doi:10.1017/S1471068425100069

Checkification: A Practical Approach for Testing Static Analysis Truths

DANIELA FERREIRO, IGNACIO CASSO and JOSE F. MORALES
Universidad Politécnica de Madrid (UPM), Madrid, Spain
IMDEA Software Institute, Madrid, Spain
(*e-mails:* `d.ferreiro@alumnos.upm.es`, `ignacio.casso@imdea.org`, `josefrancisco.morales@upm.es`)

PEDRO LÓPEZ-GARCÍA
Spanish Council for Scientific Research (CSIC), Madrid, Spain
IMDEA Software Institute, Madrid, Spain
(*e-mail:* `pedro.lopez@csic.es`)

MANUEL V. HERMENEGILDO
Universidad Politécnica de Madrid (UPM), Madrid, Spain
IMDEA Software Institute, Madrid, Spain
(*e-mail:* `manuel.hermenegildo@upm.es`)

submitted 17 January 2025; revised 22 April 2025; accepted 30 April 2025

Abstract

Static analysis is an essential component of many modern software development tools. Unfortunately, the ever-increasing complexity of static analyzers makes their coding error-prone. Even analysis tools based on rigorous mathematical techniques, such as abstract interpretation, are not immune to bugs. Ensuring the correctness and reliability of software analyzers is critical if they are to be inserted in production compilers and development environments. While compiler validation has seen notable success, formal validation of static analysis tools remains relatively unexplored. In this paper we present *checkification*, a simple, automatic method for testing 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 lies in its simplicity, which stems directly from framing it within the `Ciao` assertion-based validation framework, and its blended static/dynamic assertion checking approach. We demonstrate that in this setting, the analysis can be tested with little effort by combining the following components already present in the framework: 1) the *static analyzer*, which outputs its results as the original program source with assertions interspersed; 2) the assertion *run-time checking* mechanism, which instruments a program to ensure that no assertion is violated at run time; 3) the *random test case generator*, which generates random test cases satisfying the properties present in assertion preconditions; and 4) the *unit-test framework*, which executes those test cases. We have applied our approach to the `CiaoPP` static analyzer, resulting in the identification of many bugs with reasonable overhead. Most of these bugs have been either fixed or confirmed, helping us detect a range of errors not only related to analysis soundness but also within other aspects of the framework.

KEYWORDS: static analysis, testing, run-time checks, assertions, abstract interpretation, logic programing, constraint logic programing

1 Introduction

Static analysis tools play an important role in different stages of the software development cycle, such as code verification and optimization. However, building modern analyzers for programing languages presents significant challenges since these systems are typically large and complex, making them prone to bugs. This is a limitation to their applicability in real-life production compilers and development environments, where they are used in critical tasks that need reassurance about 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. This is probably 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 validation technique, 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 challenging.

In this paper we propose *checkification*, a simple, automatic, technique for testing static analyzers. We believe the approach is general in nature, and can be applied effectively to a wide class of static analyzers, provided some kind of run-time checking is feasible for the properties inferred. Herein we develop the proposal for concreteness in the context of the `Ciao` (Hermenegildo *et al.* 2012) logic programing-based, multi-paradigm language. The `Ciao` programing environment includes `CiaoPP`, a large and complex abstract interpretation-based static analysis tool which faces the specific challenges that we are addressing. Recently, there has been some interesting work (Stade *et al.* 2024) aimed at verifying the partial correctness of the PLAI analysis *algorithm* (also referred to as "the top-down solver") that lies at the heart of `CiaoPP` using the Isabelle prover (Paulson, 1990), but verification of the actual implementation remains a challenge. Like other "classic" analyzers, the `CiaoPP` formal framework has evolved for a long time, incorporating a large number of abstract domains, features, and techniques, adding up to over half a million lines of 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 (see Table 1 for an overview of some of the abstract domains that are bundled with the system and their maturity status). We show in Section 4.3 how our proposed method reveals bugs, not only in the less-developed parts of the system but also in corner cases of the more mature components, such as the handling of *built-ins*, run-time checking instrumentation, etc.

Table 1. *Abstract domains*

Abstract Domain	Properties Abstracted	Maturity Level	Reference
gr	aliasing, modes	intermediate	Bueno *et al.* (Bueno et al., 2006)
def	aliasing, modes	intermediate	García de la Banda *et al.* (1996)
sharing	aliasing, modes	mature	Muthukumar and Hermenegildo (1992)
shfr	aliasing, modes	mature	Muthukumar and Hermenegildo (1991)
shfr+nonvar	aliasing, modes	intermediate	
shareson	aliasing, modes	intermediate	Codish *et al.* (1993)
shfrson	aliasing, modes	intermediate	
son	aliasing, modes	mature	Søndergaard (1986)
share_amgu	aliasing, modes	mature	
shfr_amgu	aliasing, modes	mature	
shfrlin_amgu	aliasing, modes, linearity	mature	
share+clique	aliasing, modes	mature	Navas *et al.* (2006)
shfr+clique	aliasing, modes	mature	Navas et al. (2006)
share+clique+def	aliasing, modes	experimental	
shfr+clique+def	aliasing, modes	experimental	
eterms	types	mature	Vaucheret and Bueno (2002)
polyhedra	numerical	experimental	Bagnara *et al.* (2002)
depth-k	term structure	intermediate	Sato and Tamaki (1984)
det	determinacy	mature	Lopez-Garcia *et al.* (2005, 2010)
nfg	(non)failure	intermediate	Debray *et al.* (1997); Bueno *et al.* (2004)

A feature of `Ciao` that will be instrumental to our approach is the use of a unified assertion language across the components of the `Ciao` framework (Hermenegildo *et al.* 1999, 2003), which together implement a unique blend of static and dynamic assertion checking. These components (and their algorithms) include:

1. The *static analyzer* (Muthukumar and Hermenegildo, 1992; Hermenegildo *et al.* 2000; Garcia-Contreras *et al.* 2020), (top-down analysis framework) which expresses the inferred information as assertions interspersed within the original program.
2. The assertion *run-time checking framework* (Stulova *et al.* 2015, 2016), 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 (Casso *et al.* 2020), which generates random test cases satisfying the properties present in assertion preconditions.
4. The *unit-test framework* (Mera *et al.* 2009), 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, obtaining the analysis results expressed as assertions interspersed within the original code. Then, the *status* of these assertions is *switched* into *run-time checks*, that will ensure that violations of those assertions are reported at execution time. Finally, random test cases are generated and executed to exercise those *run-time checks*.

Given that these assertions (the analyzer output) must cover all possible concrete executions (and assuming the correctness of our checking algorithm implementation), if any assertion violation is reported, assuming that the run-time checks are correct, it means that the assertion was incorrectly inferred by the analyzer, thus revealing an error in the analyzer itself. The error can of course sometimes also be in the run-time checks, but typically run-time checking is simpler than inference. This process is automatable, 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 a tool environment that follows `Ciao` assertion model, is comparatively simple, at least conceptually.

The idea of checking at run time the properties or assertions inferred by the analysis for different program points is not new. For example, Wu *et al.* (2013) successfully applied this technique for checking a range of different aliasing analyses. However, these approaches require the development of tailored instrumentation or monitoring, and 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 `Ciao` assertions model. As mentioned before, by developing the approach within such a framework, it can be implemented with many of the already existing algorithms and components in the system, in a very simple way. As a result, our initial prototype was quite simple, even if, inevitably, the current working version has of course grown quite a bit in order to add functionality, make it easier to use, include specific instrumentation, collect performance data, etc. Moreover, the components and algorithms of the `Ciao` system used by our implementation have been extensively validated, providing greater confidence in the correctness of our approach. Consequently, when our method flags a runtime checking error, we can be more certain that it identifies an actual error in the analyzer. If no error exists in the analyzer, such runtime checking errors help us locate and fix bugs in our implementation. These fixes will likely address bugs in other `Ciao` system components used by our implementation. Importantly, these components can also serve other purposes in the software development process. In conclusion, any runtime checking error flagged by our approach contributes to improving the entire `Ciao` system.

We also argue 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, run-time checking framework, and unified assertion language, provided only that the assertion language is extended to include the properties that are inferred by the domains.

The rest of the paper is structured as follows: Section 2 provides the background needed for describing the main ideas and contributions of the paper. In particular, we describe the basic components used in our approach. Section 3 then presents and discusses our proposed "checkification" algorithm for testing static analyzers, with an initial illustrative example, the basic reasoning behind the approach (Section 3.1), the operation of the algorithm (Section 3.2), and discussions of some of additional issues involved (Sections 3.3 and 3.4). In Section 4 we present our experimental evaluation and results. We explain the evaluation setup (Section 4.1) including experiments, analyzer configuration, abstract domains and properties studied, and programs analyzed. We then present and discuss the results of this evaluation (Section 4.2), in terms of the errors found and cost of the technique. Section 4.3 then presents further discussion with examples of the classes of errors detected which also serves to go over some of the practical uses of the approach. We conclude by discussing additional related work in Section 5, and presenting some conclusions and perspectives in Section 6.

