Testing Your (Static Analysis) Truths [*]

Ignacio Casso[1] [0000-0001-9196-7951], José F. Morales[1] [0000-0001-6098-3895], P. López-García[1,3] [0000-0002-1092-2071], and Manuel V. Hermenegildo[1,2] [0000-0002-7583-323X]

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

Abstract. 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. In this paper, we propose a simple, automatic method for testing abstract interpretation-based static analyzers. Broadly, it consists in checking, over a suite of benchmarks, that the properties inferred statically are satisfied dynamically. The main advantage of our approach is its simplicity, which stems directly from framing it within the `Ciao` assertion-based validation framework, and its blended static/dynamic assertion checking approach. We show that in this setting, the analysis can be tested with little effort by combining the following components already present in the framework: the *static analyzer*, the assertion *run-time checking* mechanism, the *random test case generator*, and the *unit-test framework*. Together they compose a tool that can effectively discover and locate errors in the different components of the analysis framework. We apply our approach to test some of `CiaoPP`'s analysis domains over a wide range of programs, successfully finding non-trivial, previously undetected bugs, with a low degree of effort.

Keywords: Static Analysis, Run-time Checks, Random Testing, Assertions, Abstract Interpretation, Program Analysis, (Constraint) Logic Programming.

1 Introduction and Motivation

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

[*] Partially funded by MICINN PID2019-108528RB-C21 *ProCode* and the Madrid P2018/TCS-4339 *BLOQUES-CM* program.

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 paper is to develop a simple, automatic method for testing abstract interpretation-based static analyzers. Although the approach is general, we develop it for concreteness in the context of the Ciao [21] logic programming-based, multiparadigm language. The Ciao programming environment includes an abstract interpretation-based static analyzer, CiaoPP, 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. A recent, shallow effort of applying a new testing tool to some parts of the Ciao analyzers as a case study [10] revealed subtle bugs, not only in the less-developed parts of the system, but also in corner cases of the parts that are considered more mature, such as, e.g., in the handling of rarely-used built-ins.

Another feature of Ciao that will be instrumental to our approach is the use of a unified assertion language and framework across its different components [22, 23], which together implement its unique blend of static and dynamic assertion checking. These components include: 1) the PLAI *static analyzer* [40, 25, 19], which expresses the inferred information as Ciao assertions interspersed within the original program; 2) the assertion *runtime-checking framework* [45, 46], which instruments the code to ensure that any assertions remaining after static verification are not violated at run time; 3) the *(random) test case* generation framework [10], which generates random test cases satisfying the properties present in an assertion preconditions; 4) the *unit-test framework* [36], which executes those test cases.

In this paper, we propose an algorithm that combines these four basic components 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, [49] 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.

The rest of the paper is structured as follows. Section 2 gives background knowledge needed to describe the main ideas and contributions of this paper. In particular, we recall some relevant aspects of the CiaoPP unified assertion framework. Then, Section 3 gives an overview of our approach illustrating it with an example. Section 4 presents our concrete algorithm to combine the different elements of the framework for the task of testing the static analyzer. In Section 5 we show some examples and applications of our approach. In Section 6 we apply the idea to testing the analysis results for a wide range of CiaoPP's abstract domains and properties. Finally, Section 7 discusses related work and Section 8 summarizes our conclusions and plans for future work.

2 Preliminaries

In this section we review in some 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 [4, 22, 42, 24, 36, 21] and their references.

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, so that they can be used as run-time checks, and which, for our purposes, are typically *native* to CiaoPP, i.e., abstracted and inferred by some domain in CiaoPP. This includes a wide range of properties, from types, modes and variable sharing, to determinism, (non)failure and resource consumption. We refer the reader to [41, 24, 21] 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. 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. 1).

