Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**

Máster Universitario en Inteligencia Artificial

Trabajo Fin de Máster

An Integrated Approach to Assertion-Based Random Testing in Logic Languages

Autor(a): Ignacio Casso San Román
Tutor(a): Manuel Hermenegildo Salinas

Madrid, Julio 2021

Trabajo Fin de Máster
Máster Universitario en Inteligencia Artificial

Título: An Integrated Approach to Assertion-Based Random Testing in Logic Languages

Julio 2021

Autor(a): Ignacio Casso San Román
Tutor(a): Manuel Hermenegildo Salinas
Departamento de Inteligencia Artificial
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

En este trabajo presentamos un método para testear programas en Prolog, integrado en un modelo de desarrollo basado en aserciones. Nuestro punto de partida es el lenguaje `Ciao` y su modelo de aserciones, que unifica la verificación y comprobación de errores dinámica y estática, usando un lenguaje de aserciones común. En este modelo, cuando alguna propiedad no se puede verificar estáticamente, se instrumenta el código fuente para que la propiedad se compruebe dinámicamente en tiempo de ejecución, y en particular en la fase de tests unitarios, si se han especificado tests con valores de entrada concretos. En este contexto, la idea de generar automáticamente entradas aleatorias para los tests unitarios a partir de las precondiciones de las aserciones surge de manera natural, dado que estas precondiciones son conjunciones de literales, y sus predicados correspondientes se pueden usar en principio como generadores. En este trabajo desarrollamos `LPtest`, una herramienta que implementa la idea anterior, generando automáticamente entradas relevantes para testear una amplia variedad de propiedades de los predicados de un programa. El algoritmo de generación está basado en ejecutar los predicados usando reglas de búsqueda aleatorias. Se proponen también métodos para soportar propiedades específicas de programación lógica, incluyendo combinaciones de tipos y compartición e instanciación de variables, así como ideas para reducir los casos de prueba y para mejorar la generación aprovechando la información inferida por el análisis estático.

Como caso de estudio de la herramienta, se aplica `LPtest` al testeo del analizador estático basado en interpretación abstracta de `Ciao`. Se proponen dos métodos diferentes para abordar el problema usando testeo aleatorio basado en aserciones. En el primero, se especifican mediante aserciones de `Ciao` algunas propiedades teóricas básicas que deben cumplir los dominios abstractos y sus operaciones para ser correctos. Después se aplica la herramienta sobre esas aserciones para validar el código de algunos dominios abstractos en `Ciao`. El segundo método consiste en comprobar, para un conjunto de programas de prueba, que las aserciones que infiere estáticamente el analizador se satisfacen dinámicamente, usando para ello `LPtest`. Esta idea se puede implementar y automatizar fácilmente aprovechando el sistema de aserciones integrado de `Ciao` y sus componentes: el *analizador estático*, que expresa sus resultados mediante un nuevo programa idéntico al original pero con aserciones intercaladas; y `LPtest`, que genera y ejecuta casos de prueba que satisfacen las propiedades que aparecen en las precondiciones de dichas aserciones. Los dos métodos se implementan y se aplican para testear `CiaoPP` y sus dominios abstractos, encontrando en el proceso errores no triviales y desconocidos anteriormente.

Abstract

We present an approach for assertion-based random testing of Prolog programs that is tightly integrated within an overall assertion-based program development scheme. Our starting point is the Ciao 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. We develop LPtest, a tool that generates valid inputs from the properties that appear in the assertions shared with other parts of the model, and uses the run time-check instrumentation of the Ciao framework to perform a wide variety of checks. The generation process is based on running standard predicates under non-standard (random) search rules. 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 and for enhancing generation by exploiting information inferred by the analyzer.

As a case study, we apply LPtest for testing Ciao's abstract interpretation-based static analyzer. We approach this problem, using assertion-based random testing, in two different ways. In the first one, we encode into Ciao assertions some basic theoretical properties that abstract domains must satisfy in order to be sound. Then we apply the tool to those assertions to check and validate the code of some the abstract domains in the Ciao system. The second method consists in checking, over a suite of benchmarks, that the assertions inferred statically by the analyzer are satisfied dynamically, testing them with LPtest. This can be implemented and automatized with little effort by framing it within the Ciao integrated assertion framework and combining its components: the *static analyzer*, which outputs its results as the original program source with assertions interspersed; and the newly developed random testing framework, which generates and executes random test cases satisfying the properties present in assertion preconditions, relying on the *run-time checking* instrumentation and the *unit-test framework*. We apply both approaches to test some of CiaoPP's analysis domains, successfully finding non-trivial, previously undetected bugs.

Contents

Chapter 1

Introduction

1.1 Motivation and Objectives

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.

The Ciao logic programming language [26] introduced a novel development workflow [5, 28, 30, 50] that integrates the two approaches above. In this model, program assertions are fully integrated 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.

Ciao assertions are linguistic constructs, which allow expressing properties of programs. They describe predicates in an abstract way, and can state declarative properties of calling and success states, such as variable types or variable sharing, and global properties of the execution, such as determinacy, non-failure, or resource usage. These properties are themselves predicates, typically written in the source language (user-defined or in libraries), and thus runnable.

The Ciao debugging framework uses these assertions as the specifications of the expected semantics and behaviour of the predicates in a program. To validate them, first Ciao's abstract interpretation-based preprocessor, CiaoPP, tries to verify (parts of) them statically. When this is not possible, *runtime-checks* instrumentation can be added to the program in order to ensure that execution paths that violate the assertions are captured during execution. Since *run-time checks* can become expensive if used indiscriminately, they are most often used before deployment as part of the unit-testing framework. Test inputs can be provided by the user, and the test driver executes them and reports the errors captured at run-time.

1.1.1 Towards Automatic Generation: Assertion-Based Random Testing

The unit-testing framework in principle requires the user to manually write individual test cases for each assertion to be tested. Hand-written test cases such as those 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 assertion properties in particular) can be used as both checkers and generators. This leads naturally to the idea of generating systematically and automatically test cases by running in generation mode the properties in the preconditions of assertions. While this idea of using properties as test case generators has always been present in the descriptions of the `Ciao` model [28, 50], it has not really been exploited or researched significantly to date. Our purpose in this work is to fill this gap.

We report on the development of `LPtest` an implementation of random testing [23] with a more natural connection with Prolog semantics, as well as with the `Ciao` framework. The goal of `LPtest` is to automatically generate and run relevant test cases which exercise the *run-time checks* of the assertions in a program, thus testing if those assertions are correct. We refer to the combination of such test generation mechanism with the run-time checking of the intervening assertions as *assertion-based (random) testing*.

1.1.2 Testing the Static Analyzer

Static analysis is nowadays an essential component of many software development toolsets. Despite some notorious successes in the validation of compilers, comparatively little work exists on the systematic validation of static analyzers, whose correctness and reliability is critical if they are to be inserted in production environments. Contributing factors may be the intrinsic difficulty of formally verifying code that is quite complex and of finding suitable oracles for testing it.

Another goal of this work is to apply `LPtest` or assertion-based random testing to tackle this problem, in the context of `Ciao` and `CiaoPP`, its abstract interpretation-based static analyzer. We do so in two different ways. The first tests only the code of abstract domains and uses `LPtest` in a straightforward manner: we encode into `Ciao` assertions some theoretical properties that abstract domains must satisfy in order to be sound, and then we apply the tool to validate those assertions.

The second method can be used to test the analyzer as a whole in a simple and automatic way, exploiting `Ciao`'s unified assertion framework. Broadly, it consists in checking, over a suite of benchmarks, that the properties inferred statically are satisfied dynamically. For each benchmark, the code is analyzed, which produces as a result a new file with the original code and the inferred assertions interspersed. Then, those assertions are tested using `LPtest`. If any assertion violation is reported, it means that the assertion was incorrectly inferred by the analyzer and thus that an error in the analyzer has been found.

1.2 Contributions

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 [14], 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 [48]. 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.

- We have applied assertion-based random testing as a case study to test soundness properties of CiaoPP's abstract domains, encoded manually into Ciao assertions.