2 Basic components

2.1 Assertion Language

Assertions are syntactic objects which allow expressing properties of programs that should hold at certain points of program execution. Assertions are used everywhere in `Ciao`, from documentation and foreign interface definitions to static analysis and dynamic debugging. Two types of `Ciao` assertions that are relevant herein are *predicate* assertions (`pred` for short) and *program-point* assertions:[1] The first ones are declarations that provide partial specifications of a predicate. They have the following syntax:

$$:-\ [\mathit{Status}]\ \texttt{pred}\ \mathit{Head}\ [:\ \mathit{Calls}]\ [=>\ \mathit{Success}]\ [+\ \mathit{Comp}].$$

and express that a) calls to predicate *Head* that satisfy precondition *Calls* are admissible and b) that, for such calls, the predicate must satisfy post-condition *Success* if it succeeds, and global computational properties *Comp*. If there are several `pred` assertions, the set of *Calls* fields define the admissible calls to the predicate. *Program-point* assertions are reserved literals that appear in the body of clauses and describe properties that hold in the run-time constraint store every time execution reaches that point in the clause at run time. Their syntax is *Status*(*State*). Both of these kinds of assertions can have different values in the *Status* field depending on their origin and intended use. Assertion statuses relevant herein include:

- `true`, which is the status of the assertions that are output from the analysis (and thus must be safe approximations of the concrete semantics);
- `check`, which indicates that the validity of the assertion is unknown and it must be checked, statically or dynamically, and is the *default* value of *Status* when not indicated; and,
- `trust`, which indicates that the analyzer should assume this assertion to be correct, even if it cannot be automatically inferred.

[1] We will also use an additional form, `entry` assertions, that will be introduced later.

Example 2.1
(Some assertions). The following code fragment provides examples of both types of assertions, predicate and program-point; all these assertions have status `check`:

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

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

The first two `pred` assertions define two different ways in which `append/3` is expected to be called. The first one states that `append/3` may be called with the two first arguments instantiated to lists and the third a variable, and that, if such a call succeeds, then the third argument should be bound to a list. This first assertion also states that when called like this, the predicate should have only one solution and should not fail (`det`, a global computational property). The second `pred` assertion states that `append/3` may also be called with the third argument instantiated to a list and the first two variables, and that, if such a call succeeds, then the first and second arguments should be bound to lists, and that in this case the predicate should produce one or more solutions (`multi`, also a global computational property), but, again, not fail. There is also a program-point assertion in the second clause of `append/3` that states that if execution reaches that point in that clause, all of `Xs`, `Ys`, and `Zs` should be bound to lists. For all these assertions the `check` status indicates that these are desired properties that need to be checked, statically or dynamically, but have not been proven true or false yet.

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. For our purposes herein, we will consider typically properties that are *native* to `CiaoPP`, that is that can be abstracted and inferred by some domain in `CiaoPP`. This includes a wide range of properties, from types, modes and variable sharing, to determinacy, (non)failure, and resource consumption. We refer the reader to Puebla *et al.* (2000); Hermenegildo *et al.* (2005, 2012) and their references for a full description of the `Ciao` assertion language.

In the `Ciao` assertion syntax, properties can also be *in-lined* in the predicate arguments, also referred to as using *modes*. Such modes are *property macros* that serve to specify in a compact way several properties referring to a predicate argument. A specific syntax, resembling that of *predicate* assertions is used to define modes.

Example 2.2
(Modes). For example, if the following modes are defined:[2]

```
:- modedef  +(A,P) : P(A).           % A has property P in calls
:- modedef  -(A,P) : var(A) => P(A). % A is var on calls and has property P on success
```

[2] Note that "-" is often also defined simply as: `:- modedef -(A,P) => P(A).` As mentioned before, in Ciao modes are user-definable.

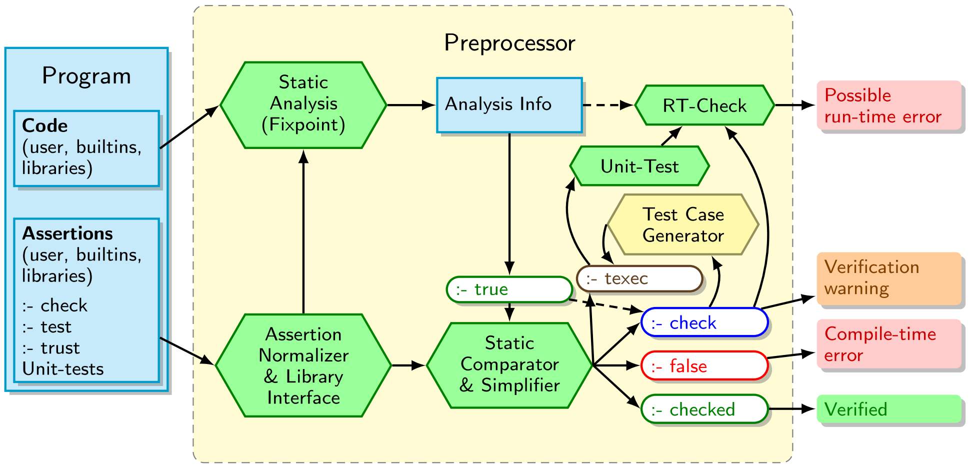

Fig. 1. The `Ciao` assertion framework (`CiaoPP`'s verification/testing architecture).

then the assertions in the previous program can be expressed equivalently as follows:

```
:- check pred append(+list,+list,-list) + det.
:- check pred append(-list,-list,+list) + multi.
```

Figure 1 depicts the overall architecture of the `Ciao` unified assertion framework. Hexagons represent tools, and arrows indicate the communication paths among them. Most of this communication is performed in terms of assertions. The input to the process is the user program, optionally including some assertions. Such assertions always include any assertions available in the libraries for *built-ins* (the basic operations of the source language), or for predicates exported by such libraries that are used by the code being analyzed (left part of Figure 1).

2.2 Static Program Analysis

Abstract interpretation is a formal framework for static analysis that allows inferring program properties that hold for all possible program executions. Different abstractions, called abstract domains, are used in this process for approximating sets of concrete run-time states. The `CiaoPP` analyzer is abstract interpretation-based, and its design consists of a common abstract-interpretation framework based on fixpoint computation parameterized by different, *pluggable* abstract domains (the *Static Analysis* hexagon in Figure 1). This means that the set of properties that are used in assertions is extensible with new abstract domains defined as *plug-ins* to support them. Depending on the selected domain or combination of domains for analysis, `CiaoPP` constructs a *program analysis graph*, starting from the program *entry points*. In this graph, nodes represent the different ways in which predicates are called. A predicate can have multiple nodes associated with it if it is called in different ways (calling contexts). For each calling context, properties are inferred that hold if the predicate succeeds (and also global properties). These properties will be emitted also as assertions, which will have status `true` (represented by the *Analysis Info* box in Figure 1). Optionally, a new source file is generated for the analyzed program, which is identical to 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 added for each predicate. In particular, if there are several different calling contexts for a given predicate, there will typically be a predicate assertion added for each of these contexts (this is also referred to as *multivariance* in the analysis). Further details of this process can be found in Muthukumar and Hermenegildo (1989, 1990, 1992). In any case, when discussing the identified bugs later in the paper, we will provide more explanations as needed.

2.3 Run-Time Checking

Static analysis is used for compile-time checking of assertions. However, due to the inherent undecidability of static analysis, sometimes properties cannot be verified statically. 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 to make it run-time safe. These dynamic checks will encode the semantics 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 `Ciao` assertion model). Note that almost all current abstract interpretation systems assume in their semantics that the run-time checks will always be executed. However, `CiaoPP` does not make this assumption by default, that is it is configurable as an option, since in some use cases run-time checks may in fact be disabled by the user for deployment.

Checking at run time program state properties, such as traditional types and modes, can be performed relatively easily: as mentioned before, most properties are *runnable*, and the `check/1` wrapper will ensure that the check will succeed or raise an error, without binding any arguments. For example, calling `check(list(X))` with `X = []`, `X = [a]`, or `X = [A,B]`, will succeed, without binding any variables, while calling `check(list(X))` with `X = a`, `X = f(a)`, or `X = A`, will raise an error.[3] In practice a quite rich set of properties is checkable, including types, modes, variable sharing, exceptions, determinacy, (non-)failure, choice-points, and more, blending smoothly static and dynamic techniques. On the other hand, checking at run time other global properties such as cost and, specially, termination, is obviously less straightforward. While checking these types of properties could conceptually be done with our proposed algorithm, in this paper we concentrate on the other properties mentioned.

2.4 Unit Tests, Test Case Generation, and Assertion-based Testing

Test inputs can be provided by the user, by means of `test` assertions (unit tests). The run-time checking mechanism can test these assertions but also any other assertion in any predicate called by the test case, that was not verified 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

[3] A discussion of *instantiation* checks and *compatibility* checks is appropriate at this point but beyond the scope of the paper. Checks are *instantiation* checks unless otherwise stated. The reader is referred to (Hermenegildo *et al.* 1999, 2003) for details.

property predicates as generators. For example, calling `list(X)` with `X` uninstantiated generates lazily, through backtracking, an infinite set of lists, `X = [], X = [_],` `x = [_,_],` etc. Stating a type for the list argument will then also generate concrete values for the list elements. This enumeration process can be combined in `Ciao` with the different supported search rules (breadth-first, iterative deepening, random search, etc.) to produce, for example fair enumerations. This idea has been extended recently (Casso *et al.* 2020) to a full 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*. In other words, *assertion-based testing* involves generating and running relevant test cases that exercise the run-time checks of the assertions in a program to test if those assertions are correct. This technique (present in the `Ciao` model since its origins) yields similar results to *property-based testing* (Claessen and Hughes, 2000) but in a more integrated way within the overall assertion model and within `CiaoPP`, rather than as a separate technique. Such automatic generation is currently 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 addition, users can also write their own generators and of course other test generation techniques and tools can be used (Fortz *et al.* 2020).