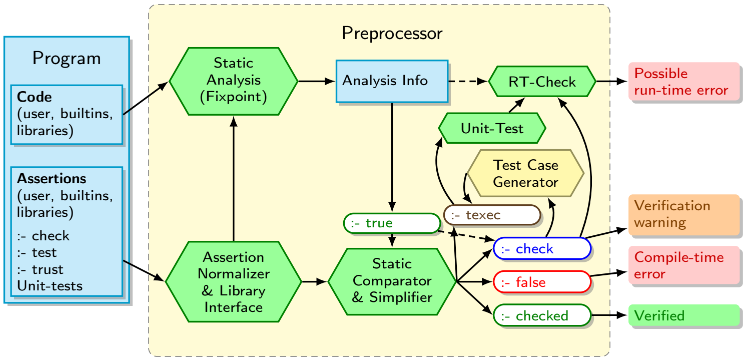

Fig. 1. The Ciao assertion framework (CiaoPP's verification/testing architecture).

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). But more importantly for this paper, assertions are 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 technical and theoretical details of how this is achieved are omitted for space constraints. 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, "pluggable" 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.

Run-time Checking. Static analysis can be used for compile-time checking of assertions (the Static Comparator & Simplifier, in Fig. 1) but the inherent imprecision of the analysis can lead to some assertions, specially those with user-defined properties that are not native to abstract domains, to not be proved or disproved statically (although perhaps they are simplified). In those cases, 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. This results in a very rich set of properties being checkable in `Ciao`, including types, modes, variable sharing, failure, exceptions, determinism, choice-points, resources, and more, blending smoothly static and dynamic techniques.

Unit Tests, Test Case Generation, and Assertion-based Testing. Test inputs can be provided by the user, by means of **`test`** assertions (unit tests), and used to test the test assertion itself as well as, through the *runtime-checking* mechanism, also any other assertion in any predicate called by the test case, that was not eliminated in the static checking. The unit-testing framework in principle requires the user to manually write individual test cases for each assertion to be tested. However, the `Ciao` model also includes mechanisms for generating test cases automatically from the assertion preconditions, using the corresponding property *predicates* as generators. This has been extended recently [10] to a *random* test case generation framework, which automatically generates, using the same technique, *random* test cases that satisfy assertion preconditions. We refer to the combination of this test generation mechanism with the run-time checking of the intervening assertions as *assertion-based testing*, that is, generating and running relevant test cases which exercise the *run-time checks* of the assertions in a program, thus testing if those assertions are correct. This yields similar results to *property-based testing* [11] but in a more integrated way within the overall model. Such automatic generation is supported for native properties, *regular types*, and user-defined properties as long as they are restricted to pure Prolog with arithmetic or mode and sharing constraints. In particular, it is always supported for the *native* properties used by the different analyses in the assertions that they output.

3 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

```
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))).
```

Fig. 2. An incorrect simple mode analysis.

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 the previous section, as a new source file with interspersed assertions, as shown in Fig. 2 (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.

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. 3. 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. 3) again through `CiaoPP` (Fig. 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

```
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))).
```

Fig. 3. The instrumented program.

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.[4]

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 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.

4.1 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

[4] 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 and there are several options, including simply ignoring them for the process or actually turning them into `trusts`, so that we switch roles and trust the user-provided properties while checking the analyzer-inferred ones. This very interesting issue of when and whether to use the user-provided assertions to be checked during analysis, and its relation to run-time checking is discussed in depth in [18].

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 $S^{I^-}_P$, and note that $S_P = S^I_P$, 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 $S^{I^-}_P$ and checking that $S^{I^-}_P \subset_P S^+_P$.

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 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: $S^{I^-}_P \subset_P S^+_P \iff \forall i \in I^-,\ S^{\{i\}}_P \subset_P S^+_P$.
- We can use the random test case generation framework for selecting each initial state i.
- Instead of checking $S^{\{i\}}_P \subset_P S^+_P$, 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 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 $\textsc{AnaTest}(P, D)$ 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* ($\textsc{Error}(\textit{input})$), it means that the concrete execution reaches a state not captured by the (over-approximation

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

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

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).

4.3 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:

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).

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.

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.

Test Case Generation. In the absence of *entry* assertions, the test case generation framework [10] 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.

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 [51], 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.

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.

5 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.

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. 6.

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 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.

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.[5] 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.

[5] Note that this, combined with non-failure analysis [15, 5], can also infer definitely_reachable/0, but that is a more complex domain.