- We have proposed a simple, automatic method for testing abstract interpretation-based static analyzers based on checking that the properties inferred statically are satisfied dynamically. Although the idea of checking at run time the properties or assertions inferred by the analysis for different program points is not new, we argue that is made more applicable, general, and scalable by the use of a unified assertion-based framework for static analysis and dynamic debugging, as the one of Ciao. While other approaches require the development of tailored instrumentation or monitoring, and require significant effort in their design and implementation, by framing it in the Ciao framework we can implement with the already existing components in the system in a very simple way (so much so that our initial prototype was, in fact, barely 50 lines of code long).

- The work on assertion-based random testing has been published in the Post-Proceedings of the 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'19), volume 12042 of LNCS [11]; and the work on testing the assertions inferred by the analyzer, in the Post-Proceedings of the 30th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'20), volume 12561 of LNCS [12].

1.3 Outline

The rest of this document is organized as follows:

Chapter 2 gives background knowledge needed to describe the main ideas and contributions of this work, that is, a more detailed description of Ciao's unified assertion framework and CiaoPP's static and dynamic debugging framework prior to this work.

Chapter 3 is dedicated to assertion-based random testing and our tool LPtest. First we review in Section 3.1 our approach to assertion-based testing in the context of Prolog and Ciao. In Sec. 3.2 we introduce our test input generation schema. In Sec. 3.3, we show how assertion-based testing can be combined with and enhanced by static analysis. Sec. 3.4 is dedicated to shrinking of test cases in LPtest.

In Chapter 4 we present our two approaches to testing static analyzers using assertion-based random testing, in Section 4.1 and Section 4.2 respectively.

Finally, Chapter 5 discusses related work and Chapter 6 summarizes our conclusions and plans for future work.

Chapter 2

Background

2.1 Ciao and The Ciao Model

In this chapter we review in more detail those aspects of the Ciao model that are relevant to our approach, including the assertion language and the blended static and dynamic assertion checking framework built around it. A more detailed presentation can be found in [6, 29, 51, 30, 43, 27] and their references.

2.2 The Assertion Language.

Ciao assertions are linguistic constructs, which allow expressing properties of programs. There are two types of assertions in Ciao that are relevant herein: *predicate* assertions and *program-point* assertions. The first ones are declarations that provide partial specifications of a predicate. They have the following syntax: `:- [Status] pred Head : [Calls] => [Success] + [Comp]`, indicating that if a call to the goal `Head` satisfies precondition `Calls`, it must satisfy post-condition `Success` on success and global computational properties `Comp`. *Program-point* assertions are reserved literals that appear in clause bodies and describe the constraint store at the corresponding program point. Their syntax is `[Status](State)`. Examples of both types of assertions are provided in the code fragment below:

```
:- check pred append(X,Y,Z) : (list(X),list(Y)) => list(Z) + is_det.
:- check pred append(X,Y,Z) : (var(X),var(Y),list(Z)) => (list(X),list(Y)) + non_det.

append([],X,X).
append([X|Xs],Ys,[X|Zs]) :-
  append(Xs,Ys,Zs),
  check((list(Xs),list(Ys),list(Zs))).
```

Assertion fields `Calls`, `Success`, `Comp` and `State`, are conjunctions of *properties*. Such properties are predicates, typically written in the source language (user-defined or in libraries) and thus runnable. They must meet certain conditions (e.g., termination) [30] and are marked as such via `prop/1` declarations. Since they are runnable, they can be used as run-time checks, and, for our purposes, they are also typically *native* to CiaoPP, i.e., abstracted and inferred by some domain in CiaoPP. A wide range of properties is supported natively: from types, modes and variable sharing, to deter-

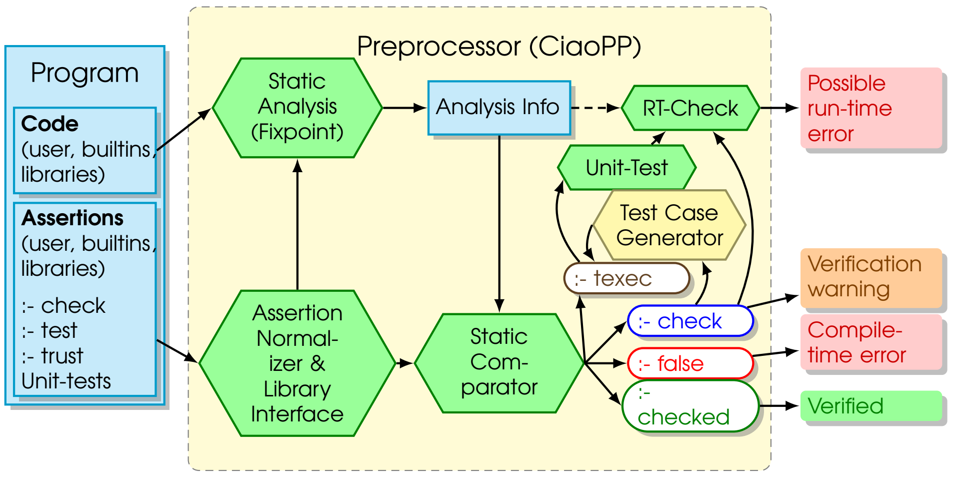

Figure 2.1: The Ciao assertion framework (CiaoPP's verification/testing architecture).

minism, (non)failure and resource consumption. We refer the reader to [49, 30, 27] and their references for a full description of the Ciao assertion language.

Assertions are used everywhere in Ciao, from documentation and foreign interface definitions to static analysis and dynamic debugging. Depending on their origin and intended use, they have a different status, the Status field in the syntax described above. Assertion statuses relevant herein include true, which is used for assertions that are output from the analysis (and thus must be safe approximations), or the default status check, which indicates that the validity of the assertion is unknown and it must be checked, statically or dynamically. We will return to this crucial issue below.

Fig. 3.1 depicts the overall architecture of the Ciao unified assertion framework. Hexagons represent tools, and arrows indicate the communication paths among them. The input to the process is the user program, *optionally* including a set of assertions; this set always includes any assertion present for predicates exported by any libraries used (left part of Fig. 3.1).

2.3 Static Analysis.

One use of Ciao assertions is as an interface to the static analyzer. As mentioned above, assertions can be used to indicate what we want the analyzer to check (the default check status), or to guide the analysis by feeding it information that it might be unable to infer by itself (trust status). The latter includes as a special case providing information on the entry points to the module being analyzed (i.e., on the calls to the predicates exported by the module –entry status). Assertions are also one of the possible output formats in which the analysis results are produced by the static analyzer (assertions with true status). If this type of output is chosen, a new source file for the analyzed program will be created, exactly as the original but with true *program-point* assertions interspersed between every two consecutive literals of each clause, and with one or more true *predicate* assertions for each predicate.

The exact technical and theoretical details of how this is achieved are out of the scope of this work. For our purposes it is sufficient to say that the CiaoPP analyzer is abstract interpretation-based, and its design consists of a common abstract-interpretation framework (the fixpoint algorithm(s)) parameterized by different, “plug-gable” abstract domains. Depending on the domain or combination of domains selected for the analysis, different properties will be inferred and will appear in the emitted `true` assertions or be used to verify or simplify `check` assertions.

2.4 Ciao’s Debugging Framework

Consider the following `Ciao` code and assertion (with the standard definition of quick-sort):

```
:- pred qs(Xs,Ys) : (list(Xs), var(Ys)) => (list(Ys), sorted(Ys)) + not_fails.

:- prop list/1.
list([]).
list([_|T]) :- list(T).

:- prop sorted/1.
...
```

This assertion has a *calls* field (the conjunction after ‘:’), a *success* field (the conjunction after ‘=>’), and a computational properties field (after ‘+’). 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.

The first step to validate that assertion in `Ciao`’s debugging framework would be to use static analysis to either detect an assertion violation, or to prove (parts of) it, verifying (simplifying) the assertion, as is illustrated in Fig. 3.1 (the Static Comparator).

