

# An Integrated Approach to Assertion-Based Random Testing in Prolog<sup>\*</sup>

Ignacio Casso<sup>1</sup>, José F. Morales<sup>1</sup>,  
P. López-García<sup>1,3</sup>, and Manuel V. Hermenegildo<sup>1,2</sup>

<sup>1</sup> IMDEA Software Institute, Madrid, Spain

<sup>2</sup> ETSI Informática, Universidad Politécnica de Madrid (UPM), Madrid, Spain

<sup>3</sup> Spanish Council for Scientific Research (CSIC), Spain

{ignacio.casso,josef.morales,pedro.lopez,manuel.hermenegildo}@imdea.org

**Abstract.** We present an approach for assertion-based random testing of Prolog programs that is integrated within an overall assertion-based development model. Our starting point is the *Ciao* assertion model, a framework that unifies unit testing and run-time verification, as well as static verification and static debugging, using a common assertion language. Properties which cannot be verified statically are checked dynamically. In this context, the idea of generating random test values from assertion preconditions emerges naturally since these preconditions are conjunctions of literals, and the corresponding predicates can in principle be used as generators. Our tool generates valid inputs from the properties that appear in the assertions shared with other parts of the model, and the run time-check instrumentation of the *Ciao* framework is used to perform a wide variety of checks. This integration also facilitates the combination with static analysis. The generation process is based on running standard predicates under non-standard (random) search rules. Generation can be fully automatic but can also be guided or defined specifically by the user. We propose methods for supporting (C)LP-specific properties, including combinations of shape-based (regular) types and variable sharing and instantiation, and we also provide some ideas for shrinking for these properties. We also provide a case study applying the tool to the verification and checking of the implementations of some of *Ciao*'s abstract domains.

## 1 Introduction and motivation

Code validation is a vital task in any software development cycle. Traditionally, two of the main approaches used to this end are *verification* and *testing*. The former uses formal methods to prove automatically or interactively some specification of the code, while the latter mainly consists in executing the code for concrete inputs or test cases and checking that the program input-output relations (and behaviour, in general) are the expected ones.

*Ciao* [11] introduced a novel development workflow [12, 13, 21] that integrates the two approaches above. In this model, program assertions are fully integrated

---

<sup>\*</sup> Research partially funded by MINECO TIN2015-67522-C3-1-R *TRACES* project, and the Madrid P2018/TCS-4339 *BLOQUES-CM* program. We are also grateful to the anonymous reviewers for their useful comments.

in the language, and serve both as specifications for static analysis and as run-time check generators, unifying run-time verification and unit testing with static verification and static debugging. This model represents an alternative approach for writing safe programs without relying on full static typing, which is specially useful for dynamic languages like Prolog, and can be considered an antecedent of the popular *gradual*- and *hybrid-typing* approaches [5, 24, 22].

**The Ciao model:** Assertions in the Ciao model can be seen as a shorthand for defining instrumentation to be added to programs, in order to check dynamically preconditions and postconditions, including conditional postconditions, properties at arbitrary program points, and certain computational (non-functional) properties. The run-time semantics implemented by the translation of the assertion language ensures that execution paths that violate the assertions are captured during execution, thus detecting errors. Optionally, (abstract interpretation-based [4]) compile-time analysis is used to detect assertion violations, or to prove (parts of) assertions true, verifying the program or reducing run-time checking overhead.

As an example, consider the following Ciao code (with the standard definition of quick-sort):

```

1 :- pred qs(Xs, Ys) : (list(Xs), var(Ys)) => (list(Ys), sorted(Ys)) + not_fails.
2
3 :- prop list/1.
4 list([]).
5 list(_|_T) :- list(T).
6
7 :- prop sorted/1.
8 ...

```

The assertion has a *calls* field (the conjunction after ':'), a *success* field (the conjunction after '=>'), and a computational properties field (after '+'), where all these fields are optional. It states that a valid calling mode for `qs/2` is to invoke it with its first argument instantiated to a list, and that it will then return a list in `Ys`, that this list will be sorted, and that the predicate will not fail. Properties such as `list/1` or `sorted/1` are normal predicates, but which meet certain properties (e.g., termination) [13] and are marked as such via `prop/1` declarations. Other properties like `var/1` or `not_fails` are builtins.

Compile-time analysis with a *types/shapes* domain can easily detect that, if the predicate is called as stated in the assertion, the `list(Ys)` check on success will always succeed, and that the predicate itself will also succeed. If this predicate appears within a larger program, analysis can also typically infer whether or not `qs/2` is called with a list and a free variable. However, perhaps, e.g., `sorted(Ys)` cannot be checked statically (this is in fact often possible, but let us assume that, e.g., a suitable abstract domain is not at hand). The assertion would then be simplified to:

```

:- pred qs(Xs, Ys) => sorted(Ys).

```

And then `sorted(Ys)` will be called at run-time within the assertion checking harness, right after calls to `qs/2`. This harness ensures that the variable bindings (or constraints) and the whole checking process are kept isolated from the normal execution of the program (this can be seen conceptually as including a Prolog `copy_term`, or calling within a double negation, `\+\+`, executing in a separate process, etc.).