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.

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).

6 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 5, by applying our prototype more systematically to some of the analyses in `CiaoPP`.

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 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

Abstract Domain	Properties Abstracted	Maturity Level	References
shfr	aliasing, modes	mature	[39]
def	aliasing, modes	intermediate	[20]
gr	aliasing, modes	intermediate	[6]
eterms	types	mature	[48]
etermsvar	types	experimental	[48]
nf	failure	mature	[15, 5]
det	determinism	mature	[33, 34]

Table 1. Domains used for the evaluation of the approach.

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.

Results. While we are planning on performing a larger set of experiments, [6] 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

[6] 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.

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 [10] 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.

7 Related Work

The need for validating program analyzers was discussed by [8], 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 [12], some automatic and interactive proofs, such as [16, 44], and some verification efforts, which include [2, 31, 26]. Testing efforts for program analyzers include e.g., static analyzers [49, 52, 13, 28], symbolic execution engines [27], refactoring engines [14], compilers [50, 29, 47, 30, 43, 32], SMT solvers [3], 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., [50, 27, 3]) or differential testing [35], (e.g., [50, 29, 27, 3, 28]). In [7] and [38] abstract domain properties are tested, the latter using QuickCheck [11]. Among the different approaches mentioned, the closest to ours are those that cross-check dynamically observed and statically inferred properties [49, 52, 13, 1].

In [49] 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 [52] 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 [13] 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.

8 Conclusions and Future Work

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, not only old errors in previous versions (that are obviously less interesting since they were fixed in newer versions), but also new, interesting and unexpected, non-trivial, previously undetected bugs.

Our technique can also be applied to testing the correctness of the analyzers for many other types of properties that were not discussed for brevity, such as the *computational* properties inferred by CiaoPP. These include, e.g., determinacy, non-failure, upper and lower bounds on costs and complexity, or accumulated costs and profiling, and the required run-time checking support exists for many of them (see, e.g., [37]). Of course some properties cannot be checked fully (e.g., termination, beyond just checking for timeouts). There are also many other interesting sanity checks enabled by `Ciao`'s integrated and unified assertion language and framework which we have left as future work, such as testing the assertion simplifier, which simplifies programs discarding (parts of) `check` assertions that have been proven statically. This could be done by analyzing a benchmark without assertions, simplifying the assertions output, and checking that there are no assertions left. We also plan to use the test case generation framework to do differential testing of several program optimizations and transformations over a suite of benchmarks, by just checking that they produce the same outputs for the same randomly generated inputs. A recent paper [9] suggested defining and using distances in abstract domains and between abstract semantics (i.e., between abstract AND-OR trees inferred by the analyzer). We plan to implement an instrumentation that uses such distances to test analysis precision and measure coverage within our approach: if the distance between the dynamic under-approximation and the static over-approximation of the program semantics is small, it means that the analysis was precise and the random inputs had good coverage; otherwise, either the analysis was imprecise, or the test case generation had poor coverage. We plan to investigate heuristics to distinguish both cases. Another interesting avenue for future research is to combine our approach with more directed testing techniques, such as, e.g., concolic testing [17].

References

1. Andreasen, E.S., Møller, A., Nielsen, B.B.: 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. p. 31–36. SOAP 2017, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3088515.3088521, https://doi.org/10.1145/3088515.3088521 17
2. Blazy, S., Laporte, V., Maroneze, A., Pichardie, D.: Formal verification of a C value analysis based on abstract interpretation. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis. pp. 324–344. Springer Berlin Heidelberg, Berlin, Heidelberg (2013) 16
3. Brummayer, R., Biere, A.: Fuzzing and delta-debugging SMT solvers. In: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories. p. 1–5. SMT '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1670412.1670413, https://doi.org/10.1145/1670412.1670413 16, 17
4. Bueno, F., Deransart, P., Drabent, W., Ferrand, G., Hermenegildo, M.V., Maluszynski, J., Puebla, G.: 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. pp. 155–170. U. of Linköping Press, Linköping, Sweden (May 1997), ftp://cliplab.org/pub/papers/aadebug_discipldeliv.ps.gz 4
5. Bueno, F., Lopez-Garcia, P., Hermenegildo, M.V.: Multivariant Non-Failure Analysis via Standard Abstract Interpretation. In: 7th Int'l. Symposium on Functional and Logic Programming. LNCS, vol. 2998, pp. 100–116. Springer-Verlag (April 2004) 13, 15