For example, compile-time analysis with a types/*shapes* domain can easily detect that, if the predicate in the example above 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).
```

Cases like this, where (parts of) assertions can not be proven nor disproven statically, are quite common, due to the inherent imprecision of the analysis, specially with user-defined properties that are not native to abstract domains. In those occasions, the remaining unproved (parts of) assertions are written into the output program with *check* status and then this output program can optionally be instrumented with *run-time checks*. These dynamic checks will encode the meaning of the *check* assertions, ensuring that an error is reported at run-time if any of these remaining assertions is violated (the dynamic part of the model). Note that the fact that properties are written in the source language and runnable is essential in this process, and allows checking new user-defined and native properties without having to extend the *run-time checking* framework.

For our example, sorted(Ys) will be called at run time within the assertion checking-harness, right after calls to qs/2, and if the property fails and error will occur.

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.:

```
:- pred revrev(X) : list(X) + not_fails.
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 [43]. For example, if the following (unit) tests are added to qs/2:

```
:- test qs(Xs,Ys) : (Xs = []) => (Ys = []).
:- 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* [43]:

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

and the rest of the work (checking the assertion fields) is done by the standard assertion 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. This overall process is depicted in Fig. 3.1.

Chapter 3

Assertion-Based Random Testing

3.1 Overview

As mentioned before, the goal of LPtest is to integrate random testing of assertions within Ciao's assertion-based verification and debugging framework (Fig. 3.1). 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 also mentioned in the introduction, the Ciao framework already provides most of the components 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 concrete goals to check those assertions. We only need to be able to generate terms satisfying the assertion preconditions and feed them into the other parts of the framework (the new yellow box in Fig. 3.1). This generation of test cases is discussed in Sec. 3.2, 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:

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

Following Fig. 3.1, the assertions of the qs module are verified statically [30]. As a result, parts of each assertion may be proved true or false (in which case no testing is needed for them), and, if any other parts are left after this process, run-time checking and/or testing is performed for them. 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:

```
Testing assertion:
```

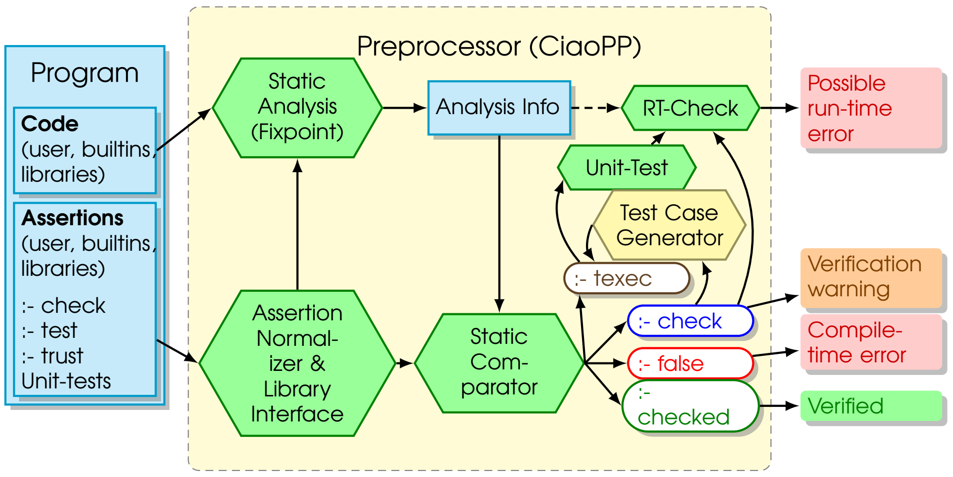

Figure 3.1: The Ciao assertion framework (CiaoPP's verification/testing architecture).

```
:- pred qs(Xs,Ys) : (list(Xs,int), var(Ys))
                 => (list(Ys,int), sorted(Ys)) +  not_fails.

Assertion was partially verified statically:
  :- checked pred qs(Xs,Ys) (list(Xs,int), var(Ys)) => list(Ys,int).
Left to check::
  :- check   pred qs(Xs,Ys) => sorted(Ys) + not_fails.
```

LPtest will then try to 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 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:

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

```
Test case shrinked to:
   :- texec qs([0,0],_).
```

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

1. (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. 3.2. For each test case, go to **3**. Then go to **4**.
3. (RTcheck) 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 RTcheck. For each of them, go to **5** to shrink them, and then report them (using RTcheck).
5. (LPtest) Generate a simpler test case not generated yet.
   - If not possible, finalize and return current test case as shrinked 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.

3.1.1 Examples

The use of the Ciao static verification and 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:

3.1.1.1 (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:

```
:- pred rev(Xs,Ys) : list(Xs) => list(Ys).
:- pred sort(Xs,Ys) : list(Xs,int) => (list(Ys,int), sorted(Ys)).
:- pred numbervars(Term,N,M) => ground(Term).
:- 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.

3.1.1.2 Computational Properties.

LPtest can also be used to check properties regarding the computation of a predicate. These properties are typically 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:

```
:- pred rev(X,Y) : list(X) + (not_fails, is_det, no_choicepoints).
:- pred append(X,Y,Z) : list(X) => cost(steps,ub,length(X)).
```

3.1.1.3 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:

```
:- prop sorted_int_list(X).

sorted_int_list([]).
sorted_int_list([N]) :- int(N).
sorted_int_list([N,M|Ms]) :- N =< M, sorted_int_list([M|Ms]).

:- pred insert_ord(X,Xs,XsWithX)
    : (int(X), sorted_int_list(X))
   => sorted_int_list(XsWithX).
```

3.2 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.

3.2.1 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:

```
:- 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:

```
solve_goal(G) :- random_clause(G,Body), solve_body(Body).

random_clause(Head,Body) :-
    findall(cl(Head,B),meta_clause(Head,B),ClauseList),
    once(shuffle(ClauseList,ShuffleList)),
    member(cl(Head,Body),ShuffleList). % Body=[] for facts

solve_body([]).
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).

3.2.2 Mode, Sharing, and Arithmetic Constraints.

We extend the subset of the language for which generation is supported with arithmetic (e.g., int/1, flt/1, </2), mode-related extralogical predicates and properties (e.g., free/1, gnd/1), and sharing-related native properties (e.g., mshare/1, which describes the sharing –aliasing– relations of a set of variables using *sharing sets* [32], 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 instantiated 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.

3.2.3 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, and only the native properties discussed and the regular types are considered suitable by default.

3.3 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 Sec. 2.1 and Fig. 3.1, `CiaoPP` first performs a series of static analyses through which some of the assertions may be verified statically, possibly partially. Thus, only some parts of some assertions may need to be checked in the testing phase [30].

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 (inputs that generate failure are also interesting of course, but not to check properties that should hold on *success*). 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 variabless 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.

3.3.1 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, a set of assertions equivalent to the original one would result by generating three different assertions with the same success but preconditions `ground(X)`, `var(X)`, and `(nonground(X), nonvar(X))`. The idea is to generalize this to arbitrary, maybe infinite abstract domains, for which a given abstract value is not so easily partitioned as in the example above. Alternatively, the test exploration can be limited to subsets of the domain, since in any case the testing process cannot achieve completeness in general. The core of an algorithm for this domain partition would be the following: to test an assertion for a given entry $A \in D_\alpha$, the assertion is proved by the analysis or tested recursively for a set of abstract values $S \subseteq \{B | B \in D_\alpha, B \sqsubseteq A\}$ 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 γ 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 abstract values is fine-grained enough, this algorithm also ensures coverage of the test input space during generation.

