] More precisely, these are *instantiation* checks. *Compatibility* checks are also supported, but the discussion is beyond the scope of this paper. See the previously cited bibliography on the Ciao assertion model for details.

Table 1. Syntax of the search strategy language.

⟨*sr_decl*⟩	::= `:- search_rule(`⟨*sr_name*⟩, ⟨*pred*⟩`)`
⟨*sr_definition*⟩	::= `:- def_sr(`⟨*sr_name*⟩, [⟨*options*⟩]`)`
⟨*pred*⟩	::= ⟨*pred_name*⟩ | ϵ
⟨*pred_name*⟩	::= Pred/⟨*arity*⟩
⟨*arity*⟩	::= Integer
⟨*options*⟩	::= ⟨*sr_engine_decls*⟩, ⟨*control_parameter*⟩, ⟨*apply_sr*⟩
⟨*sr_engine_decls*⟩	::= `extends(`⟨*sr_name*⟩`)`, ⟨*sr_engine_decl*⟩
⟨*sr_engine_decl*⟩	::= ⟨*sr_engine_pred*⟩ | ⟨*sr_engine_pred*⟩, ⟨*sr_engine_decl*⟩
⟨*sr_engine_pred*⟩	::= `set_sr(`⟨*pred_name*⟩`)` = ⟨*sr_name*⟩ | ϵ
⟨*sr_name*⟩	::= Sr | ⟨*sr_engine*⟩ | [⟨*sr_rules*⟩]
⟨*sr_engine*⟩	::= `df` | `bf` | `rnd` | `id` | `af`
⟨*sr_rules*⟩	::= ⟨*sr_rule*⟩ | ⟨*sr_rule*⟩, ⟨*sr_rules*⟩
⟨*sr_rule*⟩	::= _(⟨*args*⟩, ⟨*sr_name*⟩) :- ⟨*condition*⟩
⟨*args*⟩	::= Var | Var, ⟨*args*⟩
⟨*control_parameter*⟩	::= ⟨*sr_limit*⟩, ⟨*sr_selector*⟩, ⟨*delay*⟩
⟨*sr_limit*⟩	::= `time` = Integer | `depth` = Integer | `steps` = Integer | ϵ
⟨*sr_selector*⟩	::= `first_solution` | `all_solutions` | `num_solutions` = Integer | ϵ
⟨*delay*⟩	::= `delay` = ⟨*pred_name*⟩ | ϵ
⟨*apply_sr*⟩	::= `apply_to_pre =` [⟨*asserts*⟩] | ϵ
⟨*asserts*⟩	::= ⟨*assert*⟩ | ⟨*assert*⟩ ⟨*asserts*⟩
⟨*assert*⟩	::= Id | ⟨*pred_name*⟩ | ϵ

in generation mode using a random search rule in order to produce a set of valid test cases. Once the test inputs are generated, the existing run-time check instrumentation provided is reused to perform the actual verification of assertions during execution. This combination allows a wide range of properties to be tested automatically, including those specific to (Constraint) Logic Programming, such as shape-based (regular) types, variable sharing, and instantiation patterns. By interpreting assertions both as specifications and as generators of input data, the Ciao model provides a smooth connection between specification, verification, and testing.

3 Strategy Specification and Assertion Languages

In this section, we describe the language for defining search strategies. Since program testing is the primary application in this paper, we also present the assertion language and explain how the two languages interact during assertion-based testing.

3.1 The Search Strategy Language

Figure 1 presents the main grammar rules for the search strategy language. Such language enables assigning search strategies to a modular pro-

gram through *search strategy declarations* (⟨*sr_decl*⟩) and search strategy definitions (⟨*sr_definition*⟩). The latter allow the definition of new strategies as compositions of existing ones. Each strategy defines the search engine (⟨*sr_engine_decls*⟩) to be used for a specific predicate (⟨*pred_name*⟩) or for all the predicates of a whole module. The only mandatory element for defining a search strategy is the use of the `extends(Sr)` declaration, which specifies the search rule being extended and defines the *default* search rule. This declaration ensures that the new strategy inherits the complete behavior of search rule `Sr`. The search engine determines the order in which nodes of the SLD-tree are expanded, such as depth-first (`df`), breadth-first (`bf`), random (`rnd`), etc., user-defined (⟨*sr_name*⟩), or conditional composition of different searches. At each node, the control parameters (⟨*control_parameters*⟩) impose quantitative constraints on the exploration, such as `depth`, `time`, or solution selectors (`first_solution`, `all_solutions`, `num_solutions`), and determine the conditions under which goals and clauses are delayed.[6] These parameters compose the set of options (⟨*options*⟩) in a search strategy definition, together with ⟨*apply_sr*⟩, which is used as an interface between the assertion language and the search strategy language, and will be described in detail in Section 3.2. The grammar includes the following constants: "Pred" (any valid predicate name in the underlying language, normally non-empty strings starting with a lower-case letter or enclosed in quotes); "Var" (which corresponds to variable names, normally non-empty strings of characters that start with a capital letter or an underscore); "Integer" (which denotes any valid integer); and "Sr" and "Id" (which can be non-empty strings starting with a lower-case letter).