**Testing vs. run-time checking:** The checking of `sorted/1` in the example above will occur in principle during execution of the program, i.e., in *deployment*. However, in many cases it is not desirable to wait until then to detect errors. This is the case for example if errors can be catastrophic or perhaps if there is interest in testing, perhaps for debugging purposes, more general properties that have not been formally proved and whose main statements are not directly part of the program (and thus, will never be executed), such as, e.g.:

```
1 :- pred revrev(X) : list(X) + not_fails.
2 revrev(X) :- reverse(X,Y),reverse(Y,X).
```

This implies performing a *testing* process prior to deployment. The **Ciao** model includes a mechanism, integrated with the assertion language, that allows defining *test assertions*, which will run (parts of) the program for a given input and check the corresponding output, as well as *driving* the run-time checks independently of concrete application data [16]. For example, if the following (unit) tests are added to `qs/2`:

```
1 :- test qs(Xs,Ys) : (Xs = []) => (Ys = []).
2 :- test qs(Xs,Ys) : (Xs = [3,2,4,1]) => (Ys = [1,2,3,4]).
```

`qs/2` will be *run* with, e.g., `[3,2,4,1]` as input in `Xs`, and the output generated in `Ys` will be checked to be instantiated to `[1,2,3,4]`. This is done by extracting the *test drivers* [16]:

```
1 :- texec qs([],_).
2 :- texec qs([3,2,4,1],_).
```

and the rest of the work (checking the assertion fields) is done by the standard run-time checking machinery. In our case, this includes checking at run-time the simplified assertion “`:- pred qs(Xs,Ys) => sorted(Ys).`”, so that the output in `Ys` will be checked by calling the implementation of `sorted/1`.

**Towards automatic generation:** Hand-written test cases such as those above are quite useful in practice, but they are also tedious to write and even when they are present they may not cover some interesting cases. An aspect that is specific to (Constraint-)Logic Programming (CLP) and is quite relevant in this context is that predicates in general (and properties in particular) can be used as both checkers and generators. For example, calling `list(X)` from the `revrev/1` example above with `X` uninstantiated generates lazily, through backtracking, an infinite set of lists, `Xs = []`; `Xs = [_]`; `Xs = [_,_,_]` ..., which can be used to catch cases in which an error in the coding of `reverse/2` makes `revrev/1` fail. This leads naturally to the idea of generating systematically and automatically test cases by running in generation mode (i.e., “backwards”) the properties in the calls fields of assertions.

While this idea of using properties as test case generators has always been present in the descriptions of the **Ciao** model [12, 21], it has not really been exploited significantly to date. Our purpose in this paper is to close this gap. We report on the development of **LPtest**, an implementation of random testing [8] with a more natural connection with Prolog semantics, as well as with the **Ciao** framework. Due to this connection and the use of assertions, this *assertion-based testing* allows supporting complex properties like combinations of shape-based (regular) types, variable sharing, and instantiation, and also non-functional properties.

Our contributions can be summarized as follows:

- We have developed an approach and a tool for assertion-based random test generation for Prolog and related languages. It has a number of characteristics in common with property-based testing from functional languages, as exemplified by **QuickCheck** [3], but provides the assertions and properties required in order to cover (C)LP features such as logical variables and non-ground data structures or non-determinism, with related properties such as modes, variables sharing, non-failure, determinacy and number of solutions, etc. In this, **LPtest** is most similar to **PrologTest** [1], but we argue that our framework is more general and we support richer properties.
- Our approach offers a number of advantages that stem directly from framing it within the **Ciao** model. This includes the integration with compile-time checking (static analysis) and the combination with the run-time checking framework, etc. using a single assertion language. This for example greatly simplifies error reporting and diagnosis, which can all be inherited from these parts of the framework. It can also be combined with other test-case generation schemes. To the extent of our knowledge, this has only been attempted partially by **PropEr** [20]. Also, since **Erlang** is in many ways closer to a functional language, **PropEr** does not support Prolog-relevant properties and it is not integrated with static analysis. In comparison to **PrologTest**, we provide combination with static analysis, through an integrated assertion language, whereas the assertions of **PrologTest** are specific to the tool, and we also support a larger set of properties.
- In our approach the automatic generation of inputs is performed by running in generation mode the properties (predicates) in those preconditions, taking advantage of the specialized SLD *search rules* of the language (e.g., breadth first, iterative deepening, and, in particular, random search) or implementations specialized for such generation. In particular, we perform automatic generation of instances of Prolog regular types, instantiation modes, sharing relations among variables and grounding, arithmetic constraints, etc. To the extent of our knowledge all previous tools only supported generation for types, while we also consider the latter.
- We have enhanced assertion and property-based test generation by combining it with static analysis and abstract domains. To the extent of our knowledge previous work had at most discarded properties that could be proved statically (which in **LPtest** comes free from the overall setting, as mentioned before), but not used static analysis information to guide or improve the testing process.
- We have implemented automatic shrinking for our tool, and in particular we have developed an automatic shrinking algorithm for Prolog regular types.