3.4 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 shrinked value follows the path that leads to the to-be-shrinked 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 equivalent to the following predicate, where the first argument is the term to be shrinked and the second a free variable to be the shrinked value on success:

```
shrink_peano_list([X|Xs],[Y|Ys]) :-
  shrink_peano_number(X,Y),
  shrink_peano_list(Xs,Ys).
shrink_peano_list(_,Ys) :-
  gen_peano_list(0,Ys). % X=[]

shrink_peano_number(s(X),s(Y)) :-
  shrink_peano_number(X,Y).
shrink_peano_number(_,Y) :-
  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-shrinked 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 the previous predicate (the one which sketches the actual workings of the method during meta-interpretation):

```
shrink_peano_list([_|Xs],Ys) :-
  shrink_peano_list(Xs,Ys).

shrink_peano_number(s(X),Y) :-
  shrink_peano_number(X,Y).
```

With this method, `[s(0),s(s(s(0)))]` would now be a valid shrinked 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. 3.2. The usual sampling of constraints is used too, instead of the random one.

The number of potential shrinked values grows exponentially with the size of the traces. To mitigate this problem, `LPtest` commits to a shrinked 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 shrinked 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 shrinked value [1] in our greedy search for the simplest counterexample.

[1]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))).

Chapter 4

Testing Static Analyzers

Static analysis tools are nowadays a crucial component of the development environments for many programming languages. They are widely used in different steps of the software development cycle, such as code optimization and verification, and they are the subject of significant research interest and practical application. Unfortunately, modern analyzers are often very large and complex software artifacts, and this makes them prone to bugs. This is a limitation to their applicability in real-life production compilers and development environments, where they are typically used in critical tasks like verification or code optimization, that need to rely strongly on the soundness of the analysis results.

However, the validation of static analyzers is a challenging problem, which is not well covered in the literature or by existing tools. Well-established methodologies or even guidelines to this end do not really exist. This is due to the fact that direct application of formal methods is not always straightforward with code that is so complex and large, even without considering the problem of having precise specifications to check against —a clear instance of the classic problem of who checks the checker. In current practice, extensive testing is the most extended and realistic option, but it poses some significant challenges too. Testing separate components of the analyzer misses integration testing, and designing proper oracles for testing the complete tool is really challenging.

Our objective in this chapter is to apply assertion-based random testing to validate and debug `CiaoPP`, `Ciao`' abstract interpretation-based static analyzer, which faces this very problem. As other "classic" analyzers, this analyzer has evolved for a long time, incorporating a large number of abstract domains, features, and techniques, adding up to over 1/2 million lines of `Ciao` code. These components have in turn reached over the years different levels of maturity. While the essential parts, such as the fixpoint algorithms and the classic abstract domains, have been used routinely for a long time now and it is unusual to find bugs, other parts are less developed and yet others are prototypes or even proofs of concept.

In Section 4.1, we encode into `Ciao` assertions some theoretical properties that `CiaoPP`'s abstract domains must satisfy in order to be sound, and apply `LPtest` to validate those assertions as a case study. In Section 4.2, we check with `LPtest`, over a suite of benchmarks, that the assertions inferred statically with `CiaoPP` are satisfied dynamically.

4.1 Testing Abstract Domain Properties

In order to better illustrate the ideas presented in the previous chapter, we present in this section a case study of LPtest 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 [46] 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.

4.1.1 Generation.

Testing these predicates required generating random values for abstract values and builtins. The latter is simple: a simple declaration of the property builtin(F,A), which simply enumerates the builtins together with their arity, is itself a generator, and using the generation scheme proposed in Sec. 3.2 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:

```
:- pred leq_reflexive(X) : shfr(X,_) + not_fails.
leq_reflexive(X) :- leq(X,X).

:- pred lub(X,Y,Z) : (shfr(X,Vs), shfr(Y,Vs)) => (leq(X,Z), leq(Y,Z)).

:- pred builtin_sucess(Func,Ar,Call,Succ)
    : (builtin(Func,Ar), length(Vs,Ar), shfr(Call,Vs))
    + (not_fails, is_det, not_further_inst([Call]))}
```

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:

```
:- pred builtins_monotonic(F, A, X, Y)
    : (builtin(F,A), length(Vs,A), shfr(X,Vs), shfr(Y,Vs), leq(X,Y))
    + not_fails.

    builtins_monotonic(F,A,X,Y) :-
        builtin_success(F,A,X,X2), builtin_success(F,A,Y,Y2), leq(X2,Y2).
```

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:

```
:- pred builtin_soundness(Blt, Args)
    : (builtin(Blt), Blt=F/A, length(Args,A), list(Args, term))
    + not_fails.

builtin_soundess(Blt,Args) :- ...
```

4.1.2 Analysis.

Many properties used in our assertions were user-defined, complex, and not native to CiaoPP, so there were many cases in which the analysis could not abstract them precisely. However, the analysis did manage to simplify or prove some of the remaining ones, particularly regular types and 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. 3.3: 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.

4.1.3 Bugs Found.

We did not find any bugs in the implementations for different domains of the lattice operations 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 (part of the description of the “transfer function” for some language built-ins) in some domains. 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 $\bot$ (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.

4.2 Testing Properties Inferred by the Analyzer

In this section, we propose an algorithm that combines LPtest and some other basic components of Ciao’s assertion framework in a novel way that allows testing the static analyzer almost for free. Intuitively, it consists in checking, over a suite of benchmarks, that the properties inferred statically are satisfied dynamically. The overall testing process, for each benchmark, can be summarized as follows: first the code is analyzed and the analysis results are expressed by the analyzer as assertions interspersed within the original code. Then these assertions are *switched* into run-time checks, that will ensure that violations of those assertions are reported at run time. Finally, random test cases are generated and executed to exercise those *run-time checks*. If any assertion violation is reported, since these assertions (the analyzer output) must cover all possible concrete executions, it means that the assertion was incorrectly inferred by the analyzer and thus that an error in the analyzer has been found. This process can be easily automated, and if it is repeated for an extensive and varied enough suite of benchmarks, it can be used to effectively validate (even if not fully verify) the analyzer or to discover new bugs. Furthermore, the implementation, when framed within the Ciao assertion-based validation framework, is very simple, since, as we will show, only a basic code transformation and a simple driver need to be implemented to obtain a very useful, working system.

The idea of checking at run time the properties or assertions inferred by the analysis for different program points is not new. For example, [57] successfully applied this technique for checking a range of different aliasing analyses. However, these approaches require the development of tailored instrumentation or monitoring, and require significant effort in their design and implementation. We argue that the testing approach is made more applicable, general, and scalable by the use of a unified assertion-based framework for static analysis and dynamic debugging, as the one of Ciao. As mentioned before, framing things in such a framework, the approach can be implemented with the already existing components in the system, in a very simple way, so much so that our initial prototype was, in fact, barely 50 lines of code long. We argue also that our approach is particularly useful in a mixed production and research setting like that of CiaoPP, in which there is a mature and domain-parametric abstract interpretation framework used routinely, but new, experimental abstract domains and overall improvements are in constant development. Those domains can easily be tested relying only on the existing abstract-interpretation framework, runtime-checking framework, and unified assertion language, provided only that the assertion language is extended to include the properties relevant for the domains.

4.2.1 Overview of the Approach

After introducing the relevant elements of the Ciao assertion model, we can now sketch the main idea of our approach with a motivating example. Assume we have this simple Prolog program, where the entry assertion indicates that the predicate is always called with its second argument instantiated to a list and the third a free variable:

```
:- entry prepend(X,Xs,Ys) : (list(Xs), var(Ys)).

prepend(X,Xs,Ys) :-
    Ys=[X|Rest],
    Rest=Xs.
```

Assume that we analyze it with a *simple modes* abstract domain that assigns to each variable in an abstract substitution one the following abstract values: *g* (variable is ground), *v* (variable is free), *ng* (variable is not ground), *nv* (variables is not free), *ngv* (variable is not ground nor free), or *any* (nothing can be said about the variable). Assume also that the analysis is incorrect because it does not consider sharing (aliasing) between variables, so when updating the abstract substitution after the Rest=Xs literal, the abstract value for Ys is not modified at all. The result of the analysis will be represented, as explained in Chapter 2, as a new source file with interspersed assertions, as shown in Fig. 4.1 (lines 3-5, 8, 10, and 12). Note that the correct result, if the analysis considered aliasing, would be that there is no groundness information for Ys at the end of the clause (line 12), since there is none for X or Xs at the beginning either. Ys could only be inferred to be *nonvar*, but instead is incorrectly inferred to be *nonground* too (line 10). Normally unknown/1 properties would not actually appear in the analysis output, but are included for clarity.

```
1    :- entry prepend(X,Xs,Ys) : (list(Xs), var(Ys)).
2
3    :- true pred prepend(X,Xs,Ys)
4        :  (unknown(X), nonvar(Xs), var(Ys))
5        => (unknown(X), nonvar(Xs), nonground(Ys), nonvar(Ys)).
6
7    prepend(X,Xs,Ys) :-
8        true((unknown(X), nonvar(Xs), var(Ys), var(Rest))),
9        Ys=[X|Rest],
10       true((unknown(X), nonvar(Xs), nonground(Ys), nonvar(Ys), var(Rest))),
11       Rest=Xs,
12       true((unknown(X), nonvar(Xs), nonground(Ys), nonvar(Ys), nonvar(Rest))).
```

Figure 4.1: An incorrect simple mode analysis.