3.2 Assertion Language Used

So far, we have presented constructs for dynamic search rule selection, with the objective of using them for generating test cases, but without relating them directly to assertions. As mentioned before, we will use Ciao Prolog's assertion language for our purposes. We now briefly describe it. The general specification of a predicate `p/n` consists of declarations that provide partial specifications of its behavior. They have the following syntax:

`:-` [*Status*] `pred` *Head* [`:` *Calls*] [`=>` *Success*] [`+` *Comp*] [`#` *Comm*]`.`

which expresses that a) calls to predicate *Head* that satisfy precondition *Calls* are admissible and b) for such calls, if they succeed, the predicate must satisfy post-condition *Success* and global computational properties *Comp*. *Calls* and *Success* are conjunctions of property literals. *Comm* is a string that contains a textual description or encapsulates an identifier of the assertion. If there are several `pred` assertions, the disjunction of the *Calls* fields defines the admissible calls to the predicate.

The following code fragment provides two `pred` assertions defining two particular ways in which predicate `app/3` is expected to be called:

[6] Any predicate can be delayed, but delaying is particularly useful for labeling.

```
:- check pred app(X,Y,Z) : (list(X),list(Y),var(X)) => list(Z) + det #"[id0]".
:- check pred app(X,Y,Z) : (var(X),var(Y),list(Z)) => (list(X),list(Y)) + multi #"[id1]".

app([],X,X).
app([X|Xs],Ys,[X|Zs]) :-
  app(Xs,Ys,Zs).
```

The first pred assertion states that if `app/3` is called with the first two arguments instantiated to lists and the third a variable, and such call succeeds, then the third argument must be bound to a list. This first assertion also states that when called with such call pattern, predicate `app/3` must be deterministic (`det`, a global computational property). The second `pred` assertion states that `app/3` may also be called with the third argument instantiated to a list and the first two as variables, and that, if such a call succeeds, then the first and second arguments should be bound to lists. It also states that, if called that way, the predicate will produce one or more solutions, but not fail (`multi`, also a global computational property). Each of the two assertions is labeled with an (optional) identifier. The `check` status indicates that these are desired properties that need to be checked, statically or dynamically, but have not been proven true or false yet.

What links the generation properties of assertion preconditions with their corresponding search strategies is the identifier of each assertion, specified through the $\langle apply\_sr \rangle$ option in the search rule declaration. This option defines the list of assertions to which the search rule applies. The list can include specific assertion identifiers or predicates, indicating that the search rule applies to all assertions associated with those predicates. For instance, the declaration on the left:

```
:- def_sr(sr1, [
    apply_to_pre = [id1],
    extends(bf)
]).
```

```
:- def_sr(sr2, [
    apply_to_pre = [app/3],
    extends(bf)
]).
```

states that breadth-first should be used as search rule for generating the properties that appear in the precondition of the assertion with identifier `id1`. In turn, the declaration on the right states that the search rule should be applied to the preconditions of all assertions for predicate `app/3`. If no explicit search rule declaration is provided, the default is that all properties are generated using a random strategy.

3.3 Illustrative examples

We now illustrate the use of the strategy specification language through examples that show how to use it to generalize the test generation mechanism. This extension enables finer control over the generation of test inputs compared to previous work [3] in the Ciao system.

Dynamic Strategy Switching Consider verifying a predicate `sumlist(L, S)` that computes the sum `S` of all elements in list `L`:

```
sumlist([], 0).
sumlist([H|T], Sum) :-
  sumlist(T, Rest),
  Sum is H + Rest.
```

A fundamental property is that the sum should be invariant under permutation, i.e., reordering the elements should not change the total sum. We can express this property formally as:

$$\forall L. \forall L'. \forall S. (sumlist(L, S) \land permutation(L, L')) \rightarrow sumlist(L', S)$$

which can be encoded as follows:

```
:- pred sum_perm(L) : list(int,L) + multi.
sum_perm(L) :- sumlist(L, S), permutation(L, L1), sumlist(L1, S).
```

where `permutation/2` is defined as:

```
permutation([], []).
permutation([X|Xs], [R|Rs]) :-
  select(R, [X|Xs], Y),
  permutation(Y, Rs).

select(X, [X|Xs], Xs).
select(X, [Y|Ys], [Y|Zs]) :-
  select(X, Ys, Zs).
```

During testing of our `sumlist` implementation, test lists `L` are generated, and then their sum `S` is computed, all `L'` permutations are generated for each test list, and it is then verified that `sumlist(L',S)` holds for each permutation.

For list generation, we would like to produce lists incrementally by length: `[]`, `[_]`, `[_,_]`, and so on. Native properties, such as `list/2` or `int/1`, rely on a specific implementation for generation, which requires dynamically switching to that internal code. The internal generators for these native properties can be executed under different search strategies. For instance, we apply a depth-first strategy to construct the list structure, and then fill the elements with randomly generated values.