6. Bueno, F., Lopez-Garcia, P., Puebla, G., Hermenegildo, M.V.: A Tutorial on Program Development and Optimization using the Ciao Preprocessor. Tech. Rep. CLIP2/06, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain (January 2006) 15
7. Bugariu, A., Wüstholz, V., Christakis, M., Müller, P.: Automatically testing implementations of numerical abstract domains. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. p. 768–778. ASE 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3238147.3240464, https://doi.org/10.1145/3238147.3240464 17
8. Cadar, C., Donaldson, A.: Analysing the program analyser. In: International Conference on Software Engineering, Visions of 2025 and Beyond Track (ICSE V2025). pp. 765–768 (5 2016) 16
9. Casso, I., Morales, J.F., Lopez-Garcia, P., Giacobazzi, R., Hermenegildo, M.V.: Computing Abstract Distances in Logic Programs. In: Gabbrielli, M. (ed.) Post-Proceedings of the 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'19). LNCS, vol. 12042. Springer-Verlag (April 2020). https://doi.org/10.1007/978-3-030-45260-5_4 18
10. Casso, I., Morales, J.F., Lopez-Garcia, P., Hermenegildo, M.V.: An Integrated Approach to Assertion-Based Random Testing in Prolog. In: Gabbrielli, M. (ed.) Post-Proceedings of the 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'19). LNCS, vol. 12042, pp. 159–176. Springer-Verlag (April 2020). https://doi.org/10.1007/978-3-030-45260-5_10 2, 6, 11, 16
11. Claessen, K., Hughes, J.: QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In: Fifth ACM SIGPLAN Int'l. Conf. on Functional Programming. pp. 268–279. ICFP'00, ACM (2000) 6, 17
12. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyzer. Lecture Notes in Computer Science **3444**, 21–30 (Sep 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
13. Cuoq, P., Monate, B., Pacalet, A., Prevosto, V., Regehr, J., Yakobowski, B., Yang, X.: Testing static analyzers with randomly generated programs. In: Goodloe, A.E., Person, S. (eds.) NASA Formal Methods. pp. 120–125. Springer Berlin Heidelberg, Berlin, Heidelberg (2012) 16, 17
14. Daniel, B., Dig, D., Garcia, K., Marinov, D.: 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. p. 185–194. ESEC-FSE '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1287624.1287651, https://doi.org/10.1145/1287624.1287651 16
15. Debray, S., Lopez-Garcia, P., Hermenegildo, M.V.: Non-Failure Analysis for Logic Programs. In: 1997 International Conference on Logic Programming. pp. 48–62. MIT Press, Cambridge, MA, Cambridge, MA (June 1997) 13, 15
16. Dubois, C.: Proving ML Type Soundness within Coq. In: Aagaard, M., Harrison, J. (eds.) Theorem Proving in Higher Order Logics. pp. 126–144. Springer Berlin Heidelberg, Berlin, Heidelberg (2000) 16
17. Fortz, S., Mesnard, F., Payet, É., Perrouin, G., Vanhoof, W., Vidal, G.: An SMT-Based Concolic Testing Tool for Logic Programs. In: Functional and Logic Programming - 15th International Symposium, FLOPS. Lecture Notes in Computer Science, vol. 12073, pp. 215–219. Springer (September 2020). https://doi.org/10.1007/978-3-030-59025-3_13 18