The rest of this paper is organized as follows. In Sec. 2 we overview our approach to assertion-based testing in the context of Prolog and **Ciao**. In Sec. 3 we introduce our test input generation schema. In Sec. 4, we show how assertion-based testing can be combined with and enhanced by static analysis. Sec. 5 is dedicated to shrinking of test cases in **LPtest**. We show some preliminary results of a case of study in which our tool is applied to prove the correctness of **CiaoPP** (the **Ciao** “preprocessor”) static analysis domain operations in Sec. 6. Finally, we review the related work in Sec. 7 and provide our conclusions in Sec. 8.

## 2 Using LPtest within the Ciao model

As mentioned before, the goal of `LPtest` is to integrate random testing of assertions within `Ciao`'s assertion-based verification and debugging framework. Given an assertion for a predicate, we want to generate goals for that predicate satisfying the assertion precondition (i.e., valid call patterns for the predicate) and execute them to check that the assertion holds for those cases or find errors. As mentioned briefly in the introduction, the current framework already provides most of the pieces needed for this task: the run-time checking framework allows us to check at runtime that the assertions for a predicate are not violated, and the unit-test framework allows us to specify and run the concrete goals to check those assertions. We only need to be able to generate terms satisfying assertion preconditions and to integrate everything. Generation of test cases is discussed in Sec. 3, and the following example shows how everything is integrated step by step.

Consider again a similar assertion for the `qs/2` predicate, and assume that the program has a bug and *fails* for lists with repeated elements:

```
1 :- module(qs,[qs/2],[assertions, nativeprops]).
2 ...
3 :- pred qs(Xs,Ys) : (list(Xs,int), var(Ys))
4                   => (list(Ys,int), sorted(Ys)) + not_fails.
5 ...
6 partition([],_,[],[]).
7 partition([X|Xs],Pv,[X|L],R) :- X < Pv, !, partition(Xs,Pv,L,R). % should be =<
8 partition([X|Xs],Pv,L,[X|R]) :- X > Pv, partition(Xs,Pv,L,R).
```

When we invoke `LPtest` to test the assertions of the `qs` module, first it will use `CiaoPP` to analyze the module and try to verify the assertions statically [13]. As a result each assertion may be proved true or false, and then it can be ignored in the testing phase, or it is left for run-time checking and testing, possibly simplified. `CiaoPP` generates a new source file which includes the original assertions marked with *status* `checked`, `false`, or, for the ones that remain for run-time checking, `check`. `LPtest` starts by reporting a simple adaptation of `CiaoPP`'s output. E.g., for our example, `LPtest` will output:

```
1 Testing assertion:
2 :- pred qs(Xs,Ys) : (list(Xs,int), var(Ys))
3                   => (list(Ys,int), sorted(Ys)) + not_fails.
4
5 Assertion was partially verified statically:
6 :- checked pred qs(Xs,Ys) (list(Xs,int), var(Ys)) => list(Ys,int).
7 Left to check::
8 :- check pred qs(Xs,Ys) => sorted(Ys) + not_fails.
```

`LPtest` will then try test dynamically the remaining assertion. For that, it will first collect the `Ciao` properties that the test case must fulfill (i.e., those in the precondition of the assertion, which is taken from the *original* assertion, which is also output by `CiaoPP`), and generate a number (100 by default) of *test case drivers* (`texec`'s) satisfying those properties. Those test cases will be pipelined to the *unit-test framework*, which, relying on the standard run-time checking instrumentation, will manage their execution, capture any error reported during run-time checking, and return them to `LPtest`, which will output:

```
1 Assertion
2 :- checked pred qs(Xs,Ys) => sorted(Ys) + not_fails.
3 proven false for test case:
4 :- texec qs([5,9,-3,8,9,-6,2],_).
```

```

5 because:
6   call to qs(Xs,Ys) fails for
7   Xs = [9,-3,8,9,-6,2]

```

Finally, `LPtest` will try to shrink the test cases, enumerating test cases that are progressively smaller and repeating the steps above in a loop to find the smallest test case which violates the assertion. `LPtest` will output:

```

1 Test case shrunk to:
2   :- texec qs([0,0],_).

```

The testing algorithm for a module can thus be summarized as follows:

1. (Call `CiaoPP`) Use *static analysis* to check the assertions. Remove proved assertions, simplify partially proved assertions.
2. (`LPtest`) For each assertion, *generate  $N$  test cases* from the properties in the precondition, following the guidelines in Sec. 3. For each test case, go to 3. Then go to 4.
3. (`RTchecks`) Use the unit-test framework to execute the test case and capture any run-time checking error (i.e., assertion violation).
4. (`LPtest`) Collect all failed test cases from `RTchecks`. For each of them, go to 5 to shrink them, and then report them (using `RTchecks`).
5. (`LPtest`) Generate a simpler test case not generated yet.
  - If not possible, finalize and return current test case as shrunk test case. If possible, go to 3 to run the test.
    - \* If the new test case fails, go to 5 with the new test case.
    - \* If it succeeds, repeat this step.