Since we subsequently apply permutations, it is not worthwhile to generate very large lists, as simpler cases already provide good coverage for testing. We can define this search strategy and name it `sr_list` as follows:

```
:- def_sr(sr_list, [
    extends(df),
    set_sr(int/1) = sr_int
]).
```

```
:- def_sr(sr_int, [
    extends(rnd),
    first_solution
]).
```

which extends the depth-first search rule by adding the declaration `set_sr(int/1) = sr_int`, which in turn sets the generation of numbers to be random.

This customized search strategy gives us the following output:

```
?- list(int, L).
L = [] ;
L = [-3] ;
L = [-51,-37] ;
L = [94,-73,-59] ;
L = [-69,-75,-86,-64] ;
L = [-63,-34,37,-83,-82] ;
...
```

Dependent Strategy Selection Search strategy selection can be made dependent on the input. This enables, for instance, predicates to use two different search strategies depending on whether the input is ground or contains variables.

The following search rule declaration for the predicate `permutation/2` specifies different search strategies depending on the length of the input list:

```
:- def_sr(sr_perm, [
    extends(df),
    set_sr(permutation/2) = [
        (_(L,P,bf)  :- var(P), length(L,N), N =< 7),
        (_(L,P,rnd) :- var(P), length(L,N), N > 7),
        (_(L,P,bf)  :- var(L), ground(P))
    ],
    num_solutions = 500
]).
```

It expresses that for any call `permutation(L,P)`, if the condition `var(P), length(L,N), N =< 7` holds (i.e., `P` is unbound and `L` is a list whose length is less than or equal to `7`), then breadth-first search is used; otherwise, random search is applied. The declaration also states that if `L` is unbound and `P` ground, then breadth-first search must be used.

Finally, the parameter `num_solutions = 500` indicates that at most 500 solutions should be generated. This last parameter is one of many termination criteria supported by our framework, including: *depth-based* termination, which halts exploration upon reaching a specified depth; *solution-based* termination, which stops after discovering a target number of solutions; *step-based* termination, which bounds the number of transitions performed before a valid solution is produced; and *time-based termination*, which enforces a maximum time budget. The resulting search strategy definition for `permutation/2`, including these search rules is shown in Figure 1.

3.4 Some implementation details

As mentioned earlier, our goal is to maintain compatibility with modules, built-ins, and other libraries and features to a high degree. For example, supporting modules means that generators are not limited to local predicates. A predicate defined in one module can freely invoke predicates from other modules, inheriting or overriding their corresponding search strategies as needed.

The implementation of the search strategy language is built around a collection of module-aware meta-interpreters and orchestrated around a single predicate: `call_with_sr(Sr, Goal)`, which executes a given goal `Goal` under a search strategy `Sr`. When a module specifies a default search rule, predicates called from a context where no other `call_with_sr/2` is active are executed implicitly as if called through `call_with_sr/2` (using the default search rule specified for the module or for each of its predicates). That `call_with_sr/2` can also be invoked directly in the program to execute any subgoal under an explicitly specified search rule. To determine which search strategy engine or meta-interpreter needs to be used at each call, the system maintains a global stack that is updated whenever `call_with_sr/2` is invoked. The top of the stack records the current search rule, and when execution of that particular call ends, restores

```
:- search_rule(prop_sr).

:- def_sr(prop_sr, [
    apply_to_pre = [id2],
    extends(df),
    set_sr(list/2) = sr_list,
    set_sr(permutation/2) = [
      (_(L,P,bf)  :- length(L,N), N =< 7),
      (_(L,P,rnd) :- length(L,N), N > 7),
      (_(L,P,bf)  :- var(L), ground(P))
  ],
    num_solutions = 500
]).

:- def_sr(sr_list, [
    extends(df),
    set_sr(int/1) = sr_int
]).

:- def_sr(sr_int, [
    extends(rnd),
    first_solution
]).

:- pred prop_sum_perm(X) : list(int,X) + multi # "[id2]".
prop_sum_perm(L) :- sumlist(L, S), permutation(L, L1), sumlist(L1, S).
```

Fig. 1. Search strategy definition for `permutation/2`.

the previous most recently active `call_with_sr/2` invocation. Thus, the search rule applied to a predicate P is determined by looking up the definitions in the current search rule according to the following decreasing priority order: i) search rule declarations for the predicate; ii) declarations from inherited search rules; iii) the primitive search rule.

Once the search rule is determined, execution proceeds with the particular engine or meta-interpreter. For that, a dual representation is maintained for each predicate: the compiled version, for efficient execution under standard depth-first semantics, and the source-level version for metaprogramming. The dual representation allows access to the rules of the program at run-time through the built-in predicate `clause/2`. The implementation design relies on these two key aspects that preserve Prolog's module system semantics, while providing control over which predicates are affected by a given search rule.