```
1    :- entry prepend(X,Xs,Ys) : (list(Xs), var(Ys)).
2
3    :- check pred prepend(X,Xs,Ys)
4        :  (unknown(X), nonvar(Xs), var(Ys))
5        => (unknown(X), nonvar(Xs), nonground(Ys), nonvar(Ys)).
6
7    prepend(X,Xs,Ys) :-
8        check((nonvar(Xs), var(Ys), var(Rest))),
9        Ys=[X|Rest],
10       check((nonvar(Xs), nonground(Ys), nonvar(Ys), var(Rest))),
11       Rest=Xs,
12       check((nonvar(Xs), nonground(Ys), nonvar(Ys), nonvar(Rest))).
```

Figure 4.2: The instrumented program.

What we would like at this point, is to be able to check dynamically the validity of the `true` assertions from the analyzer. Thanks to the different aspects of the `Ciao` model presented previously, the only thing needed in order to achieve this is to (**1**) *turn the status of the `true` assertions produced by the analyzer into `check`*, as shown in Fig. 4.2. This would normally not make any sense since these `true` assertions have been proved by the analyzer. But that is exactly what we want to check, i.e., whether the information inferred is incorrect. To do this, (**2**) we run the transformed program (Fig. 4.2) again through `CiaoPP` (Fig. 3.1) but *without performing any analysis*. In that case the `check` literals (stemming from the `true` literals of the previous run) will not be simplified in the comparator (since there is no abstract information to compare against) and instead will be converted directly to run-time tests. I.e., the `check(Goal)` literals will be expanded and compiled to code that, every time that this program point is reached, in every execution, will check dynamically if the property (or properties) within the `check` literal (i.e., those in `Goal`) succeed, and an error message will be emitted if they do not. The only missing step to complete the automation of the approach is to (**3**) use the random test case generator to generate a set of test cases for `prepend/3`, and run those test cases. The framework will ensure that instances of the goal `prepend(X,Xs,Ys)` are generated where `Xs` is a list and `Ys` is a free variable, but otherwise `X` and the elements of `Xs` will be instantiated to random terms. In this example, as soon as a test case is generated where both `X` and all elements in `Xs` are ground, the program will report a runtime-checking error in the check in line 12, letting us know that the third program-point assertion, and thus the analysis, is incorrect.[1]

The same procedure can be followed to debug different analyses with different benchmarks. If the execution of any test case reports a runtime-checking error for one assertion, it will mean that the assertion was not correct and the analyzer computed an incorrect over-approximation of the semantics of the program. Alternatively, if this experiment, which can be automated easily, is run for an extensive suite of benchmarks without errors, we can gain more confidence that our analysis implementation is correct, even if perhaps imprecise (although of course we cannot have actual correctness in general by testing).

4.2.2 The Algorithm

In this section we present in more detail the actual algorithm for combining the components of the framework used in order to test the static analyzer.

4.2.2.1 Basic Reasoning Behind the Approach

We start by establishing more concretely the basic reasoning behind the approach in terms of abstract interpretation and safe upper and lower approximations. The mathematical notation in this subsection is purely for readability, as a proper formalization is outside the scope of the paper, and in any case arguably not really necessary, thanks to the simplicity of the approach.

An abstract interpretation-based static analysis computes an over-approximation S_P^+ of the collecting semantics S_P of a program P. Such collecting semantics can be broadly defined as a control flow graph for the program decorated at each node with the set of all possible states that could occur at run-time at that program point. Different approximations of this semantics will have smaller or larger sets of possible states at each program point. Let us denote by $S'_P \subset_P S''_P$ the relation that establishes that an approximation of S_P, S''_P, is an over-approximation of another, S'_P. The analysis will be correct if indeed $S_P \subset_P S_P^+$.

Since S_P is undecidable, this relation cannot be checked in general. However, if we had a good enough under-approximation S_P^- of S_P, it can be tested as $S_P^- \subset_P S_P^+$. If it does not hold and $S_P^- \not\subset_P S_P^+$, then it would imply that $S_P \not\subset_P S_P^+$, and thus, the results of the analysis would be incorrect, i.e., the computed S_P^+ would not actually be an over-approximation of S_P.

An under-approximation of the collecting semantics of P is easy to compute: it suffices with running the program with a subset I^- of the set I of all possible initial states. We denote the resulting under-approximation $S_P^{I^-}$, and note that $S_P = S_P^I$, which would be computable if I is finite and P always terminates. That is the method that we propose for testing the analysis: selecting a large and varied enough I^-, computing $S_P^{I^-}$ and checking that $S_P^{I^-} \subset_P S_P^+$.

A direct implementation of this idea is challenging. It would require tailored instrumentation and monitoring to build and deal with a partially constructed collecting semantic under-approximation as a programming structure, which then would need to be compared to the one the analysis handles. However, as we have seen the process

[1]In the discussion above we have assumed for simplicity that the original program did not already contain check assertions. In that case these need to be treated separately.

Algorithm 1 Analysis Testing Algorithm (for program P and domain D)