3 The checkification algorithm

This section provides a detailed overview of the proposed algorithm for testing the static analyzer, which incorporates all the components mentioned above.

Illustrative example. Let us start by sketching the main idea of our approach with a motivating example. Assume we have the following simple Prolog program, where we use an `entry` assertion to define the entry point for the analysis. The `entry` assertion indicates that the predicate is called with its second argument instantiated to a list and the third a free variable (we use the mode definitions of Example 2.2):

```
:- entry prepend(_,+list,-).

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 of the following abstract values:

- `ground` (the variable is ground),
- `var` (the variable is free),
- `nonground` (the variable is not ground),
- `nonvar` (the variable is not free),
- `ngv` (the variable is neither 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`

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

Fig. 2. An incorrect simple mode analysis.

literal, the abstract value for `Ys` is not modified at all.[4] 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 Figure 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 `any/1` properties (i.e., top, or unknown) would not actually be included in the analysis output for conciseness, but are included in Figure 2 for clarity.

The objective of our approach is to check dynamically the validity of these `true` assertions from the analyzer, that in this case contain an error. The insight is that, thanks to the different capabilities of the `Ciao` model presented previously, this can be achieved by (**1**) *turning the status of the* `true` *assertions produced by the analyzer into* `check`, as shown in Figure 3.[5] 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, that is whether the information inferred is incorrect. To do this, (**2**) we run the transformed program (Figure 3) again through `CiaoPP` (Figure 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. In other words, 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**) run `prepend/3` on a set of test cases. These may in general be already available as test assertions in the program or, alternatively, the random test case generator can be used to generate them. for example for `prepend/3` the test generation framework will ensure that instances of the goal `prepend(X,Xs,Ys)` are generated, where `Xs` is constrained to be a list, and `Ys` remains a free variable. However, `X` and the elements of `Xs` will otherwise 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

[4] Note that early LP analyzers often had errors of this kind, which led to very active development of *variable sharing* analysis domains. These constituted some of the very first Abstract Interpretation-based pointer aliasing analyses for any programing language.

[5] Again, we include the `any/1` property in Figure 3 for clarity of exposition, and for consistency with Figure 2.

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

Fig. 3. The instrumented program.

a run-time error in the `check` in line 12, letting us know that the third program point, and thus the analysis, is incorrect.[6]

The same procedure can be followed to debug different analyses with different benchmarks. If the execution of any test case reports a run-time 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).

3.1 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 meant for providing a more precise explanation, rather than deep formalization, which is arguably not really necessary, thanks to the simplicity of the approach that builds on the different parts of the system that act as trusted base.

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

[6] 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` (assertions to be taken as granted by the analyzer), 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 Garcia-Contreras *et al.* (2019).

of the analysis would be incorrect, that is 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 programing 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: $S_P^{I^-} \subset_P S_P^+ \iff \forall i \in I^-,\ S_P^{\{i\}} \subset_P S_P^+$.
- 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, to make sure 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.

3.2 Operation of the algorithm

We now show the concrete algorithm for implementing our proposal, that is 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 $\mathcal{D}$ from a collection of benchmarks and domains, and execute the AnaTest$(P, \mathcal{D})$ 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 $(P, \mathcal{D})$ pair more than once.[7] 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 benchmark programs with a single entry point with no inputs (for example just a single `void main()` function as entry point for C). In `Ciao` program signatures and types 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 or skipped altogether. Note also that if the program P 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. In the absence of assertions, the test case generation framework has already some

[7] Clearly, coverage of the program and coverage of the analyzer code could be a useful metric here to decide when to finish.

Algorithm 1 The "Checkification" Analysis Testing Algorithm