Let us use Figure 2 to illustrate this idea. The module `main` defines the predicate `gen/2`, which uses the predicate `prop/2` defined in module `prop_def` and declares the search rule `sr0`. Within `sr0`, the search rule declares a specific search strategy for predicate `gen_/2` and extends the search rule `srA`. `srA` can refer either to the identifier of another search rule declaration or to a primitive search rule (e.g., breadth-first, depth-first, random, etc.). In the case where `srA` refers to another search rule, the system recursively traces it until a primitive search rule is reached and establishes it as the *general* search rule for `sr0`.

For instance, if the goal `?- gen(X,Y)` is executed, it starts under the search strategy `srA`. Then `prop/2` is called; note that it is inside an explicit `call_with_sr/2`. Therefore, the search rule inside `call_with_sr/2` takes prece-

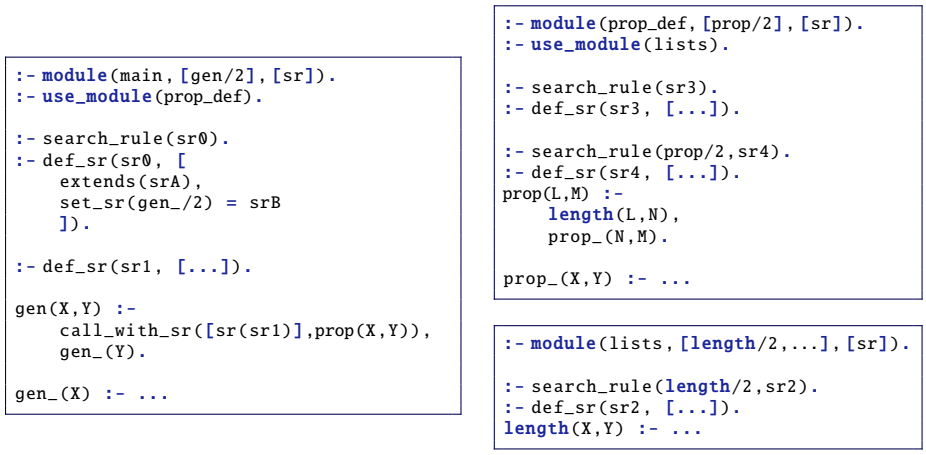

Fig. 2. Example of three modules with different search strategies defined.

dence over the search rule defined in module `prop_def` and the call to `prop/2` is executed using search rule `sr1`. Note that if the query executed is `?- prop(X,Y)`, it will be executed using `sr4`, the module default `sr3` is ignored because predicate level declarations take priority. Continuing the example, `prop/2` calls `length/2` from the `lists` module. The search rule applied is the one defined in the `lists` module, which is `sr2`. `prop/2` also calls the auxiliary predicate `prop_/2`. In this case, it inherits the strategy of `prop/2`, since the most recent active `call_with_sr/2` is that of `prop/2`. Once the execution of `prop/2` ends, `gen/2` calls `gen_/2`. In this case, the current search rule is `srA`. Since `srA` extends another search rule, the system looks up the extends chain to check whether a specific search rule is defined for the predicate `gen_/2`. In this example, it first looks up in the search declaration of `sr0` whether such a rule exists, and it is applied. Thus, `gen_/2` is executed using `srB`. Once the search finishes, the current search rule reverts to `srA`, and the call under the search rule `srA` terminates, completing the execution of goal `gen(X, Y)`.

As can be seen, even some system-defined predicates, such as `length/2` in the above example, can be executed using non-standard search rules, in particular those used as properties in assertions, such as `int/1`. To this end, we have developed alternative *generating* implementations of these built-ins, and these alternative implementations are selected dynamically based on the search strategy.

An important open question is the applicability of our mechanism in the presence of extra-logical constructs. While the approach works for pure programs, including those with constraints, predicates such as `assert/1`, `retract/1`, or other side-effects introduce dependencies on sequentiality that may interact non-trivially with some search rules. It is the programmer's responsibility to ensure that the behavior is as intended if these built-ins are used. Moreover, infinite or unbounded search spaces can be mitigated by imposing explicit limits,

```
:- prop complex_property/2 # "A predicate generator".
complex_property(X,S) :-
    tree(X),
    sorted_tree(X),
    tsum(X,S).

:- regtype tree/1 # "Simple binary tree with integer nodes".
tree(empty).
tree(tree(LC,X,RC)) :-
    X #> 0, X #=< 100,
    tree(LC),
    tree(RC).

% Check if tree T is sorted
sorted_tree(T) :- ...

% tsum(T,N) : Constrains the sum of all node values in tree T to N
tsum(T,N) :- ...

% insert(X,T0,T1) : The result of inserting the node X into the tree T0 is T1
:- pred insert(X,T0,T1) : (nnegint(X), tree(T0), sorted_tree(T0))
                        => (sorted_tree(T1), non_empty_tree(T1))
                        + det # "[id2]".

insert(X,empty,tree(empty,X,empty)).
insert(X,tree(LC,X,RC),tree(LC,X,RC)).
insert(X,tree(LC,Y,RC),tree(LC_p,Y,RC)) :-
    X #=< Y, % <-- There is a bug!
    insert(X,LC,LC_p).
insert(X,tree(LC,Y,RC),tree(LC,Y,RC_p)) :-
    X #> Y,
    insert(X,RC,RC_p).

... % rest of the implementation of module predicates
```

Fig. 3. Binary tree library.

such as bounding the number of solutions, execution time, or search depth, but a more general mechanism for controlling such behavior remains as future work.