18. Garcia-Contreras, I., Morales, J., Hermenegildo, M.V.: Multivariant Assertion-based Guidance in Abstract Interpretation. In: Post-Proceedings of the 28th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'18). pp. 184–201. No. 11408 in LNCS, Springer-Verlag (January 2019). https://doi.org/10.1007/978-3-030-13838-7_11 8
19. Garcia-Contreras, I., Morales, J., Hermenegildo, M.V.: Incremental Analysis of Logic Programs with Assertions and Open Predicates. In: Proceedings of the 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'19). pp. 36–56. LNCS, Springer-Verlag (2020). https://doi.org/10.1007/978-3-030-45260-5_3 2
20. García de la Banda, M., Hermenegildo, M.V., Bruynooghe, M., Dumortier, V., Janssens, G., Simoens, W.: Global Analysis of Constraint Logic Programs. ACM Trans. on Programming Languages and Systems **18**(5), 564–615 (1996) 15
21. Hermenegildo, M.V., Bueno, F., Carro, M., Lopez-Garcia, P., Mera, E., Morales, J., Puebla, G.: An Overview of Ciao and its Design Philosophy. Theory and Practice of Logic Programming **12**(1–2), 219–252 (January 2012). https://doi.org/doi:10.1017/S1471068411000457, http://arxiv.org/abs/1102.5497 2, 4
22. Hermenegildo, M.V., Puebla, G., Bueno, F.: Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In: Apt, K.R., Marek, V., Truszczynski, M., Warren, D.S. (eds.) The Logic Programming Paradigm: a 25–Year Perspective, pp. 161–192. Springer-Verlag (July 1999) 2, 4
23. Hermenegildo, M.V., Puebla, G., Bueno, F., Lopez-Garcia, P.: Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In: 10th International Static Analysis Symposium (SAS'03). pp. 127–152. No. 2694 in LNCS, Springer-Verlag (June 2003) 2
24. Hermenegildo, M.V., Puebla, G., Bueno, F., Lopez-Garcia, P.: 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). https://doi.org/10.1016/j.scico.2005.02.006 4
25. Hermenegildo, M.V., Puebla, G., Marriott, K., Stuckey, P.: Incremental Analysis of Constraint Logic Programs. ACM Transactions on Programming Languages and Systems **22**(2), 187–223 (March 2000) 2
26. Jourdan, J.H., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A Formally-Verified C Static Analyzer. SIGPLAN Not. **50**(1), 247–259 (Jan 2015). https://doi.org/10.1145/2775051.2676966, https://doi.org/10.1145/2775051.2676966 16
27. Kapus, T., Cadar, C.: Automatic testing of symbolic execution engines via program generation and differential testing. In: IEEE/ACM International Conference on Automated Software Engineering (ASE 2017). pp. 590–600 (11 2017) 16, 17
28. Klinger, C., Christakis, M., Wüstholz, V.: Differentially testing soundness and precision of program analyzers. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 239–250. ISSTA 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3293882.3330553, https://doi.org/10.1145/3293882.3330553 16, 17
29. Le, V., Afshari, M., Su, Z.: Compiler validation via equivalence modulo inputs. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 216–226. PLDI '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2594291.2594334, https://doi.org/10.1145/2594291.2594334 16, 17