The use of the `Ciao` run-time checking framework in this (pseudo-)algorithm, together with the rich set of native properties in `Ciao`, allows us to specify and check a wide range of properties for our programs. We provide a few examples of the expressive power of the approach:

*(Conditional) Postconditions.* We can write postconditions using the *success* (`=>`) field of the assertions. Those postconditions can range from user-defined predicates to properties native to `CiaoPP`, for which there are built-in checkers in the run-time checking framework. These properties include types, which can be partially instantiated, i.e., contain variables, and additional features particular to logic programming such as modes and sharing between variables. As an example, one can test with `LPtest` the following assertions, where `covered(X,Y)` means that all variables occurring in `X` also occur in `Y`:

```

1 :- pred rev(Xs,Ys) : list(Xs) => list(Ys).
2 :- pred sort(Xs,Ys) : list(Xs,int) => (list(Ys,int), sorted(Ys)).
3 :- pred numbervars(Term,N,M) => ground(Term).
4 :- pred varset(Term,Xs) => (list(Xs,var), covered(Term,Xs)).

```

For this kind of properties, `LPtest` tries to ensure that at least some of the test cases do not succeed trivially (by the predicate just failing), and warns otherwise.

*Computational Properties.* `LPtest` can also be used to check properties regarding the computation of a predicate. These properties are all native and talk about features that range from determinism and multiplicity of solutions to resource usage (cost). They can be checked with `LPtest`, as long as the run-time checking

framework supports it (e.g., some properties, like termination, are not decidable). Examples of this would be:

```
1 :- pred rev(X,Y) : list(X) + (not_fails, is_det, no_choicepoints).
2 :- pred append(X,Y,Z) : list(X) => steps_o(length(X)).
```

*Rich generation.* The properties supported for generation include not only types, but also modes and sharing between variables, and arithmetic constraints, as well as a restricted set of user-defined properties. As an example, **LPtest** can test the following assertion:

```
1 :- prop sorted_int_list(X).
2
3 sorted_int_list([]).
4 sorted_int_list([N]) :- int(N).
5 sorted_int_list([N,M|Ms]) :- N =< M, sorted_int_list([M|Ms]).
6
7 :- pred insert_ord(X,Xs,XsWithX)
8       : (int(X), sorted_int_list(X))
9       => sorted_int_list(XsWithX).
```

### 3 Test Case Generation

The previous section illustrated specially the parts that **LPtest** inherits from the **Ciao** framework, but a crucial step was skipped: the generation of test cases from the *calls* field of the assertions, i.e., the generation of Prolog terms satisfying a conjunction of **Ciao** properties. This was obviously one of the main challenges we faced when designing and implementing **LPtest**. In order for the tool to be integrated naturally within the **Ciao** verification and debugging framework, this generation had to be as automatic as possible. However, full automation is not always possible in the presence of arbitrary properties potentially using the whole Prolog language (e.g., cuts, dynamic predicates, etc.). The solution we arrived at is to support fully automatic and efficient generation for reasonable subsets of the Prolog language, and provide means for the user to guide the generation in more complex scenarios.

**Pure Prolog.** The simplest and essential subset of Prolog is pure Prolog. In pure Prolog every predicate, and, in particular, every **Ciao** property, is itself a generator: if it succeeds with some terms as arguments, those terms will be (possibly instances of) answers to the predicate when called with free variables as arguments. The problem is that the classic depth-first search strategy used in Prolog resolution, with which those answers will be computed, is not well suited for test-case generation. One of **Ciao**'s features comes here to the rescue. **Ciao** has a concept of *packages*, syntactic and/or semantic extensions to the language that can be loaded module-locally. This mechanism is used to implement language extensions such as functional syntax, constraints, higher order, etc., and, in particular *alternative search rules*. These include for example (several versions of) breadth first, iterative deepening, Andorra-style execution, etc. These rules can be activated on a per-module basis. For example, the predicates in a module that starts with the following header:

```
1 :- module( myprops, _, [bf] ).
```

(which loads the **bf** package) will run in breadth-first mode. While breadth-first is useful mostly for teaching, other alternative search rules are quite useful in practice. Motivated by the **LPtest** context, i.e., with the idea of running properties in

generation mode, we have developed also a *randomized alternative search strategy* package, **rnd**, which can be described by the following simplified meta-interpreter:

```

1 solve_goal(G) :- random_clause(G,Body), solve_body(Body).
2
3 random_clause(Head,Body) :-
4     findall(cl(Head,B),meta_clause(Head,B),ClauseList),
5     once(shuffle(ClauseList,ShuffleList)),
6     member(cl(Head,Body),ShuffleList). % Body=[] for facts
7
8 solve_body([]).
9 solve_body([G|Gs]) :- solve_goal(G), solve_body(Gs).

```

The actual algorithm used for generation is of course more involved. Among other details, it only does backtracking on failure (on success it starts all over again to produce the next answer, without repeating traces), and it has a growth control mechanism to avoid getting stuck in traces that lead to non-terminating generations.

Using this search strategy, a set of terms satisfying a conjunction of pure Prolog properties can be generated just by running all those properties sequentially with unbounded variables. This is implemented using different versions of each property (generation, run-time check) which are generated automatically from the declarative definition of the property using instrumentation. In particular, this simplest subset of the language allows us to deal directly with regular types (e.g., `list/1`).