4 A Case Study: Generation of Binary Trees

Consider the excerpt in Figure 3 from a library using binary trees. The `tree/1` property describes the shape of trees, which can be an empty tree (a terminal node) or a compound term (a non-terminal node) with a value, and left and right subtrees as arguments. The node values are integers between 1 and 100.

Generating a more complex property. Our intention is to generate values for `complex_property/2`, which is composed of several sub-properties: the tree structure, the ordering constraint (`sorted_tree/1`), and the total sum of node values (`tsum/2`). Such ordering constraint states that every node's left subtree contains only values less than the node's value, and every node's right subtree contains only values greater than the node's value. This ensures that an in-order traversal of the tree yields a sorted sequence.

To study the behavior of different search strategies, we conducted an experiment based on the property `complex_property(T,S)`, varying S. Each generation

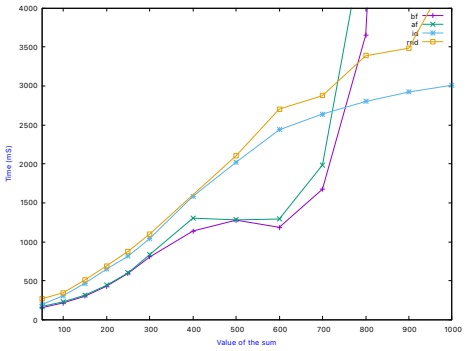

Fig. 4. Comparing search rules executing `complex_property(T,S)`.

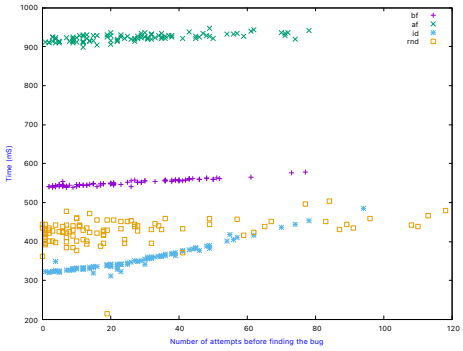

Fig. 5. Execution time and number of attempts until the bug is found.

was executed with a limit of 100 solutions and a maximum time of 180 seconds per run. Figure 4 shows the execution time for several strategies: breadth-first (`bf`), breadth-first AND-fair (`af`), iterative deepening (`id`), and random (`rnd`), which is the default one. For small sums, $S \leq 500$, breadth-first yields the best performance, as it explores shallow trees first and quickly finds valid configurations. However, as S increases, the number of possible combinations grows exponentially, and breadth-first becomes less efficient. Iterative deepening (`id`) incurs moderate overhead due to repeated traversals but remains complete and scales reasonably well. In contrast, the random strategy (`rnd`) exhibits higher variance and slower convergence, particularly for larger values of S, as it lacks structural guidance and may repeatedly explore redundant branches. Finally, depth-first (`df`) proved the least effective. Because the number of required solutions was capped at 100, it frequently descended down deep branches and spent substantial time exploring a single path before backtracking, often resulting in timeouts.

The experimental results suggest that an effective search strategy for generating values for `complex_property/2` is the following:

```
:- def_sr(complex_prop_sr, [
     extends(df),
     set_sr(complex_property/2) = [
          (_(X,Y,bf) :- Y =< 500),
          (_(X,Y,id) :- Y > 500)
     ],
     delay = labeling/2,
     num_solutions = 100,
     time = 180
  ]).
```

This definition specifies that the `complex_property/2` predicate dynamically adapts its search strategy based on the metric value `S`. As suggested by the plot, if $S \leq 500$, breadth-first (`bf`) search is used; if $S > 500$, iterative deepening (`id`) is used. The remaining predicates that compose the generator are executed using depth-first search for efficiency. Additionally, labeling (via `labeling/2`) is delayed until the end of the computation, adopting a *constrain-and-generate* approach,

common in constraint logic programming: constraints are accumulated during structure generation, and numeric variables are instantiated later.

Assertion checking. We now turn to testing the predicate `insert/3`, also implemented in Figure 3, which returns the tree resulting from adding a node to an existing tree. We have intentionally introduced a bug in the third clause: instead of using the correct constraint `#<`, the clause uses `#=<`. As a result, when the tree contains a node with the same number of input nodes, two clauses succeed, i.e., the clauses are not mutually exclusive, which violates the intended ordering property of the tree and the computational property `det` (deterministic). In this case, we evaluate which search rule for generating trees requires less time and fewer attempts to detect the bug. For `nnegint/1`, which generates non-negative integers, random search is used.