```
1:  procedure AnaTest($P, D$)                                ▷ For program $P$ and domain $\mathcal{D}$
2:      $result \leftarrow$ None
3:      $P_{an} \leftarrow$ analyze and annotate $P$ with domain $\mathcal{D}$ (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 a run-time error at line $l$ then
9:              $result \leftarrow$ Error($input, l$)
10:         else if maximum time or number of test executions is reached then
11:             $result \leftarrow$ Timeout
12:     until $result \neq$ None return $result$
```

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 calling contexts, or with custom user-defined generators in their absence.

When the algorithm detects a faulty program-point assertion for some *input* (Error$(input, l)$), it means that the concrete execution reaches a state not captured by the (potentially safely over-approximated) result of analysis. It is important to note that although error diagnosis and debugging are primarily left for the user to manually perform, our tool facilitates the task in some aspects. Firstly, it is possible to reconstruct (or store together with the test output) additional information 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). Secondly, the *assertion-based testing* tool supports shrinking of failed test cases, so we can expect reasonably small variable substitutions in the errors reported. Lastly, as sketched in Algorithm 1, the error location and trace reported by the run-time verification framework provide an approximate idea of the point where the analysis went wrong, even if not necessarily of the original reason why (which requires a different step of diagnosis). If the run-time check error points to a program-point assertion right after a call to an imported predicate, then the analysis erred in applying the *entry* declaration for the predicate, the *entry* declaration was wrong, or if there was no *entry* declaration, the analysis failed to compute the "topmost" abstract state reachable from the call abstract state.

3.3 Some considerations on properties

In order to test an analysis with the algorithm proposed, two conditions must be met. The first one is to have a translation from the internal representation of the abstract values in the domain to `Ciao` user-level properties. These properties that can represent the information inferred by a given domain or domains are called the *native properties* of the domain(s). Note that these are already requirements for any abstract domain intended to make full use of the framework, so normally all implemented domains include the definition of the corresponding native properties and the translation from abstract domain values to them.

The other condition is to be able to perform run-time checks for those properties, that is that they can be used by the run-time checking framework. As discussed in Section 2, such run-time checks can range from very simple or even already built into the language (like, e.g., var/1), to intermediate (like, e.g., aliasing or groundness), to more complex and costly (like, e.g., costs or side-effects), to theoretically impossible (like, e.g., termination). But they can also be safely approximated to detect errors or to issue warnings (e.g., in the case of termination by detecting repeated identical calls which lead to non-termination or by timeouts). It is also important to note that complex analyses such as termination are typically dependent on a number of other instrumental analyses which can themselves be checked.

In general, the availability of run-time checks is a standard requirement for domains to be able to make full use of the framework, in order to support the dynamic checks that are generated when properties cannot be proved statically. This functionality is normally implemented when a new abstract domain is added to the system, by also defining the related properties to be used in assertions. If the definition of these native properties is provided directly in the source language, then such properties are typically already runnable and thus available for run-time checking; however, it is also possible to provide an implementation specialized for run-time checking if desired. For properties that are declared native but are not written in the source language, then a run-time test version must be provided. In practice, most current `Ciao` abstract domains include the mentioned functionalities and can be tested as is with the proposed approach.

3.4 Multivariance and path-sensitivity

As presented, it could appear that our approach could miss some analysis errors even if the right test cases are used, since we have, to all appearances, disregarded *multi-variance and path-sensitivity*. In fact, in `CiaoPP` the information inferred is fully multi-variant, and separate path information is kept for each variant (i.e., calling context). However, by default, the analyzer produces an output that is easy for the programmer to inspect, that is close to the source program. This means that when outputting the analysis results, by default the different versions of each predicate (and the associated information) are combined 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. This issue is controlled by a flag that, when enabled, ensures that the different *versions* are not collapsed and are instead *materialized* into different predicate instances. This way, multiple *versions* may be generated for a given predicate, if there are separate paths to them with different abstract states, 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 Evaluation

In this section we report on the different experiments that we have conducted in order to benchmark the checkification approach and assess its practicality.

4.1 Evaluation setup

4.1.1 The experiments

The experiments have consisted in, for a set of benchmarks, analyzing them with different abstract domains, performing the checkification transformation on the analysis results, and testing the resulting programs on sample inputs with run-time checks activated. The objective has been to assess whether we can indeed find errors using the technique and to estimate the cost involved in detecting those errors.

4.1.2 Analyzer configuration

The experiments were run with `Ciao/CiaoPP` version 1.23. The tested analyses used the standard, default configuration of the abstract interpretation framework, that is the default values of the different flags, such as, for example, using multi-variance on calls, using the original *PLAI* fixpoint algorithm, etc., but, of course, differ in the abstract domain selected for performing each analysis.

4.1.3 Properties and domains

In the experiments we used a wide range of analyses for different properties that are typically of interest when describing or verifying logic programs (the list of all the abstract domains used is provided in Table 1):[8]

- The first class of properties is aimed at capturing variable instantiation state, that is which variables are bound to ground terms, or free, and, if they are not ground, the variable sharing relationships among them. These properties are approximated using *aliasing* and *modes*-style abstract domains.
- The second set of properties refers to the shapes of the data structures constructed by the program in memory. These properties are tracked by the *term structure* and *types* classes of abstract domains.
- The third class of properties that we have considered refers to the numerical relations among program variables, which are useful to describe properties of numerical parts of programs. For these, we use in our experiments *numerical*-style abstract domains.
- Finally, we also evaluate the approach on analyses for computational properties, that is properties of whole computation subtrees, in particular *determinacy* and *(non)failure*. These analyses sometimes do not provide the information at the program points between literals, but rather at the predicate level. In these cases, the transformation and tests are done at the predicate level.

[8] The table also provides references for each domain, except for some that are combinations of other domains not explicitly described in other papers.

Table 2. *Benchmarks*

`Bench`	
`mmatrix`	matrix multiplication for two matrices with dimensions $n \times n$;
`qsort`	the quicksort program;
`exp`	exponential calculation;
`aiakl`	initialization for abstract unification in AKL analyzer;
`ham`	a program that generates the sequence of Hamming numbers;
`fft`	fast Fourier transformation calculation;
`factorial`	recursive factorial calculation;
`witt`	the WITT clustering system implementation;
`poly`	a program that raises a polynomial $(1 + x + y + z)$ to the 10th power symbolically;
`deriv`	symbolic differentiation of a given formula;
`grammar`	a simple sentence parser;
`fib`	a program that finds N-th Fibonacci number;
`boyer`	a theorem prover implementation based on Lisp by R. Boyer (nqthm system), performs symbolic evaluation of a formula;
`queens`	the N queens program (number of the queens being the input);
`jugs`	the water jugs problem;
`bid`	compute opening bid for bridge hand;
`nreverse`	naive list reversal;
`guardians`	prison guards game;
`crypt`	crypto-multiplication puzzle solver;

4.1.4 Programs analyzed

The programs used in our experiments can be divided into two different groups:

The first group, listed in Table 2, comprises a number of well-known, classic benchmarks. Some of them also represent kernels of applications. For example, `aiakl` is the main part of an analyzer for the AKL language; `boyer` is the kernel of a theorem prover; and `witt` is the central part of a conceptual clustering application.

The second group comprises complete systems that are in current use:

- A `filtering` tool (Ferreiro *et al.* 2023), used regularly for creating teaching materials. This tool contains 1.1K lines of code in its kernel, and uses also a number of `Ciao` libraries. It is an interesting example since it includes different *built-in* predicates for handling files and streams.
- `Deepfind` (Garcia-Contreras *et al.* 2016), a tool that facilitates searching code repositories and libraries by querying for semantic characteristics of the code. Its kernel consists of around 10 files where the Prolog code is about 1.5K lines, and uses in addition a good number of `CiaoPP` libraries to perform program analysis and assertion comparison.
- The classic `chat-80` program (Warren and Pereira 1982), a natural language interface to a geographical database. It comprises 4.8k lines of code across 22 files. While typically used more as a demo than a real application, we have included it in this group because it is of good size, contains a number of system libraries

using different Prolog *built-ins* and library predicates, and is known to stress several abstract interpretation domains.
- `LPdoc`, a documenter for LP systems used by `Ciao` and XSB (Hermenegildo, 2000; Hermenegildo and CLIP Group, 1997). This is the largest example with its kernel Prolog code being analyzed comprising about 22K lines, plus the use of many `Ciao` libraries.
- `Spectector` (Guarnieri *et al.* 2020), a tool for automatically detecting leaks introduced by speculatively executed instructions in x64 assembly programs. It consists of 15 modules with around 1.6K lines of code.
- The `s(CASP)` (Arias *et al.* 2018) system is a top-down interpreter for ASP programs with constraints. The Prolog code is distributed into 44 modules.

No special analysis-related criteria were used in benchmark selection, and the code was analyzed and run as is, without modifications. The classic benchmarks, by default, include annotations with program assertions that describe the expected behavior, while no additional information is provided in the real-world programs.