**Mode, sharing, and arithmetic constraints.** We extend the subset of the language for which generation is supported with arithmetic (e.g., `integer/1`, `float/1`, `</2`), mode-related extralogical predicates and properties (e.g., `var/1`, `ground/1`), and sharing-related native properties (e.g., `mshare/1`, which describes the sharing –aliasing– relations of a set of variables using *sharing sets* [15], and `indep/2`, that states that two variable do not share). When a goal or a property of this kind appears during generation, the variables occurring in it are constrained using a constraints domain. The domain ensures that those constraints are satisfiable during all steps of generation, failing and backtracking otherwise. There is a last step in generation in which all free variables are randomly further intantiated in a way that those constraints are satisfied.

This can be seen conceptually as choosing first a trace at random for each property and collecting constraints in the trace, and then randomly sampling (enumerating) the constrains. However, since the constraints introduced by unification are terms, it is equivalent to solving a predicate with the random search strategy and treating each builtin or native property as a constraint. In practice, we support more builtins for generation in properties (e.g., `==/2` just unifies two variables, we have shape constraints that handle `=./2`, and support negation to some extent), but the approach has only been tested significantly for the subset of Prolog presented so far.

In the last phase of constraints (random sampling), unconstrained free variables can be further instantiated with some probability, using random shape and sharing constraints, chosen among native properties and properties defined by the users on modules that are loaded at the time. This way, random terms are still generated for an assertion without precondition, or the generated term for `list(X)` is not always a list of free variables. This is also the technique used to further instantiate a free variable constrained as *ground* but for which no shape information is available.

**Generation for other properties.** For the remaining properties which use Prolog features not covered so far (e.g., dynamic predicates), there is a last step in the



generation algorithm in which they are simply checked for the terms generated so far. User-defined generators are encouraged for assertions with preconditions that are complex enough to reach this step. There is a limit to how many times generation can reach this step and fail, to avoid getting stuck in an inefficient or non-terminating generate-and-check loop. To recognize these properties without inspecting the code (left as future work), users are trusted to mark the properties suitable for generation with special syntax, and only the native properties discussed and the regular types are considered suitable by default.

## 4 Integration with static analysis

The use of a unified assertion framework for testing and analysis allows us to enhance `LPtest` random testing by combining it with static analysis.

First of all, as illustrated in Section 2, by performing static analysis first, some of the assertions can be (partially) verified statically, so they can be simplified or ignored for the testing phase. This process is performed automatically without any effort on `LPtest`'s part: `CiaoPP` is already capable of doing static assertion checking and simplification [13], and generates a new source file with the assertions left to be checked, which are the ones we test with our tool.

Beyond this, and perhaps more interestingly in our context, statically inferred information can also help while testing the remaining assertions. In particular, it is used to generate more relevant test cases in the generation phase. Consider for example the following assertion:

```
:- pred qs(Xs, Ys) => sorted(Ys).
```

Without the usual precondition, `LPtest` would have to generate arbitrary terms to test the assertion, most of which would not be relevant test cases since the predicate would fail for them, and therefore the assertion would be satisfied trivially. However, static analysis typically infers the output type for this predicate:

```
:- pred qs(Xs, _) => list(Xs, int).
```

I.e., analysis infers that on success `Xs` must be a list, and so on call it must be *compatible* with a list if it is to succeed. Therefore the assertion can also be checked as follows:

```
:- pred qs(Xs, _) : compat(Xs, list(int)) => sorted_int_list(Xs).
```

where `compat(Xs, list(int))` means that `Xs` is either a list of integers or can be further instantiated to one. Now we would only generate relevant inputs (generation for `compat/2` is implemented by randomly uninstantiating a term), and `LPtest` is able to prove the assertion false. The same can be done for modes and sharing to some extent: variables that are inferred to be free on success must also be free on call, and variables inferred to be independent must be independent on call too. Also, when a predicate is not exported, the *calls* assertions inferred for it can be used for generation. In general, the idea here is to perform some backwards analysis. However, this can also be done without explicit backwards analysis by treating testing and (forward) static analysis independently and one after the other, which makes the integration conceptually simple and easy to implement.

**A finer-grain integration.** We now propose a finer-grained integration of assertion-based testing and analysis, which still treats analysis as a black box, although not as an independent step. So far our approach has been to try to check an assertion with static analysis, and if this fails we perform random testing. However, the analysis often fails to prove the assertion because its precondition (i.e., the entry abstract substitution to the analysis) is too general, but it can prove it for refinements of that entry, i.e., refinements of the precondition. In that case, all test cases satisfying that refined precondition are guaranteed to succeed, and therefore useless in practice. We propose to work with different refined versions of an assertion, by adding further, exhaustive constraints in a native domain to the precondition, and performing testing only on the versions which the analysis cannot verify statically, thus pruning the test case input space. For example, for an assertion of a predicate of arity one, without mode properties, three different assertions equivalent to the first would result by adding to the precondition ( $\mathbf{ground}(X)$ ,  $\mathbf{var}(X)$ ), or ( $\mathbf{nonground}(X)$ ,  $\mathbf{nonvar}(X)$ ). The idea is to generalize this to arbitrary, maybe infinite abstract domains, for which an abstract value is not so easily partitioned as in the example above. It is still in development, but the core of the algorithm would be the following: to test an assertion to a given entry  $A \in D_\alpha$ , the assertion is proved by the analysis or tested recursively for a set of abstract values  $S \subseteq \{B \mid B \in D_\alpha, B \sqsubseteq A\}$  lower than that entry, and random test cases are generated in the “space” between the entry and those lower values  $\gamma(A) \setminus \bigcup \gamma(B)$ , where  $\gamma$  is the concretization function in the domain. For this it is only necessary to provide a suitable sampling function in the domain, and a rich generation algorithm for that domain. But note that, e.g., for the *sharing-freeness* domain, we already have the latter: we already have generation for mode and sharing constraints, and a transformation scheme between abstract values and mode/sharing properties. Note also that all this can still be done while treating the static analysis as a black box, and that if the enumeration of values is fine-grained enough, this algorithm also ensures coverage of the test input space during generation.