```
1: procedure AnaTest($P, D$)
2:     $result \leftarrow$ None
3:     $P_{an} \leftarrow$ analyze and annotate $P$ with domain $D$ (incl. program-point assertions).
4:     $P_{check} \leftarrow P_{an}$ where *true* assertion status is replaced by *check*
5:     $P_{rtcheck} \leftarrow$ instrument $P_{check}$ with *run-time checks*
6:     repeat
7:         Choose an exported predicate $p$ and generate a test case $input$
8:         if $p(input)$ in $P_{check}$ produces runtime errors then
9:             $result \leftarrow$ Error($input$)
10:        else if maximum number of test executions is reached then
11:            $result \leftarrow$ Timeout
12:    until $result \neq$ None return $result$
```

can be greatly simplified by reusing some of the components already in the system, following these observations:

- We can work with one initial state i at a time, following this reasoning: $S_P^{I^-} \subset_P S_P^+ \iff \forall i \in I^-,\ S_P^{\{i\}} \subset_P S_P^+$.

- We can use the random test case generation framework for selecting each initial state i.

- Instead of checking $S_P^{\{i\}} \subset_P S_P^+$, we can instrument the code with *run-time checks* to ensure the execution from initial state i does not contradict the analysis at any point. That is, that the state of the program at any program point is contained in the over-approximation of the set of possible states that the analysis inferred and output as `Ciao` assertions.

4.2.2.2 The Algorithm

We now show the concrete algorithm for implementing our proposal, i.e., the driver that combines and inter-operates the different components of the framework to achieve the desired results. The essence of the algorithm (Alg. 1) is the following: non-deterministically choose a program P and a domain D from a collection of benchmarks and domains, and execute the AnaTest(P, D) procedure until an error is found or a limit is reached. Unless the testing part is ensured to explore the complete execution space, it could in principle be useful to revisit the same (P, D) pair more than once. When the algorithm detects a faulty program-point assertion for some $input$ (Error($input$)), it means that the concrete execution reaches a state not captured by the (over-approximation of the) analysis. In such case it is possible to reconstruct (or store together with the test output) additional information to diagnose the problem. E.g., comparing the concrete execution trace (which is logged during testing) with the analysis graph (recoverable from P_{an}, the program annotated with analysis results), domain operations (inspecting the analysis graph), and transfer functions (from predicates that are *native* to each domain).

4.2.2.3 Other Details and Observations

We now discuss some details and observations on the algorithm that may have been left out or oversimplified in the algorithm sketch:

4.2.2.3.1 Analysis Crashes. An implicit assumption throughout our discussion so far is that the analysis always terminates without errors, but the results computed may be unsound. Of course, it is also possible that a bug in the analysis produces a crash, or even leads to non-termination. It is also possible that the analysis output is malformed (e.g., there are missing assertions in P_{an}). Those errors are of course also checked and reported by our tool. Non-termination is handled with timeouts and possible warnings (both for analyses and concrete executions).

4.2.2.3.2 Benchmark Selection. No prior requirement is imposed on the origin or characteristics of the benchmark suite. It could consist of automatically generated programs, an existing benchmark suite, or just real-life code. Each may have its own advantages and disadvantages (e.g., automatically generated code may test more convoluted or corner cases, but real-life code may find the bugs that actually occur in programs), but in principle, our approach is agnostic in this regard.

4.2.2.3.3 Entry Points. There is no restriction regarding the number of entry points or inputs to a program to be analyzed for. It is common in tools related to ours to use as benchmarks programs with a single entry point with no inputs (e.g., just a single `void main()` function as entry point for C). Our benchmarks are typically `Ciao` modules, and their entry points to analysis and testing are their exported predicates. In `Ciao` programs signatures and types (as well as *entry* assertions) are optional. Admissible inputs (i.e., the initial set of possible states for analysis or test case generation) can be specified by writing assertions for the exported predicates, by means of *entry* assertions, or skipped altogether. Note also that if our benchmarks had the restriction mentioned above (in our case, exporting only a `main/0` predicate), then test case generation would not be needed for our algorithm.

4.2.2.3.4 Test Case Generation. In the absence of *entry* assertions, the test case generation framework [11] has already some mechanisms to generate relevant test cases, instead of random, nonsensical inputs which would exercise few *run-time checks* before failing. However, these generators have limitations, and the assertion-based testing framework is in fact best used with assertions that have descriptive-enough call patterns, or with custom user-defined generators in their absence. To tackle this problem, our tool makes also use of *test* assertions when available in the benchmarks, using also the test cases specified in the benchmarks besides those randomly generated. This can help, e.g., when using a benchmark that works with files and has paths as input, for which relevant test cases would not likely be found with random generation. Note however that the tool would still work without any *entry* or *test* assertions; it would just become less effective.

4.2.2.3.5 Error Diagnosis and Debugging. It is important to note that although error diagnosis and debugging is primarily left for the user to manually perform, our tool facilitates the task in some aspects. Firstly, the *assertion-based testing* tool supports shrinking of failed test cases, so we can expect reasonably small variable substitutions in the errors reported. Note however that benchmark reduction, e.g., by delta debugging [59], is currently not supported. Secondly, as sketched in Algorithm 1, the error location and trace reported by the *runtime-checks* instrumentation provide an approximated idea of the point where the analysis went wrong, if not of the reason why. For example, if the *runtime-check* error points to a *program-point* assertion right after a call to a builtin, then we typically know that the analysis erred in the builtin handler.

4.2.2.3.6 Multivariance and Path-Sensitivity. As presented, our approach might miss some analysis errors even when the right test cases are used, since we have apparently disregarded multi-variance and path-sensitivity. In fact in `CiaoPP` the information inferred is fully multi-variant, and separate path information is kept to each variant. However, in order to produce an output that is easy for the programmer to inspect, i.e., that is close to the source program, when outputting the analysis results `CiaoPP` by default combines the different versions of each predicate (and the associated information) into a single code version and a single combined assertion for each program point and predicate. If this default output is used when implementing our approach, it is indeed entirely possible that the analysis errs at a program point in one path but the algorithm never detects it: this can happen if, for example, in another path leading to the same program point (such that the two paths and their corresponding analysis results are collapsed –lubbed– together at the same program point) the analysis infers a too general value (higher in the domain lattice) at that program point and thus, the error is not detected. However, this potential problem is easily addressed by simply changing the corresponding flag in `CiaoPP` so that the different versions are not collapsed and are instead *materialized* into different predicate instances. This is done in `CiaoPP` by selecting the *versions* transformation prior to emitting the output. In this case multiple versions may be generated for a given predicate, if there are separate paths to it with different abstract information, and the corresponding analysis information will be annotated separately for each abstract path through the program in the program text of the different versions, avoiding the problem mentioned above.

4.2.3 Applications and Examples

In this section we discuss interesting use cases and applications of our approach. As observed before, our testing technique can be seen as a sanity or coherence check, and thus it can be targeted to test different components of the system depending on which ones are assumed to be trusted. Some examples follow. A few of them have actually been implemented and we report on them in the following section. We hope to implement the others in our future work.

4.2.3.1 Debugging Abstract Domains.

The first application of our approach, which has been illustrated in the examples, is to test the abstract domains. In general the `Ciao` abstract interpretation engine (the *fixpoint algorithms* and all the surrounding infrastructure of the system, into which the domains are "plugged-in") includes the components of the analyzer we trust most, since they have been used and refined for more than 30 years. Thus, it makes sense to take this as the trusted base and try to find errors in the domains.

This situation is realistic and frequent, since CiaoPP is at the same time a production and a research tool, and new domains are constantly being developed. In order to test a new domain with the algorithm proposed, two components need to be present. The first one is a translation interface from the abstract values in the domain to Ciao properties, which is needed to express the analysis results as assertions. But note that this is actually already a requirement for any abstract domain that intends to make full use of the framework, so it is normally implemented anyway in all domains. The other component is to have builtin checks for those properties to be used by the *run-time checking* framework, if those properties are declared native and not written in the source language and thus already runnable and checkable. This is also a standard requirement on domains to be able to make full use of the framework, so they are typically also implemented with the domain. In particular, all current Ciao abstract domains include the functionalities mentioned, and can be tested as is with the proposed approach. We show the results for some of them in the case study described in Sec. 4.2.4.

4.2.3.2 Debugging Trust Assertions and Custom Transfer Functions.

One feature of CiaoPP's analyses is that they can be guided by the user, which can feed the analyzer with information that can be assumed to be true at points where otherwise the analysis would lose precision. We have already introduced in Sec. 2.1 one of these mechanisms, *trust* assertions, but there are others. One is custom abstract transfer functions, similar to those that need to be implemented for abstracting each builtin within each domain, but that the user can provide for any predicate. A particular instance of this mechanism is when the user specifies that one predicate is indistinguishable from or should behave like another with respect to a domain: the *equiv* declaration. Our approach can be used to test these mechanisms too. Both to test that they are applied correctly by the analyzer, if the user-provided information is trusted to be correct, and to test that the user-provided information is correct, if what is trusted is that the information is applied correctly. The latter is in particular very useful, since even a completely sound analyzer can produce unsound results if it assumes some property to be true when it is actually not, and thus there will always be the need to test such properties.