4.2 Results

4.2.1 Cost of the technique

While run-time overhead is not our primary focus, we have evaluated this aspect in order to study whether the algorithm has an acceptable cost. This cost obviously has two components: the analysis time and the testing time (the transformation time is negligible). Regarding the testing time, note that the execution time of run-time tests can be reduced significantly through caching techniques (Koukoutos and Kuncak, 2014; Stulova *et al.* 2015). However, we decided not to use these optimizations for a number of reasons: to simplify the implementation; to avoid dependence on the implementation of other parts; and to avoid any bugs that optimizations could potentially hide, making it harder to identify them. The results are presented in Tables 3, 4, 5 and 6 for the different benchmarks and domains. The experiments were run on a MacBook Air with the Apple M1 chip and 16 GB of RAM. Each column in these tables corresponds to an abstract domain and in turn contains two sub-columns. The first sub-column is the absolute execution time in seconds for each benchmark once checkification has been applied, for the properties inferred by the corresponding abstract domain. This time includes both the transformation process, in which the annotations of the analysis are modified by replacing the status of *true* assertions by *check* status and inserting *run-time checks*, as well as the testing time of the instrumented program. The numbers in parentheses in the second sub-column provide the analysis times in seconds for the benchmarks, again for the different abstract domains.

Regarding the testing times, when an error is found during a test run, the time to do so is typically negligible. In these cases, we report instead the testing time after fixing the analysis so that no bugs are detected and the testing runs to completion. This is obviously also the case when no bug is detected to begin with. Thus, the testing times reported are always for *complete runs*, which we feel are more useful for estimating the testing cost.

Table 3. *Timings and errors detected (1)*

	Absolute run time, s (Analysis time, s)									
Program	*gr*		*def*		*sharing*		*sharefree*		*shfr+nonvar*	
mmatrix	0.2	(0.25)	0.2	(0.25)	0.4	(0.25)	0.4	(0.25)	0.4	(0.25)
qsort	1.0	(0.21)	1.0	(0.21)	1.7	(0.21)	1.7	(0.22)	1.7	(0.22)
exp	$0.4{\cdot}10^{-1}$	(0.22)	$0.4{\cdot}10^{-1}$	(0.22)	0.2	(0.23)	0.2	(0.23)	0.2	(0.23)
aiakl	$0.2{\cdot}10^{-3}$	(0.25)	$0.3{\cdot}10^{-3}$	(0.24)	$0.8{\cdot}10^{-3}$	(0.25)	$0.9{\cdot}10^{-3}$	(0.26)	$0.9{\cdot}10^{-3}$	(0.25)
ham	0.1	(0.33)	0.1	(0.34)	0.3	(0.34)	0.3	(0.34)	$\dbox{\text{n/a}^{\dagger}}$	(0.26)
fft	0.9	(0.36)	$\boxed{2.5^{\dagger}}$	(0.35)	6.8	(0.36)	6.8	(0.36)	6.8	(0.37)
factorial	0.1	(0.20)	0.1	(0.20)	0.2	(0.20)	0.2	(0.21)	0.2	(0.20)
witt	11.4	(1.08)	23.2	(1.08)	57.7	(1.12)	60.9	(1.14)	$\dbox{\text{n/a}^{*}}$	(1.14)
poly	0.8	(0.31)	8.7	(0.34)	32.3	(0.44)	30.9	(0.42)	31.7	(0.52)
deriv	$0.2{\cdot}10^{-3}$	(0.28)	$0.5{\cdot}10^{-3}$	(0.28)	$2.4{\cdot}10^{-3}$	(0.29)	$0.2{\cdot}10^{-2}$	(0.29)	$0.3{\cdot}10^{-2}$	(0.29)
grammar	$0.2{\cdot}10^{-4}$	(0.23)	$0.5{\cdot}10^{-4}$	(0.22)	$1.5{\cdot}10^{-4}$	(0.23)	$0.2{\cdot}10^{-3}$	(0.23)	$0.2{\cdot}10^{-3}$	(0.24)
fib	0.1	(0.21)	0.1	(0.21)	0.3	(0.20)	0.3	(0.20)	0.3	(0.29)
boyer	0.7	(0.50)	2.7	(0.52)	15.8	(0.56)	16.0	(0.56)	16.2	(0.57)
queens	6.0	(0.23)	6.0	(0.24)	12.4	(0.23)	13.2	(0.24)	13.2	(0.23)
jugs	$0.2{\cdot}10^{-3}$	(0.31)	$0.9{\cdot}10^{-3}$	(0.32)	$0.3{\cdot}10^{-2}$	(0.34)	$0.3{\cdot}10^{-2}$	(0.34)	$0.3{\cdot}10^{-2}$	(0.32)
bid	$0.2{\cdot}10^{-2}$	(0.38)	$0.2{\cdot}10^{-2}$	(0.37)	$0.4{\cdot}10^{-2}$	(0.38)	$0.5{\cdot}10^{-2}$	(0.38)	$\dbox{\text{n/a}^{*}}$	(0.39)
nreverse	$0.3{\cdot}10^{-1}$	(0.20)	1.3	(0.21)	4.0	(0.22)	$\utilde{4.0}$	(0.22)	4.1	(0.21)
guardians	0.4	(0.28)	0.4	(0.28)	0.7	(0.29)	0.7	(0.29)	$\dbox{\text{n/a}^{*}}$	(0.29)
crypt	$0.1{\cdot}10^{-1}$	(0.31)	$0.1{\cdot}10^{-1}$	(0.31)	n/a	(n/a)	0.1	(0.32)	0.1	(0.31)
exfilter	0.2	(6.10)	21.9	(6.84)	83.4	(14.58)	83.6	(13.55)	n/a	(n/a)
deepfind	0.4	(7.12)	0.3	(6.39)	1.9	(23.46)	2.1	(14.06)	1.0	(9.72)
chat-80	$0.4{\cdot}10^{-1}$	(5.41)	$\dbox{\text{n/a}^{*}}$	(5.78)	5.4	(54.60)	5.5	(54.28)	5.7	(54.29)
LPdoc	0.3	(22.20)	1.0	(23.03)	52.3	(72.45)	53.6	(65.06)	53.0	(112.12)
Spectector	0.6	(4.27)	$\dbox{\text{n/a}^{*}}$	(4.43)	29.1	(5.72)	24.0	(5.75)	24.5	(5.77)
s(CASP)	0.6	(10.35)	24.2	(10.69)	56.1	(237.75)	56.6	(197.62)	$\dbox{\text{n/a}^{*}}$	(205.86)

Classes of bugs found are marked with: $\dbox{}$ = abstract domain implementation, ... = fixpoint algorithms, $\boxed{}$ = semantic inconsistencies, $\sim$ = run-time check instrumentation, $=$ = third-party libraries.

If the time is labeled as **n/a**, it indicates that there is no time recorded due to the presence of a timeout or an unresolved bug, or that no analysis results were available. The latter can be caused by a crash during analysis or by the analysis output being malformed, for example missing assertions.

The results show that execution time of tests is quite reasonable, typically taking no more than around 60 s. Performance can be improved if needed by activating only the run-time semantics of the predicate assertions and/or disabling multi-variance.

4.2.2 Errors found

We now turn our attention to the most important point of whether the technique can indeed find errors in the analyses. We have manually analyzed the root causes of the

Table 4. *Timings and errors detected (2)*

	Absolute run time, s (Analysis time, s)									
Program	*shareson*		*shfrson*		*sondergaard*		*share_amgu*		*shfr_amgu*	
`mmatrix`	0.6	(0.25)	0.7	(0.25)	0.4	(0.25)	0.4	(0.25)	0.4	(0.26)
`qsort`	2.8	(0.21)	2.8	(0.21)	1.7	(0.21)	1.7	(0.21)	1.7	(0.21)
`exp`	0.2	(0.23)	0.2	(0.23)	0.2	(0.23)	0.2	(0.23)	0.2	(0.23)
`aiakl`	$0.1\cdot10^{-2}$	(0.26)	$0.1\cdot10^{-2}$	(0.26)	$0.1\cdot10^{-2}$	(0.26)	$0.8\cdot10^{-3}$	(0.25)	$0.9\cdot10^{-3}$	(0.26)
`ham`	0.5	(0.36)	0.5	(0.37)	0.4	(0.34)	0.3	(0.35)	0.3	(0.34)
`fft`	9.5	(0.38)	9.6	(0.38)	7.5	(0.36)	6.8	(0.36)	6.8	(0.37)
`factorial`	0.3	(0.20)	0.3	(0.21)	0.2	(0.20)	0.2	(0.20)	0.2	(0.20)
`witt`	60.7	(3.93)	n/a	(n/a)	n/a	(n/a)	57.7	(1.10)	61.0	(1.14)
`poly`	31.8	(0.48)	32.0	(0.49)	31.7	(0.39)	32.4	(0.57)	31.2	(0.48)
`deriv`	$0.3\cdot10^{-2}$	(0.29)	$0.4\cdot10^{-2}$	(0.30)	$0.2\cdot10^{-2}$	(0.29)	$0.2\cdot10^{-2}$	(0.29)	$0.2\cdot10^{-2}$	(0.29)
`grammar`	$0.2\cdot10^{-3}$	(0.23)	$0.2\cdot10^{-2}$	(0.24)	$0.2\cdot10^{-3}$	(0.24)	$0.1\cdot10^{-3}$	(0.24)	$0.2\cdot10^{-3}$	(0.23)
`fib`	0.4	(0.21)	0.4	(0.21)	0.3	(0.20)	0.3	(0.22)	0.3	(0.21)
`boyer`	20.3	(0.57)	$\boxed{20.7^{\dagger}}$	(0.59)	19.7	(0.56)	15.7	(0.58)	16.2	(0.57)
`queens`	21.6	(0.24)	23.1	(0.28)	13.6	(0.24)	12.3	(0.23)	13.4	(0.24)
`jugs`	$0.3\cdot10^{-2}$	(0.33)	$0.3\cdot10^{-2}$	(0.33)	$0.3\cdot10^{-2}$	(0.33)	$0.3\cdot10^{-2}$	(0.33)	$0.3\cdot10^{-2}$	(0.33)
`bid`	$0.7\cdot10^{-2}$	(0.39)	$0.7\cdot10^{-2}$	(0.40)	$0.4\cdot10^{-2}$	(0.39)	$0.4\cdot10^{-2}$	(0.38)	$0.4\cdot10^{-2}$	(0.39)
`nreverse`	4.0	(0.22)	4.0	(0.22)	3.9	(0.22)	4.0	(0.21)	3.9	(0.22)
`guardians`	54.1	(0.29)	1.1	(0.31)	0.7	(0.29)	0.6	(0.29)	0.7	(0.29)
`crypt`	0.1	(0.42)	0.1	(0.35)	0.1	(0.32)	n/a	(n/a)	0.1	(0.31)
`exfilter`	n/a	(n/a)	n/a	(n/a)	n/a	(54.82)	83.7	(14.91)	83.5	(13.45)
`deepfind`	1.9	(9.49)	2.0	(10.12)	1.6	(50.12)	0.5	(49.88)	0.4	(12.14)
`chat-80`	0.4	(56.53)	0.5	(55.72)	0.5	(59.15)	0.4	(43.86)	0.5	(39.94)
`LPdoc`	53.0	(160.86)	53.5	(119.31)	52.1	(22.89)	52.7	(200.87)	52.5	(262.71)
`Spectector`	24.1	(5.96)	24.5	(5.80)	24.6	(8.71)	36.9	(10.97)	24.2	(5.91)
`s(CASP)`	56.1	(207.61)	56.1	(198.59)	56.3	(33.28)	57.0	(291.39)	55.8	(170.38)

Classes of bugs found are marked with: ⬚ = abstract domain implementation, . . . = fixpoint algorithms, □ = semantic inconsistencies, $\sim$ = run-time check instrumentation, ═ = third-party libraries.

errors found and classified them into different categories. The classification is indicated in Tables 3 to 6 by surrounding the numbers with different patterns. These bug categories include (I) defects in the implementation of the abstract domain, (II) defects in the implementation of the fixpoint algorithms, (III) semantic inconsistencies between components of the framework, (IV) run-time check instrumentation issues, and (V) defects related to third-party libraries. Table 7 provides a summary of identified bugs. The "Status" column indicates whether the bug is new or it was already a known issue at the time of running the experiments. The "Class" column lists the bug category, while the "Description" column provides a more detailed description of each bug. We will discuss stylized examples from each bug category in Section 4.3. In each column, bugs are additionally marked with symbols (e.g., †) to group those that correspond to the same issue.

The results of the experiments conducted so far are promising, allowing us to draw several significant conclusions and observations.

First and foremost, a good number of bugs and inconsistencies were found using the technique. These bugs were quite diverse, illustrating the power of the algorithm in finding all sorts of issues of different nature.

Table 5. *Timings and errors detected (3)*

Program	Absolute run time, *s* (Analysis time, *s*)									
	shfrlin_amgu		*share+clique*		*shfr+clique*		*share+clique+ def*		*shfr+clique+ def*	
`mmatrix`	0.5	(0.26)	0.4	(0.25)	0.4	(0.25)	0.4	(0.26)	0.4	(0.25)
`qsort`	1.8	(0.21)	1.7	(0.21)	1.7	(0.22)	1.7	(0.21)	1.7	(0.21)
`exp`	0.2	(0.23)	$0.4{\cdot}10^{-1}$	(0.23)	$0.4{\cdot}10^{-1}$	(0.23)	$0.4{\cdot}10^{-1}$	(0.22)	$\dbox{0.5{\cdot}10^{-1\dagger}}$	(0.23)
`aiakl`	$0.1{\cdot}10^{-2}$	(0.26)	$0.8{\cdot}10^{-3}$	(0.25)	$0.9{\cdot}10^{-3}$	(0.26)	$0.8{\cdot}10^{-3}$	(0.25)	$0.9{\cdot}10^{-3}$	(0.25)
`ham`	0.4	(0.35)	0.2	(0.34)	0.2	(0.35)	0.2	(0.36)	$\dbox{0.2^{\dagger}}$	(0.35)
`fft`	7.1	(0.37)	6.7	(0.36)	6.8	(0.37)	6.8	(0.37)	6.8	(0.37)
`factorial`	0.2	(0.20)	0.2	(0.20)	0.2	(0.20)	0.2	(0.20)	$\dbox{0.2^{\dagger}}$	(0.20)
`witt`	67.9	(1.18)	$\boxed{49.4^{\dagger}}$	(1.12)	$\dbox{\text{n/a}^{\dagger}}$	(1.15)	$\boxed{49.2^{\dagger}}$	(1.13)	$\dbox{\text{n/a}^{*}}$	(1.15)
`poly`	31.5	(0.53)	2.4	(0.36)	$\utilde{2.4^{*}}$	(0.35)	2.4	(0.38)	2.4	(0.38)
`deriv`	$0.3{\cdot}10^{-2}$	(0.30)	$0.2{\cdot}10^{-2}$	(0.29)	$0.2{\cdot}10^{-2}$	(0.29)	$0.2{\cdot}10^{-2}$	(0.29)	$\dbox{0.2{\cdot}10^{-2\dagger}}$	(0.29)
`grammar`	$0.2{\cdot}10^{-3}$	(0.23)	$0.2{\cdot}10^{-3}$	(0.23)	$0.2{\cdot}10^{-3}$	(0.23)	$0.2{\cdot}10^{-3}$	(0.24)	$0.2{\cdot}10^{-3}$	(0.23)
`fib`	0.3	(0.22)	0.3	(0.21)	0.3	(0.21)	0.3	(0.21)	0.3	(0.21)
`boyer`	17.1	(0.60)	15.1	(0.55)	15.2	(0.55)	15.0	(0.57)	$\dbox{15.5^{\dagger}}$	(0.59)
`queens`	14.6	(0.25)	12.7	(0.24)	13.2	(0.23)	12.4	(0.24)	13.2	(0.24)
`jugs`	$0.4{\cdot}10^{-2}$	(0.34)	$0.6{\cdot}10^{-3}$	(0.32)	$0.6{\cdot}10^{-3}$	(0.32)	$0.6{\cdot}10^{-3}$	(0.32)	$\dbox{0.6{\cdot}10^{-3\dagger}}$	(0.32)
`bid`	$0.5{\cdot}10^{-2}$	(0.42)	$0.4{\cdot}10^{-2}$	(0.39)	$\dbox{\text{n/a}^{\dagger}}$	(0.39)	$0.3{\cdot}10^{-2}$	(0.39)	$\dbox{\text{n/a}^{*}}$	(0.39)
`nreverse`	4.1	(0.21)	0.1	(0.22)	0.1	(0.21)	0.1	(0.21)	0.2	(0.21)
`guardians`	0.7	(0.29)	0.7	(0.29)	$\dbox{\text{n/a}^{\dagger}}$	(0.30)	0.7	(0.29)	$\dbox{\text{n/a}^{*}}$	(0.23)
`crypt`	0.1	(0.33)	$0.9{\cdot}10^{-2}$	(71.83)	0.1	(0.32)	$0.1{\cdot}10^{-1}$	(71.32)	$\dbox{0.8{\cdot}10^{-1\dagger}}$	(0.32)
`exfilter`	87.1	(15.31)	85.4	(20.72)	86.3	(18.59)	84.5	(20.40)	$\dbox{\text{n/a}^{*}}$	(15.25)
`deepfind`	1.5	(15.69)	1.8	(419.87)	2.1	(28.18)	1.0	(3.19)	$\dbox{\text{n/a}^{*}}$	(3.20)
`chat-80`	0.6	(60.55)	0.2	(5.77)	0.2	(5.82)	0.1	(5.92)	$\dbox{\text{n/a}^{*}}$	(5.39)