## 5 Shrinking

One flaw of random testing is that often the failed test cases reported are unnecessary complex, and thus not very useful for debugging. Many property-based tools introduce shrinking to solve this problem: after one counter-example is found, they try to reduce it to a simpler counter-example that still fails the test in the same way. **LPtest** supports shrinking too, both user-guided and automatic. We present the latter.

The shrinking algorithm mirrors that of generation, and in fact reuses most of the generation framework. It can be seen as a new generation with further constraints: bounds on the shape and size of the generated goal. The traces followed to generate the new term from a property must be “subtraces” of the ones followed to generate the original one. The random sampling of the constraints for the new terms must be “simpler” than for the original ones. The final step in which the remaining properties are checked is kept.

We present the algorithm for the first step. Generation for the shrunk value follows the path that leads to the to-be-shrunk value, but at any moment it can non-deterministically stop following that trace and generate a new subterm using size parameter 0. Applying this method to shrink lists of Peano numbers is equiva-

lent to the following predicate, where the first argument is the term to be shrunk and the second a free variable to be the shrunk value on success:

```

1 shrink_peano_list([X|Xs],[Y|Ys]) :-
2   shrink_peano_number(X,Y),
3   shrink_peano_list(Xs,Ys).
4 shrink_peano_list(_,Ys) :-
5   gen_peano_list(0,Ys). % X=[]
6
7 shrink_peano_number(s(X),s(Y)) :-
8   shrink_peano_number(X,Y).
9 shrink_peano_number(_,Y) :-
10  gen_peano_number(0,Y). % Y=0.

```

This method can shrink the list  $[s(0),0,s(s(s(0)))]$  to  $[s(0),0]$  or  $[s(0),0,s(s(0))]$ , but never to  $[s(0),s(s(s(0)))]$ . To solve that, we allow the trace of the to-be-shrunked term to advance non-deterministically at any moment to an equivalent point, so that the trace of the generated term does not have to follow it completely in parallel. It would be as if the following clauses were added to the previous predicate (the one which sketches the actual workings of the method during meta-interpretation):

```

1 shrink_peano_list([_|Xs],Ys) :-
2   shrink_peano_list(Xs,Ys).
3
4 shrink_peano_number(s(X),Y) :-
5   shrink_peano_number(X,Y).

```

With this method,  $[s(0),s(s(s(0)))]$  would now be a valid shrunked value.

This is implemented building shrinking versions of the properties, similarly to the examples presented, and running them in generation mode. However, since we want shrinking to be an enumeration of simpler values, and not random, the search strategy used is the usual depth-first strategy and not the randomized one presented in Sec. 3. The usual sampling of constraints is used too, instead of the random one.

The number of potential shrunked values grows exponentially with the size of the traces. To mitigate this problem, **LPtest** commits to a shrunked value once it checks that it violates the assertion too, and continues to shrink that value, but never starts from another one on backtracking. Also, the enumeration of shrunked values returns first the values closer to the original term, i.e., if **X** is returned before **Y**, then shrinking **Y** could never produce **X**. Therefore we never repeat a shrunked value<sup>4</sup> in our greedy search for the simplest counterexample.

## 6 A Case Study

In order to better illustrate our ideas, we present now a case study which consists in testing the correctness of the implementation of some of **CiaoPP**'s abstract domains. In particular, we focus herein on the *sharing-freeness* domain [19] and the correctness of its structure as a lattice and its handling of builtins. Tested predicates include **leq/2**, which checks if an abstract value is below another in the lattice, **lub/3** and **glb/3**, which compute the *least upper bound* and *greatest lower bound* of two abstract values, **builtin\_success/3**, which computes the *success substitution of a builtin from a call substitution*, and **abstract/2**, which computes the abstraction for a list of substitutions.

<sup>4</sup> Actually, we do not repeat subtraces, but two different subtraces can represent the same value (e.g., there are two ways to obtain  $s(0)$  from  $s(s(0))$ ).