Figure 5 shows a scatter plot comparing the search rules in terms of execution time and the number of tests required to find the bug. We conducted the experiments using the same search rules as before, performing 100 runs for each. We observe that, regarding execution time, the iterative-deepening strategy is the fastest in finding bugs, typically requiring a relatively small number of attempts (mostly between 0 and 80). Breadth-first and breadth-first AND-fair strategies are more consistent in the number of attempts needed: most `bf` runs find the bug in fewer than 50 attempts, while `af` usually requires fewer than 65 attempts. However, both `bf` and `af` tend to take more time overall, especially `af`. The random strategy exhibits the highest variability, with some runs finding the bug quickly, while others require substantially more attempts. This strategy performs reasonably well in this example. However, it does not guarantee that the bug will be found within a limited number of attempts. If another search rule can provide such a guarantee, such as iterative deepening in this example, it is preferable to use it, as it offers more predictable and controllable search behavior.

Improving fairness in random search. Let us now illustrate an extension of our search algorithm, useful to improve fairness when using random search strategies. As an example, consider the problem of generating random linear constraints:

```
constraint(=(L1, L2))  :- lin_expr(L1, L2).
constraint(=<(L1, L2)) :- lin_expr(L1, L2).
constraint(>=(L1, L2)) :- lin_expr(L1, L2).
constraint(<(L1, L2))  :- lin_expr(L1, L2).
constraint(>(L1, L2))  :- lin_expr(L1, L2).

lin_expr(L1, L2) :- lin_expr(L1), lin_expr(L2).

lin_expr(C)         :- coefficient(C).
lin_expr(V)         :- var(V).
lin_expr(+(E))      :- lin_expr(E).
lin_expr(-(E))      :- lin_expr(E).
lin_expr(+(E1, E2)) :- lin_expr(E1), lin_expr(E2).
lin_expr(-(E1, E2)) :- lin_expr(E1), lin_expr(E2).
lin_expr(*(C, E))   :- coefficient(C), lin_expr(E).
lin_expr(*(E, C))   :- coefficient(C), lin_expr(E).

coefficient(C) :- C in 0..500, labeling([rnd],C).
% $VAR used as ground representation of variables
var(A)         :- uppercase_char(Arg), A = '$VAR'(Arg).
```

The previous rules define arithmetic constraints recursively, using relational operators (=, =<, >, etc.) and linear expressions that combine coefficients, variables and subexpressions through addition, subtraction, and multiplication. Each coefficient is an integer variable constrained to lay within a given range (here, 0 to 500). The final call to `labeling/2` enumerates concrete integer values for these coefficients.

These constraints are employed to synthesize programs that can serve as benchmarks for evaluating the polyhedra abstract domain [9]:

```
p(C,F,G) :-
    5*A-67+F<100,
    A*(142*4)=G,
    3< -(47*M*78)*291,
    88* -(311*F)>=C,
    X*215=<0,
    - -H>(90+(C*(178*J)))*67.
```

Naive random exploration in this example may lead to non-termination when execution is trapped in the creation of more recursive clauses than base cases. To make search generation fair, our work proposes a method based on identifying recursive predicates within a given program, using the implementation of Tarjan's algorithm [32, 33] used in CiaoPP to classify recursive and non-recursive predicates. Using this information, we alternate between base and recursive cases during generation, ensuring that the search process avoids infinite recursion. As future work, we plan to explore this idea further by leveraging static analysis to further enhance search strategies.

5 Related work

Property-based testing [5] frameworks have become a widely used approach for validating program properties by generating and executing test cases. Originally developed for `Haskell` and functional programming [5], these frameworks allow developers to automatically generate random input data from some given properties. It has since been adapted to additional languages, including `Erlang` [25], `Curry` with `EasyCheck` [4], and `Prolog` [1, 23, 9, 7]. In the context of Constraint Logic Programming (CLP) and Constrained Horn Clauses (CHC), using predicates as property-based generators is particularly natural, since calling a predicate with free variables can automatically instantiate them. `PrologCheck` [1] provides a language for specifying properties and custom generators for Prolog programs. For complex data structures, such as sorted lists or AVL trees, naive random generation is often insufficient, and developers must provide specialized generators. The tool supports writing such generators, including properties that consider the modes of the arguments. As mentioned before, Ciao Prolog's `LPcheck` [23, 9] introduced the concept of *assertion-based testing*, based on using the properties in its assertion language directly as generators. `ProSyT` [7] generates data structures for `Erlang` programs by combining symbolic data structure generation, constraint solving, and randomized variable instantiation via coroutining. Thus, it reduces the programmer's effort in writing custom generators.