4.2.3.3 Testing the Abstract Interpretation Engine.

Another idea that comes to mind is whether we can test the abstract interpretation engine (the *fixpoint algorithms* and all the surrounding infrastructure of the framework) instead of the domains, by using domains that are simple enough to be used as a trusted base. While the classic algorithms are quite stable, new fixpoints are also added to the system (e.g., recently a modular and incremental fixpoint) which can of course bring new bugs. A first abstract domain that could be useful for this purpose is the *concrete domain* itself (which is actually implemented in CiaoPP as the *pd* –partial deduction– domain). If we give the analysis a singleton set of initial states as entry point, the analyzer should behave as an interpreter for the program starting from that initial state, provided the program terminates. The assertions resulting from this "analysis" will use the =/2 property and be essentially a program which is adorned at each program point with the concrete states(s) that the analyzer infers will be occurring at run time, expressed as conjunctions of substitutions using =/2. Then, when running this program, the *run-time checks* would check that the variables are indeed instantiated to the concrete values inferred. Non-deterministic programs could be equally handled with `member/2` ($\in$) instead of `=/2` ($=$). A second domain that could be useful in this context is the *pdb* domain, which can be used to perform *reachability* analysis. The properties appearing in the assertions resulting from this analysis would just be `possibly_reachable/0` ($\top$) and `not_reachable/0` ($\bot$), which indicates if a program point is definitely unreachable at run-time.[2] The *run-time checks* would just report an error any time a check for the property `not_reachable/0` ($\bot$) is invoked at run time. This test would then detect if the analyzer incorrectly marks reachable parts of the program as unreachable.

4.2.3.4 Testing the Overall Consistency of the Framework.

So far we have focused on applications in testing analysis soundness. But doing so has the implicit assumption that there are clear semantics and specifications for the analyzer to follow, and that is not always the case. Sometimes the semantics is underspecified, and then a discrepancy between what the analysis infers and what the program executes is not so much an error but a disparity in the interpretation of such an under-specification. In those cases our tool helps ensure that at least the analysis and run-time semantics are consistent. A relevant example can be found in the case of the abstraction of built-ins within abstract domain implementations. For some of them the specification is not complete (sometimes even the ISO-Prolog standard) and again our tool can at least check for inconsistencies in the interpretations made by the analyses and the run-time system.

In this same line, the tool has helped us find inconsistencies between the understanding of `Ciao` properties in the analysis and in the *runtime-checks* framework. With many properties this cannot happen (e.g., with pure predicates) because both the analysis and the run-time checking derive the semantics from the actual code defining the property. But for more complex properties the implementations may be different, perhaps developed by different people, with different interpretations of the property semantics. An actual example is the property `cardinality/3`, which provides upper and lower bounds to the number of solutions that a predicate might produce. It is a property that has not seen a lot of use (determinacy and/or non-failure are the ones used most frequently), and our experimental evaluation exposed that for `cardinality/3` the analysis was considering only different solutions while the *runtime-checks* framework counted also repeated ones.

4.2.3.5 Integration Testing of the Analyzer and Third Parties.

Finally, even if every piece of the analyzer is validated separately, our tool can still help in testing how all its parts integrate together to form a functional and sound analyzer, and, even more interestingly, it can also test the correctness of the different integrations with external or third party solvers used by the analyzer (e.g., the PPL library).

[2]Note that this, combined with non-failure analysis [18, 7], can also infer `definitely_reachable/0`, but that is a more complex domain.

Abstract Domain	Properties Abstracted	Maturity Level	References
shfr	aliasing, modes	mature	[47]
def	aliasing, modes	intermediate	[20]
gr	aliasing, modes	intermediate	[8]
eterms	types	mature	[56]
etermsvar	types	experimental	[56]
nf	failure	mature	[18, 7]
det	determinism	mature	[40, 41]

Table 4.1: Domains used for the evaluation of the approach.

4.2.4 A More Detailed Case Study

As a case study, in order to validate our approach and confirm its effectiveness, we have studied further the *Debugging Abstract Domains* application of Section 4.2.3, by applying our prototype more systematically to some of the analyses in `CiaoPP`.

4.2.4.1 Setup.

The analyses tested all use the standard configuration of the abstract interpretation framework (i.e., the *PLAI* fixpoint, multi-variance on calls, etc.) but differ in the abstract domains used for the analysis. The complete list of abstract domains tested can be seen in the first column of Table 4.1. The second column indicates the different properties which the domains reason about, such as variable aliasing, variable modes, variable types, (non)failure, or determinism. The domains range in maturity, from stable domains like *shfr* and *eterms*, to mere prototypes like *etermsvar*. The third column of the figure indicates this level of maturity with three different values: *mature*, *intermediate*, *experimental*. For more details about the domains we refer to the citations in the fourth column.

The experiment has been run over some selected benchmarks with increasing levels of complexity and language features. We have started with simple, existing `CiaoPP` benchmarks used for, e.g., demos, statistics and integration testing, for which in principle the analyses tested should be correct. Then we have continued with a large database of anonymized solutions for Prolog assignments in undergraduate courses, which on one hand are not expected to use necessarily the most sophisticated features of the language (although there are always exceptions), but on the other hand are known to exhibit a high degree of creativity in combining language elements in unusual and unpredictable ways, including many that do not make sense at all. The intuition is that these combinations may exercise corner cases of the analyses in a similar (but hopefully somewhat more focused way) than random program generation. Finally, we have applied the experiment to some selected modules of the `Ciao` code base using more advanced features. Additionally, we have cherry-picked some benchmarks which were expected to reveal some known bugs, either still unfixed or explicitly reintroduced in the system for this experiment, and some using deliberately features not supported by a particular analysis such as, e.g., attributed variables. Some of the benchmarks have been modified by adding *entry* assertions to guide test case generation, and existing test cases from unit tests (i.e., *test* assertions) have been used in modules where using random test cases is ineffective or just plain dangerous (e.g., predicates that have files as input). The experiments were run with

`Ciao/CiaoPP` version 1.19-221.

4.2.4.2 Results.

While we are planning on performing a larger set of experiments, [3] the results so far are promising and have allowed us to draw some interesting conclusions and observations. A good number of bugs and inconsistencies were indeed found using the technique, many of them known but also some new ones. First, our experiment was successful in finding known bugs in previous versions of the analyses, that have now been fixed, and also in revealing known limitations of different analyses for some language features. For example, the fact that some of the aliasing domains do not support rational terms was easily detected, and also that many domains do not support attributed variables. Some new, but still not unexpected bugs were found in one of the most experimental domains (*etermsvar*). Furthermore, also a few new bugs were found even in mature domains. These are typically related to the handling of rarely-used built-ins, which explains why they have gone unnoticed, but they are still bugs and have been (or are being) fixed. In addition, while the testing process was aimed at the domains, it also uncovered some bugs in related components of the `Ciao` assertion framework and their integration, which have been fixed too. We thus conclude that our approach is indeed effective in revealing and discovering bugs and inconsistencies in the domains and also in the overall framework.

Another overall conclusion from the experiment is that benchmark selection is very important when focusing our approach on testing specific domains. No bugs were found for the most mature domains using standard benchmarks and the undergraduate Prolog assignments. The subtle bugs mentioned before in less-used built-ins were found instead when using benchmarks extracted from `Ciao`'s code base, i.e., in complex, system code. On the other hand, a good number of errors were found in the experimental domain with even the simpler benchmarks. In fact, in this case, the many errors triggered obfuscated sometimes the real (possibly multiple) origin of the problems, but this is to be expected in immature code: consider for example that just the ISO-standard contains a very large set of built-ins and the implementation of an experimental domain typically does not support all of them.

Finally, it is important to point out that we also found out that there are some bugs that are unlikely to be found with benchmarks like the ones used in the tests, because they are bugs that will probably never occur in realistic programs. One example is the simple bug found in [11] for the handler of the builtin `=/2` in the *sharing-freeness* domain. The code did not consider that the two arguments could be the same variable, and thus the analysis failed for any program with the literal `X=X`. Since that literal always succeeds and is redundant in every program, it will likely not appear in any reasonable benchmark and this error would not be detected by our tool. To find bugs of this kind with our approach, randomly generated benchmarks would be needed.

[3]We are working on including the technique as part of the `Ciao` continuous integration infrastructure, and plan to report on a larger number of `CiaoPP` analyses over a wider range of programs.

Chapter 5

Related Work

Random testing has been used for a long time in Software Engineering [23]. As mentioned before, the idea of using properties and assertions as test case generators was proposed in the context of the Ciao model [5, 28, 50] for logic programs, although it had not really been exploited significantly until this work. QuickCheck [14] provided the first full 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 [31]. It uses a domain-specific language of testable specifications and generates test data based on Haskell types. ErlangQuickCheck and PropEr [48] 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 [13] and CurryCheck [24] 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., [22, 21] 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* [45], 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) [25]. Schrijvers proposed Tor [53] as a mechanism for supporting the execution of predicates using alternative search rules, similar in spirit to Ciao's implementation of search-strategies via packages.