*Generation.* Testing these predicates required generating random values for abstract values and builtins. The latter is simple: a simple declaration of the regular type `builtin(F,A)`, which simply enumerates the builtins together with their arity, is itself a generator, and using the generation scheme proposed in Sec. 3 it becomes a random generator, while it can still be used as a checker in the run-time checking framework. The same happens for a simple declarative definition of the property `shfr(ShFr,Vs)`, which checks that `ShFr` is a valid *sharing-freeness* value for a list of variables `Vs`. This is however not that trivial and proves that our generation scheme works and is useful in practice, since that property is not a regular type, and among others it includes sharing constraints between free variables. These two properties allowed us to test assertions like the following:

```

1 :- pred leq_reflexive(X) : shfr(X,_) + not_fails.
2 leq_reflexive(X) :- leq(X,X).
3
4 :- pred lub(X,Y,Z) : (shfr(X,Vs), shfr(Y,Vs)) => (leq(X,Z), leq(Y,Z)).
5
6 :- pred builtin_success(Blt,Call,Succ)
7   : (builtin(F,A), Blt=F/A, length(Vs,A), shfr(Call,Vs))
8   + (not_fails, is_det, not_further_inst([X,Y]))}

```

To check some assertions we needed to generate related pairs of abstract values. That is encoded in the precondition as a final literal `leq(ShFr1,ShFr2)`, as in the next assertion:

```

1 :- pred builtins_monotonic(Blt, X, Y)
2   : (builtin(Blt), Blt=F/A, length(Vs,A), shfr(X,Vs), shfr(Y,Vs), leq(X,Y))
3   + not_fails.
4
5   builtins_monotonic(Blt,X,Y) :-
6   ....

```

In our framework the generation is performed by producing first the two values independently, and checking the literal. This became inefficient, so we decided to write our own generator for this particular case. Finally, we tested the generation for arbitrary terms with the following assertion, which checks that the abstract value resulting from executing a builtin and abstracting the arguments on success is lower than the one resulting of abstracting the arguments on call and calling `builtin_success/3`:

```

1 :- pred builtin_soundness(Blt, Args)
2   : (builtin(Blt), Blt=F/A, length(Args,A), list(Args, term))
3   + not_fails.
4
5 builtin_soundness(Blt,Args) :- ...

```

*Analysis.* Many properties used in our assertions were user-defined and not native to `CiaoPP`, so the analysis could not abstract them precisely. However, the analysis did manage to simplify or prove some of the remaining ones, particularly those dealing with determinism (+ `is_det`) and efficiency (`no_choicepoints`).

Additionally, we successfully did the experiment of not defining the regular type `builtin/2`, and letting the analysis infer it on its own and use it for generation. We also tested by hand the finer integration between testing and analysis proposed in Sec. 4: some assertions involving builtins could not be proven for the general case, but this could be done for some of the simpler builtins, and thus testing could be avoided for those particular cases.

*Bugs found.* We did not find any bugs for the predicates `leq/2`, `lub/2`, and `glb/2`. This was not surprising; they are relatively simple and commonly used in `CiaoPP`. However, we found several bugs in `builtin_success/2`. Some of them were minor and thus had never been found or reported before: some builtin handlers left unnecessary choicepoints, or failed for the abstract value  $\perp$  (with which they are never called in `CiaoPP`). Others were more serious: we found bugs for less commonly-used builtins, and even two larger bugs for the builtins `=/2` and `==/2`. The handler failed for the literal `X=X` and for literals like `f(X)==g(Y)`, both of which do not normally appear in realistic programs and thus were not detected before.

## 7 Related Work

Random testing has been used for a long time in Software Engineering [8]. `QuickCheck` [3] provided the first implementation of a property-based random test generation system. It was first developed for Haskell and functional programming languages in general and then extended to other languages, and has seen significant practical use [14]. It uses a domain-specific language of testable specifications and generates test data based on Haskell types.

`ErlangQuickCheck` and `PropEr` [20] are closely related systems for Erlang, where types are dynamically checked and the value generation is guided by means of functions, using quantified types defined by these generating functions. We use a number of ideas from `QuickCheck` and the related systems, such as applying shrinking to reduce the test cases. However, `LPtest` is based on the ideas of the (earlier) `Ciao` model and we do not propose a new assertion language, but rather use and extend that of the `Ciao` system. This allows supporting Prolog-relevant properties, which deal with non-ground data, logical variables, variable sharing, etc., while `QuickCheck` is limited to ground data. Also, while `QuickCheck` offers quite flexible control of the random generation, we argue that using random search strategies over predicates defining properties is an interesting and more natural approach for Prolog.

The closest related work is `PrologTest` [1], which adapts `QuickCheck` and random property-based testing to the Prolog context. We share many objectives with `PrologTest` but we argue that our framework is more general, with richer properties (e.g., variable sharing), and is combined with static analysis. Also, as in `QuickCheck`, `PrologTest` uses a specific assertion language, while, as mentioned before, we share the `Ciao` assertions with the other parts of the `Ciao` system. `PrologTest` also uses Prolog predicates as random *generators*. This can also be done in `LPtest`, but we also propose an approach which we argue is more elegant, based on separating the code of the generator from the random generation strategy, using the facilities present in the `Ciao` system for running code under different SLD *search rules*, such as breadth first, iterative deepening, or randomized search.

Other directly related systems are `EasyCheck` [2] and `CurryCheck` [9] for the `Curry` language. In these systems test cases are generated from the (strong) types present in the language, as in `QuickCheck`. However, they also deal with determinism and modes. To the extent of our knowledge test case minimization has not been implemented in these systems.

There has also been work on generating test cases using CLP and partial evaluation techniques, both for Prolog and imperative languages (see, e.g., [7, 6] and its references). This work differs from (and is complementary to) ours in that the test

cases are generated via a symbolic execution of the program, with the traditional aims of path coverage, etc., rather than from assertions and with the objective of randomized testing.