The integration of search strategies into declarative languages has long also been explored. CLP systems [21, 27] and most Prolog systems include Constraint Logic Programming (CLP) libraries [2, 6, 34, 13, 20, 15] that provide predefined search strategies. These search rules are often embedded in labeling predicates for solving different classes of problems. In [8], search control was investigated by proposing a flexible framework for `CHR`$\vee$ (Constraint Handling Rules with disjunction), extended with rule and search branch priorities. Schrijvers proposed `Tor` [28] as a mechanism for supporting the execution of predicates using alternative search rules. `Tor` represents a device that could certainly be useful as an implementation technique for our approach, although we currently use other mechanisms available in the `Ciao` Prolog system instead, which we have used as the basis for our experiments. In this work, however, we concentrate instead on providing a higher-level way for users to specify search strategies.

The idea of exploiting search strategies within property-based testing unifies these two lines of research. By viewing the testing process itself as a search problem, alternative search rules can guide how candidate test cases are generated and explored. Unlike earlier work where search and test generation were orthogonal, our approach treats the search strategy as a component of the testing process, providing a framework for exploring properties under different execution rules.

6 Conclusions

In this work, we have presented a framework for controlling the execution of Prolog programs through flexible search strategies. By separating the logic of predicates from their exploration order, our approach allows the same standard Prolog predicate to be executed under different search rules without modifying the program itself. This capability mitigates several common limitations of random property-based testing, such as infinite exploration, non-termination, or inefficient traversal of the search space. We introduced a search strategy specification language that combines search engines with control parameters such as clause selection, delayed evaluation, and termination criteria. The framework fully supports Prolog's expressive features, including constraints, DCGs, and modules, ensuring that predicates can call other predicates across modules while respecting or overriding their search strategies. Through several examples, we demonstrated how the proposed search strategy language improves exploration efficiency. Future work includes refining metric-based strategy selection, extending search strategies with static analysis guidance, and exploring richer scheduling policies, as well as, in the testing context, studying combinations with other orthogonal techniques such as those based on program coverage or concolic testing [11]. Overall, we argue that our framework provides a flexible, declarative, and practical approach to controlling the execution of logic programs, bridging the gap between declarative specifications and operational behavior.

References

1. Amaral, C., Florido, M., Costa, V.S.: PrologCheck - Property-Based Testing in Prolog. In: Functional and Logic Programming - 12th Int'l. Symp., FLOPS. LNCS, vol. 8475, pp. 1–17. Springer (2014)
2. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs: Including a Special Trach on Declarative Programming Languages in Education. pp. 191–206. PLILP '97, Springer-Verlag, London, UK, UK (1997), http://dl.acm.org/citation.cfm?id=646452.692956
3. Casso, I., Morales, J.F., Lopez-Garcia, P., Hermenegildo, M.: An Integrated Approach to Assertion-Based Random Testing in Prolog. In: Gabbrielli, M. (ed.) Post-Proceedings of the 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'19). LNCS, vol. 12042, pp. 159–176. Springer-Verlag (April 2020). https://doi.org/10.1007/978-3-030-45260-5_10
4. Christiansen, J., Fischer, S.: EasyCheck - Test Data for Free. In: Functional and Logic Programming, 9th Int'l. Symp., FLOPS. pp. 322–336 (April 2008)
5. Claessen, K., Hughes, J.: QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In: Fifth ACM SIGPLAN Int'l. Conf. on Functional Programming. pp. 268–279. ICFP'00, ACM (2000)
6. D. Diaz, S.A., Codognet, P.: On the implementation of GNU Prolog. Theory and Practice of Logic Programming **12**(1–2), 253–282 (January 2012)
7. De Angelis, E., Fioravanti, F., Palacios, A., Pettorossi, A., Proietti, M.: Property-based test case generators for free. In: International Conference on Tests and Proofs. pp. 186–206. Springer (2019)
8. De Koninck, L., Schrijvers, T., Demoen, B.: A Flexible Search Framework for CHR, pp. 16–47. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
9. Ferreiro, D., Casso, I., Lopez-Garcia, P., Morales, J.F., Hermenegildo, M.V.: Checkification: A Practical Approach for Testing Static Analysis Truths. Theory and Practice of Logic Programming (May 2025), https://arxiv.org/abs/2501.12093
10. Flanagan, C.: Hybrid Type Checking. In: 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006. pp. 245–256 (January 2006)
11. Fortz, S., Mesnard, F., Payet, É., Perrouin, G., Vanhoof, W., Vidal, G.: An SMT-Based Concolic Testing Tool for Logic Programs. In: Functional and Logic Programming - 15th International Symposium, FLOPS. Lecture Notes in Computer Science, vol. 12073, pp. 215–219. Springer (September 2020). https://doi.org/10.1007/978-3-030-59025-3_13
12. Garcia-Contreras, I., Morales, J., Hermenegildo, M.: Incremental Analysis of Logic Programs with Assertions and Open Predicates. In: Proceedings of the 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'19). LNCS, vol. 12042, pp. 36–56. Springer (2020). https://doi.org/10.1007/978-3-030-45260-5_3
13. García de la Banda, M.J., Jeffery, D., Marriott, K., Nethercote, N., Stuckey, P.J., Holzbaur, C.: Building constraint solvers with hal. In: ICLP'04. pp. 90–104 (2001)
14. Hamlet, D.: Random Testing. In: Marciniak, J. (ed.) Encyclopedia of Software Engineering, p. 970–978. Wiley (1994)
15. Hermenegildo, M.V., Bueno, F., Carro, M., Lopez-Garcia, P., Mera, E., Morales, J., Puebla, G.: An Overview of Ciao and its Design Philosophy.