The need for validating program analyzers was discussed by [10], and the topic has motivated interesting research over the past years. On the formal verification side, there have been some pen-and-paper proofs, such as that of the Astree analyzer [15], some automatic and interactive proofs, such as [19, 54], and some verification efforts, which include [3, 38, 33]. Testing efforts for program analyzers include e.g., static analyzers [57, 60, 16, 35], symbolic execution engines [34], refactoring engines [17], compilers [58, 36, 55, 37, 52, 39], SMT solvers [4], among others. Most of these testing approaches use programs in the target language as test cases and and apply testing techniques like fuzzing (e.g., [58, 34, 4]) or differential testing [42], (e.g., [58, 36, 34, 4, 35]). In [9] and [44] abstract domain properties are tested, the latter using QuickCheck [14]. Among the different approaches mentioned, the closest to ours are those that cross-check dynamically observed and statically inferred properties [57, 60, 16, 2].

In [57] the actual pointer aliasing in concrete executions is cross-checked with the pointer aliasing inferred by an aliasing analyzer. Compared to us, they require significant tailored instrumentation which cannot be reused for testing other analyses. However, their approach is agnostic to the (C) aliasing analyzer.

Another cross-check is done in [60] for C model checkers and the *reachability* property, but they obtain the assertions dynamically, and check them statically, complementarily to our approach. Unlike us, they again need tailored instrumentation that cannot be reused to test other analyses, and their benchmarks must be deterministic and with no input, the latter limiting the power of the approach as a testing tool. However, their approach is agnostic to the (C) model checker.

In [16] a wide range of static analysis tests are performed over randomly generated programs. Among others, they check dynamically, at the end of the program, one assertion inferred statically, and they perform the sanity check of ensuring that the analyzer behaves as an interpreter when run from a singleton set of initial states.

Chapter 6

Conclusions

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.

Building on this tool, we have proposed a simple, automatic method for testing abstract interpretation-based static analyzers based on checking that the properties inferred statically are satisfied dynamically. We have leveraged the Ciao unified assertion language and framework, and have constructed a prototype implementation of our method with little effort by combining components already present in the framework: the static analyzer, the runtime-checker, the random test-case generator, and the unit-tester. We just wrote a very reduced amount of glue code that pilots the combination and interplay of the intervening components. We have applied our prototype to a good number of the abstract interpretation-based analyses in CiaoPP, which represent different levels of code maturity. The results are encouraging and show that our tool can effectively discover and locate interesting, unexpected, non-trivial, previously undetected bugs.

Bibliography

[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] Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. Systematic approaches for increasing soundness and precision of static analyzers. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2017, page 31–36, New York, NY, USA, 2017. Association for Computing Machinery.

[3] Sandrine Blazy, Vincent Laporte, André Maroneze, and David Pichardie. Formal verification of a C value analysis based on abstract interpretation. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis*, pages 324–344, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[4] Robert Brummayer and Armin Biere. Fuzzing and delta-debugging SMT solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, SMT '09, page 1–5, New York, NY, USA, 2009. Association for Computing Machinery.

[5] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. V. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd Int'l. WS on Automated Debugging–AADEBUG*, pages 155–170. U. Linköping Press, May 1997.

[6] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. V. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd Int'l. Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.

[7] F. Bueno, P. Lopez-Garcia, and M. V. Hermenegildo. Multivariant Non-Failure Analysis via Standard Abstract Interpretation. In *7th Int'l. Symposium on Functional and Logic Programming*, volume 2998 of *LNCS*, pages 100–116. Springer-Verlag, April 2004.

[8] F. Bueno, P. Lopez-Garcia, G. Puebla, and M. V. Hermenegildo. A Tutorial on Program Development and Optimization using the Ciao Preprocessor. Technical Report CLIP2/06, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, January 2006.

[9] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. Automatically testing implementations of numerical abstract domains. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 768–778, New York, NY, USA, 2018. Association for Computing Machinery.

[10] Cristian Cadar and Alastair Donaldson. Analysing the program analyser. In *International Conference on Software Engineering, Visions of 2025 and Beyond Track (ICSE V2025)*, pages 765–768, 5 2016.

[11] I. Casso, J. F. Morales, P. Lopez-Garcia, and M. V. Hermenegildo. An Integrated Approach to Assertion-Based Random Testing in Prolog. In Maurizio Gabbrielli, editor, *Post-Proceedings of the 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'19)*, volume 12042 of *LNCS*, pages 159–176. Springer-Verlag, April 2020.

[12] Ignacio Casso, José F. Morales, Pedro López-García, and Manuel V. Hermenegildo. Testing Your (Static Analysis) Truths. In Maribel Fernández, editor, *Logic-Based Program Synthesis and Transformation - 30th International Symposium, LOPSTR 2020, Bologna, Italy, September 7-9, 2020, Post-Proceedings*, volume 12561 of *Lecture Notes in Computer Science*, pages 271–292. Springer, 2021.

[13] 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.

[14] 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.

[15] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. *Lecture Notes in Computer Science*, 3444:21–30, September 2005. 14th European Symposium on Programming, ESOP 2005, held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005 ; Conference date: 04-04-2005 Through 08-04-2005.

[16] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. Testing static analyzers with randomly generated programs. In Alwyn E. Goodloe and Suzette Person, editors, *NASA Formal Methods*, pages 120–125, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[17] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, page 185–194, New York, NY, USA, 2007. Association for Computing Machinery.

[18] S.K. Debray, P. Lopez-Garcia, and M. V. Hermenegildo. Non-Failure Analysis for Logic Programs. In *1997 International Conference on Logic Programming*, pages 48–62, Cambridge, MA, June 1997. MIT Press, Cambridge, MA.

[19] Catherine Dubois. Proving ML Type Soundness within Coq. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics*, pages 126–144, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[20] M. García de la Banda, M. V. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Trans. on Programming Languages and Systems*, 18(5):564–615, 1996.

[21] 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.

[22] 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.

[23] Dick Hamlet. Random Testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, page 970–978. Wiley, 1994.

[24] 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.

[25] 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.

[26] 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.

[27] 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. *Theory and Practice of Logic Programming*, 12(1–2):219–252, January 2012.

[28] 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.

[29] M. V. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

[30] 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.

[31] 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.

[32] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *North American Conference on Logic Programming*, 1989.

[33] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A Formally-Verified C Static Analyzer. *SIGPLAN Not.*, 50(1):247–259, January 2015.

[34] Timotej Kapus and Cristian Cadar. Automatic testing of symbolic execution engines via program generation and differential testing. In *IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*, pages 590–600, 11 2017.

[35] Christian Klinger, Maria Christakis, and Valentin Wüstholz. Differentially testing soundness and precision of program analyzers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 239–250, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 216–226, New York, NY, USA, 2014. Association for Computing Machinery.

[37] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, page 386–399, New York, NY, USA, 2015. Association for Computing Machinery.

[38] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[39] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 65–76, New York, NY, USA, 2015. Association for Computing Machinery.

[40] P. Lopez-Garcia, F. Bueno, and M. V. Hermenegildo. Determinacy Analysis for Logic Programs Using Mode and Type Information. In *Proceedings of the 14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, number 3573 in LNCS, pages 19–35. Springer-Verlag, August 2005.

[41] P. Lopez-Garcia, F. Bueno, and M. V. Hermenegildo. Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Analyses. *New Generation Computing*, 28(2):117–206, 2010.

[42] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10:100–107, 1998.

[43] 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.

[44] Jan Midtgaard and Anders Møller. QuickChecking Static Analysis Properties. *Softw. Test., Verif. Reliab.*, 27(6), 2017.

[45] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.

[46] 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.

[47] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *International Conference on Logic Programming (ICLP 1991)*, pages 49–63. MIT Press, June 1991.

[48] 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.

[49] G. Puebla, F. Bueno, and M. V. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. V. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[50] 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.

[51] G. Puebla, F. Bueno, and M. V. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag, March 2000.

[52] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 335–346, New York, NY, USA, 2012. Association for Computing Machinery.

[53] Tom Schrijvers, Bart Demoen, Markus Triska, and Benoit Desouter. Tor: Modular search with hookable disjunction. *Sci. Comput. Program.*, 84:101–120, 2014.

[54] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. *SIGPLAN Not.*, 37(1):217–232, January 2002.

[55] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, page 849–863, New York, NY, USA, 2016. Association for Computing Machinery.

[56] C. Vaucheret and F. Bueno. More Precise yet Efficient Type Inference for Logic Programs. In *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 102–116. Springer-Verlag, September 2002.

[57] Jingyue Wu, Gang Hu, Yang Tang, and Junfeng Yang. Effective dynamic detection of alias analysis errors. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 279–289, New York, NY, USA, 2013. Association for Computing Machinery.

[58] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery.

[59] Andreas Zeller. Yesterday, my program worked. today, it does not. why? *SIGSOFT Softw. Eng. Notes*, 24(6):253–267, October 1999.

[60] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. Finding and understanding bugs in software model checkers. In *Proceedings of the 13th Joint Meeting of the 18th European Software Engineering Conference and the 27th Symposium on the Foundations of Software Engineering*, pages 763–773, 2019.