`LPdoc`	51.6	(300.92)	0.7	(73.90)	0.8	(179.85)	0.7	(74.94)	$\dbox{\text{n/a}^{*}}$	(24.83)
`Spectector`	24.7	(6.02)	3.7	(4.35)	3.8	(4.69)	1.7	(4.32)	$\dbox{\text{n/a}^{*}}$	(3.28)
`s(CASP)`	56.0	(328.94)	10.1	(25.94)	$\dbox{\text{n/a}^{\dagger}}$	(24.37)	8.0	(68.63)	$\dbox{\text{n/a}^{*}}$	(50.10)

Classes of bugs found are marked with: $\dbox{\ }$ = abstract domain implementation, $\dots$ = fixpoint algorithms, $\boxed{\ }$ = semantic inconsistencies, $\utilde{\ }$ = run-time check instrumentation, $\underline{\underline{\ }}$ = third-party libraries.

No bugs were found for the most mature domains. On the other hand, bugs were indeed found in the more experimental and prototype domains, that is domains which were only partially developed and/or they or their run-time tests supported only a subset of the language at the time of the experiments (e.g., the *polyhedra* abstract domain). This included for example no support being available for certain built-in operations, or

Table 6. *Timings and errors detected (4)*

	Absolute run time, s (Analysis time, s)									
Program	*eterms*		*polyhedra*		*depth-k*		*det*		*nfg*	
`mmatrix`	0.6	(0.64)	$0.2{\cdot}10^{-1}$	(0.25)	$\uwave{0.1{\cdot}10^{-1}}^{\dagger}$	(0.26)	0.3	(0.40)	0.3	(0.37)
`qsort`	4.0	(0.53)	n/a	(n/a)	$\uwave{0.2{\cdot}10^{-1}}^{\dagger}$	(0.21)	4.4	(0.47)	4.4	(0.46)
`exp`	0.1	(0.27)	$0.4{\cdot}10^{-1}$	(0.25)	$0.2{\cdot}10^{-1}$	(0.22)	0.2	(0.20)	0.2	(0.23)
`aiakl`	$\underline{\underline{0.8{\cdot}10^{-3}}}^{\dagger}$	(0.48)	$\boxed{\text{n/a}^{\dagger}}$	(0.23)	$0.3{\cdot}10^{-4}$	(0.25)	$0.1{\cdot}10^{-3}$	(0.51)	$0.1{\cdot}10^{-2}$	(0.39)
`ham`	7.6	(0.48)	$\boxed{\text{n/a}^{\dagger}}$	(0.35)	$\uwave{0.5{\cdot}10^{-1}}^{\dagger}$	(0.33)	$0.8{\cdot}10^{-1}$	(0.35)	$0.8{\cdot}10^{-1}$	(0.27)
`fft`	$\underline{\underline{9.8}}^{\dagger}$	(3.00)	$0.5{\cdot}10^{-2}$	(0.25)	$\uwave{0.3}^{\dagger}$	(0.32)	n/a	(4.34)	13.2	(3.46)
`factorial`	0.2	(0.22)	$0.7{\cdot}10^{-1}$	(0.21)	$0.2{\cdot}10^{-1}$	(0.17)	0.2	(0.18)	0.3	(0.19)
`witt`	$\boxed{\text{n/a}^{*}}$	(0.48)	n/a	(n/a)	n/a	(n/a)	$\boxed{\text{n/a}^{\dagger}}$	(0.35)	n/a	(n/a)
`poly`	$\underline{\underline{27.8}}^{\dagger}$	(15.68)	0.2	(0.25)	0.5	(0.33)	$\boxed{\text{n/a}^{\dagger}}$	(32.70)	n/a	(n/a)
`deriv`	0.2	(2,384.47)	$0.2{\cdot}10^{-3}$	(0.29)	$0.1{\cdot}10^{-3}$	(0.30)	n/a	n/a	n/a	(n/a)
`grammar`	$0.1{\cdot}10^{-3}$	(0.37)	n/a	(n/a)	$\uwave{0.4{\cdot}10^{-4}}^{\dagger}$	(0.21)	$0.1{\cdot}10^{-3}$	(0.26)	$0.2{\cdot}10^{-3}$	(0.27)
`fib`	0.2	(0.27)	0.1	(0.23)	$0.1{\cdot}10^{-1}$	(0.20)	0.3	(0.19)	0.6	(0.20)

Table 6. *Continued*

Program	Absolute run time, s (Analysis time, s)									
	eterms		*polyhedra*		*depth-k*		*det*		*nfg*	
`boyer`	9.0	(6.41)	$\boxed{\text{n/a}^{\dagger}}$	(0.37)	0.4	(0.56)	55.0	(7.98)	n/a	(n/a)
`queens`	21.1	(0.36)	1.3	(0.25)	$\utilde{1.0^{\dagger}}$	(0.23)	26.7	(0.26)	27.3	(0.26)
`jugs`	$\underline{\underline{1.6^{\dagger}}}$	(0.82)	$\boxed{\text{n/a}^{\dagger}}$	(0.28)	$\utilde{0.1 \cdot 10^{-3\,\dagger}}$	(0.33)	$\boxed{\text{n/a}^{\dagger}}$	(0.77)	$\boxed{\text{n/a}^{\dagger}}$	(0.50)
`bid`	$\underline{\underline{0.1 \cdot 10^{-1\,\dagger}}}$	(1.46)	n/a	(n/a)	$\utilde{0.3 \cdot 10^{-3\,\dagger}}$	(0.35)	$0.1 \cdot 10^{-4}$	(0.91)	$0.1 \cdot 10^{-4}$	(0.95)
`nreverse`	3.1	(0.30)	$0.2 \cdot 10^{-1}$	(0.22)	$\utilde{0.7 \cdot 10^{-1\,\dagger}}$	(0.21)	20.6	(0.27)	25.3	(0.30)
`guardians`	$\underline{\underline{15.0^{\dagger}}}$	(0.70)	$0.1 \cdot 10^{-1}$	(0.32)	$0.8 \cdot 10^{-2}$	(0.27)	48.6	(0.47)	59.5	(0.55)
`crypt`	$\underline{\underline{0.2 \cdot 10^{-2\,\dagger}}}$	(1.15)	$0.1 \cdot 10^{-1}$	(54.92)	$\utilde{0.4 \cdot 10^{-2\,\dagger}}$	(0.29)	0.1	(0.79)	0.2	(0.72)
`exfilter`	35.5	(9.25)	$\boxed{\text{n/a}^{*}}$	(3.71)	n/a	(n/a)	n/a	(n/a)	n/a	(n/a)
`deepfind`	$\underline{\underline{\text{n/a}^{\dagger}}}$	(43.48)	n/a	(n/a)	0.2	(5.01)	$\underline{\underline{\text{n/a}^{*}}}$	(675.89)	$\underline{\underline{\text{n/a}^{*}}}$	(676.56)
`chat-80`	$\boxed{\text{n/a}^{*}}$	(334.72)	$0.2 \cdot 10^{-1}$	(1,202.55)	$\boxed{\text{n/a}^{*}}$	(1,473.90)	$\boxed{\text{n/a}^{\dagger\dagger}}$	(444.72)	$\boxed{\text{n/a}^{\dagger\dagger}}$	(443.65)
`LPdoc`	n/a	n/a	$\boxed{\text{n/a}^{\dagger}}$	(15.46)	$\boxed{\text{n/a}^{*}}$	(19.56)	n/a	n/a	n/a	n/a
`Spectector`	$\underline{\underline{\text{n/a}^{\dagger}}}$	(4.91)	$\boxed{\text{n/a}^{\dagger}}$	(3.97)	$\underline{\underline{\text{n/a}^{\dagger\dagger}}}$	(4.25)	$\underline{\underline{\text{n/a}^{*}}}$	(5.99)	$\underline{\underline{\text{n/a}^{*}}}$	(6.55)
`s(CASP)`	$\boxed{\text{n/a}^{*}}$	(90.18)	$\boxed{\text{n/a}^{*}}$	(17.13)	$\underline{\underline{\text{n/a}^{**}}}$	(20.43)	$\boxed{\text{n/a}^{**}}$	(323.73)	$\boxed{\text{n/a}^{**}}$	(271.79)

Classes of bugs found are marked with: $\boxed{}$ = abstract domain implementation, . . . = fixpoint algorithms, $\boxed{}$ = semantic inconsistencies, $\sim$ = run-time check instrumentation, $=$ = third-party libraries.

Table 7. *Details of defects found. The root causes of bugs (Class) again include: (I) abstract domain implementation; (II) fixpoint algorithms; (III) semantic inconsistencies; (IV) run-time check instrumentation; (V) third-party libraries*

#	Status	Class	Description
1	New	I	Analysis does not evaluate correctly the `arg(X,Y,Z)` predicate due to absence of an abstract description in the *def* domain for the case when `Z` is ground.
2	New	III	The name *covered* is used in two different properties with different semantics.
3	New	IV	Incorrect run-time semantics for property `mshare/1` due to being sensitive to variable ordering.
4	New	I	Analysis with the *shfr+nonvar* domain of a program containing `length(X,Y)`, where `X` is not a variable marks `length/2` as failing.
5	New	I	Abstract definition of the `\=/2` built-in not implemented correctly in the *shfr+nonvar*, *shfr+clique*, and *shfr+clique+def* domains.
6	New	I	Abstract definition of the `\==/2` built-in not implemented correctly in the *shfr+nonvar*, *shfr+clique*, and *shfr+clique+def* domains.
7	New	III	The analysis and run-time semantics of `linear/1` are inconsistent.
8	New	IV	The `mshare/1` implementation for run-time checking considers all variables at a program point. This becomes problematic when both `clique/1` and `mshare/1` are present at the same program point. When variables are “transferred” from `mshare/1` to `clique/1`, `mshare/1` should not consider these variables.
9	New	IV	Missing `clique/1` property implementation for run-time checking.
10	New	I	When analyzing with *shfr+clique+def*, the output introduces spurious variables when inferring `ground/1`.
11	Known	I	Problem in types domains with `findall/3` calls.
12	Known	I	Problem in types domains with `setof/3` calls.
13	New	III	Run-time checking instrumentation only understands parametric types that use type symbols as arguments.
14	New	V	Error in type inference due to skipping testing whether predicates declared as regular types are indeed regular types.
15	New	V	Error in *polyhedra* analysis due to not checking whether variables are numeric.
16	New	IV	Run-time implementation of *polyhedra* properties (`constraint/1`) must check if variables are instantiated: merely evaluating the (in)equality is not sufficient.
17	New	I	The *greatest lower bound* (`glb/3`) operation is not defined correctly in *depth-k* abstract domain.
18	New	IV	Missing run-time check implementation for `instance/2`.
19	Known	I	Type analysis failing to generate types for data/dynamic predicates.
20	Known	I	`=/2` built-in not defined correctly in *polyhedra* abstract domain.
21	New	II	Bug introduced in the fixpoint algorithm during code refactoring to incorporate a new transformation aimed at optimizing set-sharing-based analyses.

no run-time behavior being defined for some properties used by such domains. All this has greatly helped complete and strengthen these less mature domains, since most of the bugs found have now been fixed. One thing to take into account when reading the results in the tables is that sometimes many reported errors are due to just one bug. For example, almost all errors reported with the *shfr+clique+def* domain refer to the same bug found that affected the analysis of several benchmarks (see the markings in the tables mentioned before for other examples). Also, it is important to note that each program detects a maximum of one bug at a time, since the process halts when it detects a run-time error.

Another conclusion from the experiments is that benchmark selection is very important when testing specific domains, since each example uses different *built-ins* and library predicates, exhibits different properties, etc. Nevertheless, while complex examples such as `witt` have resulted most capable at identifying a wide range of bugs across different domains, a drawback is that for very complex benchmarks the analysis using the least mature domains is more prone to fail, and then sometimes no bug is detected.

As mentioned before, in addition to standard benchmarks, we have also applied our tool to real-life applications. In this case, rather than generating test cases, the experiment consists in compiling the application using all the source files as transformed by the algorithm and using the application as usual, for example for the LPdoc documenter, generating with it full manuals.

An interesting observation is that the benchmark program suite showed similar effectiveness to the real-world applications in exposing errors, that is most bugs identified in real applications had already been detected by the set of smaller benchmarks.

Also, the presence of duplicate bugs is indicative that the algorithm can identify problems consistently.

Finally, the fact that some bugs have already been fixed suggests that identifying the source of the detected errors in the implementation is relatively straightforward.

4.3 Further discussion of the bugs detected

This section illustrates further some of the defects found by the proposed technique during our experiments reported in Section 4.1. The programs presented are representative of the actual code fragments that triggered the detection of the bugs; however, we generally show simplified and distilled versions for brevity and clarity, since the actual code would need significant context to be understood. We divide the discussion according to the different classes of bugs (I to V) from Section 4.1. The bug numbers refer to Table 7.

4.3.1 Abstract domain implementation (Class I)

As mentioned 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. 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 a long time. Thus, it makes sense to start by taking this as the trusted base and aiming the error-finding task at the different abstract domains. This makes sense specially since CiaoPP is at the same time a production and a research tool, and new domains are constantly being developed.

Example 4.1
(Bug #4: Missing assertion in library predicate description). Abstract descriptions of the behavior of the built-ins and library predicates are provided for each abstract domain, either in the file(s) defining the domain, in the libraries themselves, or in both. A first use of our algorithm is in order to find errors in these specifications and in the implementations of these built-ins and library predicates for a given abstract domain. In particular, if during testing a run-time error is found in a program-point assertion right after a call to a built-in, this can be due to an error in either the abstract description of that builtin or in its actual implementation. For instance, the code of the `length/2` library predicate includes the following (simplified) assertions:

```
:- pred length(+list,-int) + det.
:- pred length(-list,+int) + det.
:- pred length(+list,+int) + semidet.
:- pred length(?,?) + nondet.
```

where `int/1` is a primitive property (in this case a primitive type) and `list/1` is defined in a library by the standard list predicate, and also declared to be a property (in particular, a regular type) in the usual way:

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

During the tests, in a call to `length/2`, illustrated by the following code:

```
  :- entry p(+list(num),+num,-).

  p(X,N,Y) :-
      length([N|X],Y).
```

that is a call with the second argument uninstantiated, the following analysis output was obtained:

```
  :- entry p(+list(num),+num,-int).

  p(X,N,Y) :-
      true(var(Y), ground([X,N])),
      length([N|X],Y),
      true(unreachable).
```

that is the abstract interpreter (wrongly) inferred failure or error in the call to `length/2` and thus that the point after that would be unreachable. The checkification algorithm translated this output into:

```
  :- entry p(+list(num),+num,-int).

  p(X,N,Y) :-
      check(var(Y), ground([X,N])),
      length([N|X],Y),
      check(unreachable).
```

During run-time testing the `check(unreachable)` literal was actually reached and executed, which threw the corresponding error.

This led to detecting that the last declaration in the abstract description for `length/2` (i.e., "`:- pred length(?,?) + nondet.`" in the set of assertions for `length/2` above), had been deleted by mistake while refactoring some analyzer code. That assertion is the only one that allows calling `length/2` with a variable in the second argument (in fact, in both arguments).

Example 4.2
(`Bug #10`: Combined abstract domain outputs unknown variables). While experimenting with the CLIQUE-Sharing+Freeness+Def combined domain an error was detected which can be illustrated with the following code:

```
 :- entry p(+num,-).

p(X,Y) :-
    Y=X.
```

for which `CiaoPP` inferred the following information:

```
 :- entry p(+num,-).