Theory and Practice of Logic Programming **12**(1–2), 219–252 (January 2012). https://doi.org/10.1017/S1471068411000457, https://arxiv.org/abs/1102.5497
16. Hermenegildo, M.V., Morales, J.F., Lopez-Garcia, P.: Teaching Pure LP with Prolog and a Fair Search Rule. In: Proceedings of the 40th ICLP Workshops. vol. 3799. CEUR-WS.org (October 2024), https://ceur-ws.org/Vol-3799/paper2PEG2.0.pdf
17. Hermenegildo, M., Puebla, G., Bueno, F.: Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In: Apt, K.R., Marek, V., Truszczynski, M., Warren, D.S. (eds.) The Logic Programming Paradigm: a 25–Year Perspective, pp. 161–192. Springer-Verlag (July 1999)
18. Hermenegildo, M., Puebla, G., Bueno, F., Lopez-Garcia, P.: Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In: 10th International Static Analysis Symposium (SAS'03). pp. 127–152. No. 2694 in LNCS, Springer-Verlag (June 2003)
19. Hermenegildo, M., Puebla, G., Marriott, K., Stuckey, P.: Incremental Analysis of Constraint Logic Programs. ACM Transactions on Programming Languages and Systems **22**(2), 187–223 (March 2000)
20. Holzbaur, C.: OFAI CLP(Q,R) Manual, Edition 1.3.3. Tech. Rep. TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna (1995)
21. Jaffar, J., Michaylov, S.: Methodology and Implementation of a CLP System. In: Fourth International Conference on Logic Programming. pp. 196–219. University of Melbourne, MIT Press (1987)
22. Kowalski, R.: Algorithm = logic + control. Communications of the ACM **22**(7), 424–436 (1979)
23. Mera, E., Lopez-Garcia, P., Hermenegildo, M.: Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In: 25th Int'l. Conference on Logic Programming (ICLP'09). LNCS, vol. 5649, pp. 281–295. Springer-Verlag (July 2009)
24. Muthukumar, K., Hermenegildo, M.: Compile-time Derivation of Variable Dependency Using Abstract Interpretation. Journal of Logic Programming **13**(2/3), 315–347 (July 1992)
25. Papadakis, M., Sagonas, K.: A PropEr Integration of Types and Function Specifications with Property-Based Testing. In: 10th ACM SIGPLAN workshop on Erlang. pp. 39–50 (September 2011)
26. Rastogi, A., Swamy, N., Fournet, C., Bierman, G., Vekris, P.: Safe & Efficient Gradual Typing for TypeScript. In: 42nd POPL. pp. 167–180. ACM (January 2015)
27. Schimpf, J., Shen, K.: ECLiPSe – from LP to CLP. Theory and Practice of Logic Programming **12**(1-2), 127–156 (Jan 2012). https://doi.org/10.1017/S1471068411000469, http://dx.doi.org/10.1017/S1471068411000469
28. Schrijvers, T., Demoen, B., Triska, M., Desouter, B.: Tor: Modular search with hookable disjunction. Sci. Comput. Program. **84**, 101–120 (2014)
29. Siek, J.G., Taha, W.: Gradual Typing for Functional Languages. In: Scheme and Functional Programming Workshop. pp. 81–92. University of Chicago Department of Computer Science (2006)
30. Stulova, N., Morales, J.F., Hermenegildo, M.: Practical Run-time Checking via Unobtrusive Property Caching. Theory and Practice of Logic Programming, 31st Int'l. Conference on Logic Programming (ICLP'15) Special Issue **15**(04-05), 726–741 (September 2015). https://doi.org/10.1017/S1471068415000344, https://arxiv.org/abs/1507.05986

31. Stulova, N., Morales, J.F., Hermenegildo, M.: Reducing the Overhead of Assertion Run-time Checks via Static Analysis. In: 18th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'16). pp. 90–103. ACM Press (September 2016)
32. Tarjan, R.: Depth-First Search and Linear Graph Algorithms. SIAM J. Comput. **1**, 140–160 (1972)
33. Tarjan, R.E.: Fast algorithms for solving path problems. J. ACM **28**(3), 594–614 (July 1981). https://doi.org/10.1145/322261.322273, https://doi.org/10.1145/322261.322273
34. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming **12**(1-2), 67–96 (2012). https://doi.org/10.1017/S1471068411000494