30. Le, V., Sun, C., Su, Z.: 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. p. 386–399. OOPSLA 2015, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2814270.2814319, https://doi.org/10.1145/2814270.2814319 16
31. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (Jul 2009). https://doi.org/10.1145/1538788.1538814, https://doi.org/10.1145/1538788.1538814 16
32. Lidbury, C., Lascu, A., Chong, N., Donaldson, A.F.: Many-core compiler fuzzing. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 65–76. PLDI '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2737924.2737986, https://doi.org/10.1145/2737924.2737986 16
33. Lopez-Garcia, P., Bueno, F., Hermenegildo, M.V.: 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). pp. 19–35. No. 3573 in LNCS, Springer-Verlag (August 2005) 15
34. Lopez-Garcia, P., Bueno, F., Hermenegildo, M.V.: Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Analyses. New Generation Computing **28**(2), 117–206 (2010) 15
35. McKeeman, W.M.: Differential testing for software. Digital Technical Journal **10**, 100–107 (1998) 17
36. Mera, E., Lopez-Garcia, P., Hermenegildo, M.V.: Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In: 25th Int'l. Conference on Logic Programming (ICLP'09). LNCS, vol. 5649, pp. 281–295. Springer-Verlag (July 2009) 2, 4
37. Mera, E., Trigo, T., Lopez-Garcia, P., Hermenegildo, M.V.: Profiling for Run-Time Checking of Computational Properties and Performance Debugging in Logic Programs. In: Practical Aspects of Declarative Languages (PADL'11). Lecture Notes in Computer Science, vol. 6539, pp. 38–53. Springer-Verlag (January 2011) 17
38. Midtgaard, J., Møller, A.: QuickChecking Static Analysis Properties. Softw. Test., Verif. Reliab. **27**(6) (2017). https://doi.org/10.1002/stvr.1640, https://doi.org/10.1002/stvr.1640 17
39. Muthukumar, K., Hermenegildo, M.: Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In: International Conference on Logic Programming (ICLP 1991). pp. 49–63. MIT Press (June 1991) 15
40. Muthukumar, K., Hermenegildo, M.: Compile-time Derivation of Variable Dependency Using Abstract Interpretation. Journal of Logic Programming **13**(2/3), 315–347 (July 1992) 2
41. Puebla, G., Bueno, F., Hermenegildo, M.V.: An Assertion Language for Constraint Logic Programs. In: Deransart, P., Hermenegildo, M.V., Maluszynski, J. (eds.) Analysis and Visualization Tools for Constraint Programming, pp. 23–61. No. 1870 in LNCS, Springer-Verlag (September 2000) 4
42. Puebla, G., Bueno, F., Hermenegildo, M.V.: Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In: Logic-based Program Synthesis and Transformation (LOPSTR'99). pp. 273–292. No. 1817 in LNCS, Springer-Verlag (March 2000) 4

43. Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., Yang, X.: Test-case reduction for C compiler bugs. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 335–346. PLDI '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2254064.2254104, https://doi.org/10.1145/2254064.2254104 16
44. Shao, Z., Saha, B., Trifonov, V., Papaspyrou, N.: A type system for certified binaries. SIGPLAN Not. **37**(1), 217–232 (Jan 2002). https://doi.org/10.1145/565816.503293, https://doi.org/10.1145/565816.503293 16
45. Stulova, N., Morales, J.F., Hermenegildo, M.V.: Practical Run-time Checking via Unobtrusive Property Caching. Theory and Practice of Logic Programming, 31st Int'l. Conference on Logic Programming (ICLP'15) Special Issue **15**(04-05), 726–741 (September 2015). https://doi.org/10.1017/S1471068415000344, http://arxiv.org/abs/1507.05986 2
46. Stulova, N., Morales, J.F., Hermenegildo, M.V.: Reducing the Overhead of Assertion Run-time Checks via Static Analysis. In: 18th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'16). pp. 90–103. ACM Press (September 2016) 2
47. Sun, C., Le, V., Su, Z.: Finding compiler bugs via live code mutation. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. p. 849–863. OOPSLA 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2983990.2984038, https://doi.org/10.1145/2983990.2984038 16
48. Vaucheret, C., Bueno, F.: More Precise yet Efficient Type Inference for Logic Programs. In: 9th International Static Analysis Symposium (SAS'02). Lecture Notes in Computer Science, vol. 2477, pp. 102–116. Springer-Verlag (September 2002) 15
49. Wu, J., Hu, G., Tang, Y., Yang, J.: Effective dynamic detection of alias analysis errors. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. p. 279–289. ESEC/FSE 2013, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2491411.2491439, https://doi.org/10.1145/2491411.2491439 3, 16, 17
50. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 283–294. PLDI '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1993498.1993532, https://doi.org/10.1145/1993498.1993532 16, 17
51. Zeller, A.: Yesterday, my program worked. today, it does not. why? SIGSOFT Softw. Eng. Notes **24**(6), 253–267 (Oct 1999). https://doi.org/10.1145/318774.318946, https://doi.org/10.1145/318774.318946 11
52. Zhang, C., Su, T., Yan, Y., Zhang, F., Pu, G., Su, Z.: 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. pp. 763–773 (2019). https://doi.org/10.1145/3338906.3338932 16, 17