Other related work includes *fuzz testing* [18], where “nonsensical” (i.e., fully random) inputs are passed to programs to trigger program crashes, and grammar-based testing, where inputs generation is based on a grammatical definition of inputs (similar to generating with regular types) [10]. Schrijvers proposed **Tor** [23] as a mechanism for supporting the execution of predicates using alternative search rules. Midtgaard and Moller [17] have also applied property-based testing to checking the correctness of static analysis implementations.

## 8 Conclusions and future work

We have presented an approach and a tool, **LPtest**, for assertion-based random testing of Prolog programs that is integrated with the **Ciao** assertion model. In this context, the idea of generating random test values from assertion preconditions emerges naturally since preconditions are conjunctions of literals, and the corresponding predicates can conceptually be used as generators. **LPtest** generates valid inputs from the properties that appear in the assertions shared with other parts of the model. We have shown how this generation process can be based on running the property predicates under non-standard (random) search rules and how the run time-check instrumentation of the **Ciao** framework can be used to perform a wide variety of checks. We have proposed methods for supporting (C)LP-specific properties, including combinations of shape-based (regular) types and variable sharing and instantiation. We have also proposed some integrations of the test generation system with static analysis and provided a number of ideas for shrinking in our context. Finally, we have shown some results on the applicability of the approach and tool to the verification and checking of the implementations of some of **Ciao**’s abstract domains. The tool is rather new and we are gathering experience with it (which we expect to report on at the conference). However, it has already proven itself quite useful in finding bugs in code that had survived years of use and debugging.

## References

1. C. Amaral, M. Florido, and V. Santos Costa. PrologCheck - Property-Based Testing in Prolog. In *Functional and Logic Programming - 12th Int’l. Symp., FLOPS*, volume 8475 of *LNCS*, pages 1–17. Springer, 2014.
2. Jan Christiansen and Sebastian Fischer. EasyCheck - Test Data for Free. In *Functional and Logic Programming, 9th Int’l. Symp., FLOPS*, pages 322–336, April 2008.
3. Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Fifth ACM SIGPLAN Int’l. Conf. on Functional Programming*, ICFP’00, pages 268–279. ACM, 2000.
4. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL’77*, pages 238–252. ACM Press, 1977.
5. Cormac Flanagan. Hybrid Type Checking. In *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006*, pages 245–256, January 2006.

6. M. Gómez-Zamalloa, E. Albert, and G. Puebla. On the Generation of Test Data for Prolog by Partial Evaluation. In *Proc. of WLPE'08*, pages 26–43, 2008.
7. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Test Case Generation for Object-Oriented Imperative Languages in CLP. *Theory and Practice of Logic Programming, ICLP'10 Special Issue*, 10 (4–6), 2010.
8. Dick Hamlet. Random Testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, page 970–978. Wiley, 1994.
9. Michael Hanus. CurryCheck: Checking Properties of Curry Programs. In *Logic-Based Program Synthesis and Transformation - 26th Int'l. Symp. LOPSTR 2016, Revised Selected Papers*, pages 222–239, September 2016.
10. Mark Hennessy and James F. Power. An Analysis of Rule Coverage as a Criterion in Generating Minimal Test Suites for Grammar-based Software. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, pages 104–113, November 2005.
11. M. V. Hermenegildo, F. Bueno, M. Carro, P. Lopez-Garcia, E. Mera, J.F. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *TPLP*, 12(1–2):219–252, 2012. <http://arxiv.org/abs/1102.5497>.
12. M. V. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 161–192. Springer-Verlag, 1999.
13. M. V. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.
14. John Hughes. QuickCheck Testing for Fun and Profit. In Michael Hanus, editor, *Practical Aspects of Declarative Languages*, pages 1–32, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
15. D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *North American Conference on Logic Programming*, 1989.
16. E. Mera, P. Lopez-Garcia, and M. V. Hermenegildo. Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In *25th Int'l. Conference on Logic Programming (ICLP'09)*, volume 5649 of *LNCS*, pages 281–295. Springer-Verlag, July 2009.
17. Jan Midtgaard and Anders Møller. QuickChecking Static Analysis Properties. *Softw. Test., Verif. Reliab.*, 27(6), 2017.
18. Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.
19. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *ICLP'91*, pages 49–63. MIT Press, June 1991.
20. Manolis Papadakis and Konstantinos Sagonas. A PropEr Integration of Types and Function Specifications with Property-Based Testing. In *10th ACM SIGPLAN workshop on Erlang*, pages 39–50, September 2011.
21. G. Puebla, F. Bueno, and M. V. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Proc. of LOPSTR'99*, LNCS 1817, pages 273–292. Springer-Verlag, March 2000.
22. A. Rastogi, N. Swamy, C. Fournet, G.M. Bierman, and P. Vekris. Safe & Efficient Gradual Typing for TypeScript. In *42nd POPL*, pages 167–180. ACM, January 2015.
23. Tom Schrijvers, Bart Demoen, Markus Triska, and Benoit Desouter. Tor: Modular search with hookable disjunction. *Sci. Comput. Program.*, 84:101–120, 2014.
24. Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.