p(X,Y) :-
     true(ground([X,_A])),
     Y=X,
     true(ground([X,Y,_B])).
```

where `_A` and `_B` were superfluous variables that have the first occurrence in the `true(ground(...))` literals, and thus cannot be ground. A run-time error was thus produced by the checkified program in the `check(ground([X,_A]))` call, which made us realize that the CLIQUE-Sharing+Freeness+Def abstract domain was not implementing correctly the combination of analyses. In many combined domains, the combination typically reuses the component analyses. For example, all abstract functions for CLIQUE-Sharing+Freeness+Def initially compute results for the sharefree_clique and def functions. Then, they compose the information from the different domains, eliminating redundancies over the information inferred by each analysis Codish et al. (1995). This is performed by the reduce functions, which essentially compute the reduced product. The problem detected stemmed from an accidental reversal of the sequence of operations in the implementation. At first, the abstract function of def was executed. However, the reduce function was applied before running the abstract function of sharefree_clique. The problem with this is that in some abstract functions a renaming of variables is performed. This renaming substitution replaces each variable in the term it is applied to with distinct fresh variables. If the reduce function is applied before the renaming substitution, it incorrectly treats some variables as distinct when, in fact, they are identical before renaming.

4.3.2 Fixpoint algorithms (Class II)

Another possible application of the approach is for testing the abstract interpretation engine (the *fixpoint algorithms* and all the surrounding infrastructure of the framework) instead of the domains. This can be done by using domains that are simple or developed enough to be used as a trusted base. While the classic fixpoint algorithms are quite stable, new fixpoint algorithms or modifications of existing fixpoint algorithms are sometimes added to the system. Some recent examples include a new modular and incremental fixpoint (Garcia-Contreras *et al.* 2021) and new program transformations that speed-up set sharing-based analyses by reducing the number of variables in abstractions (Jurjo *et al.* 2024). Clearly, these new contributions may introduce new bugs into the system.

A first abstract domain that is useful for this type of checks is the *concrete domain* itself. To this end, we give the analysis a singleton set of initial states as entry point, that is a concrete value, and the analyzer then behaves as a (tabling) interpreter for the program, starting from the entry point as initial concrete state. This test will then detect if the analyzer incorrectly marks reachable parts of the program as unreachable.

Example 4.3
(`Bug #21`: Code refactoring). A code refactoring in the implementation of Jurjo et al. (2024) introduced a bug in the handling of built-ins and cuts. While no issues were identified in the initial version of the code, the refactored version (where most changes involved renaming operations and variables) revealed a problem when reapplying the algorithm.[9] Specifically, the analyzer incorrectly inferred that some predicates containing built-ins and cuts were dead code, although they were indeed reachable in the concrete domain.

4.3.3 Semantic inconsistencies between components of the framework (Class III)

Even if every part of the system is validated separately, our tool can still help find inconsistencies among these parts. Most components in our system interact via assertions and thus the semantics of the assertions and of the properties used in the assertions need to be consistent across all parts. An interesting case that can occur is when there is a mismatch between the way properties are understood by the analyzer and the actual definition of the property that is used in the run-time checks. Checkification helps detect such inconsistencies.

Example 4.4
(`Bug #7`: Wrong `linear/1` run-time semantics). When the analyzer outputs `linear(X)`, the semantics is that `X` is the list of all program variables that analysis can guarantee to be bound to linear terms at the program point, that is that the terms that each variable in `X` is bound to do not contain any repeated variables. The property was inferred correctly by `CiaoPP`. In particular, for a substitution such as {`X/f(A,B), Y/g(A,C)`}, `CiaoPP` was inferring correctly `true(linear([X,Y]))`. However, after the checkification conversion to `check(linear([X,Y]))`, a run-time error was being thrown. This allowed us to notice that in the implementation of the property as a run-time check the wrong predicate had been used and what was checked instead of linearity was that the terms in the list did not share any variable, which then failed in this case since they share variable `A`.

Example 4.5
(`Bug #13`: Different representation of parametric types). This problem was found when analyzing with type domains. Assume that the analyzer has inferred, using, for example

[9] The technique has been included as a fuzz testing component of the `Ciao` system. While the overhead is relatively acceptable for fuzzing tasks, it can be less suitable for continuous integration, which requires building tests after each commit.

the `eterms` domain, that after success of a call to `p(X)`, the argument `X` is bound to a list of `a`'s. This is expressed in the output as:

```
...,
p(X)
true(list(rt1, Xs)).

:- regtype rt1/1.
rt1(a).
```

This posed no problems with the run-time checking. However, since this style of output can sometimes be verbose, for readability the user can optionally switch the output so that simple cases are expressed using quoting (`^/1`) as follows:

```
...,
p(X)
true(list(^(a), Xs)).
```

At some point the output had been switched to this format by default and then we discovered that when executing the `check(list(^(a), Xs))` the run-time check instrumentation did not implement this abridged syntax correctly. This was another inconsistency between a possible representation(s) of the abstraction and what was understood by the run-time checks.

4.3.4 Inconsistencies between properties and their specialized run-time checks (Class IV)

This is a special case of the previous class, related to the fact that it is possible to write an alternative version of a property to be used specifically in run-time checks, while also keeping the general definition which may perhaps be easier to read or to be understood by the static analyzer.

Example 4.6
(`Bug #9` and `#18`: Undefined run-time behavior). In a few instances test case failures stemmed from the specialized run-time behavior of some properties simply being declared but undefined. For example, testing flagged that the implementations for run-time checking of the comparatively less-used `clique/1` and `instance/2` properties were missing.

Example 4.7
(`Bug #8`: Incompatibility between run-time behaviors). The approach also detected more subtle issues that only arise when several properties appear together. While addressing the issue of the previous example, another problem was detected when the `clique/1` and `mshare/2` properties appeared in the same assertion. Set sharing domains approximate all possible variable sharing (aliasing) that occurs at a given program point. The property typically used to denote such sharing among variables is `mshare/1`, where the argument contains a set of sets of variables. For instance, let $\mathtt{V} = \{\mathtt{X,\ Y,\ Z,\ W}\}$ be the set of variables of interest, normally the variables of the clause. Consider the abstraction $\lambda = \{\{\mathtt{X}\},\ \{\mathtt{X,Y}\},\ \{\mathtt{X,Y,Z}\},\ \{\mathtt{X,Z}\},\ \{\mathtt{Y}\},\ \{\mathtt{Y,Z}\},\ \{\mathtt{Z}\},\ \{\mathtt{W}\}\}$. Here a set $\{\mathtt{X,Y}\}$ represents that, in the terms that `X` and `Y` are bound to at run time, there may be variables that appear in both `X` and `Y`; $\{\mathtt{W}\}$ that there may be a variable that appears only in

`W`; etc. This λ will be expressed in the analyzer output as: `true(mshare([[X], [X,Y], [X,Y,Z], [X,Z], [Y], [Y,Z], [Z], [W]]))`. We can see that for variables {`X, Y, Z`} no information (i.e., top) has been inferred. Indeed for this set of variables any aliasing may be possible since there may be run-time variables shared by any pair of the three program variables, by the three of them, or not shared at all. The idea of the clique property is to use a more compact representation for abstractions that contain a powerset, including as a widening when abstractions are too large. This has been shown to pay off in practice (Navas et al. 2006). In our example, the clique that will convey the same information with respect to {`X, Y, Z`} as the sharing set $S = \wp(\{X, Y, Z\})$ is simply `clique([X,Y,Z])`. The elements of S are then eliminated from the full sharing set λ, since the clique makes them redundant. However, when checkifying the output, it was detected that the specialized `mshare/1` run-time checking implementation was still considering all the variables in the clause ({`X, Y, Z, W`} in our case). Therefore, `X`, `Y`, and `Z`, were incorrectly being tested for groundness since they are in no sharing set, and thus are interpreted as being ground. Clearly, this is not what the analysis inferred. The `mshare/1` run-time test was modified to stop considering such variables.

Example 4.8
(`Bug #3`: `mshare/1` sensitivity to variable ordering). This case involved a subtle bug in the specialized run-time check for the sharing abstract domain. As mentioned before, `mshare([X,Y])` means that `X` may share variables with `Y` at run time. Sharing is a symmetric property which does not depend on variable ordering, that is `mshare([X,Y])` has the same meaning as `mshare([Y,X])`. Some cases were detected in which a run-time checking error was flagged even though the `mshare/1` property inferred was correct. The problem was found in an update of the specialized run-time definition of `mshare/1`, which introduced an optimization which made the result sensitive to the ordering of variables.

Example 4.9
(`Bug #16`: Incorrect definition of `constraint/1`). The `constraint/1` property is used by some numerical (e.g., polyhedra-related) domains. Its argument is a list of linear (in)equalities that relate variables and integer values. However, the specialized run-time check of this property was incorrect: it checked that the constraints were valid but also whether the arguments were variables; instead it should have been checking that they were numerical values.

4.3.5 Integration testing of the analyzer with libraries and third-party tools (Class V)

We have also used the approach to conduct integration tests of the *Regular Types Library* (an independent `Ciao` bundle) and check its correct use within the system. This library implements fundamental operations and procedures for regular types, such as type inclusion, equivalence, union, intersection, widening, simplification, etc., as well as storing and manipulating regular types. These operations are used in `CiaoPP` domains, analyses, and program transformations.

Example 4.10
(`Bug #14`: Regular types library type equivalence/simplification bug). The `eterms` domain includes the inferred regular types in the analysis output. In this process, `eterms`

performs type equivalence and simplification operations in order to present the results to users in the most readable form possible. The following (simplified) fragment is from one of the test programs:

```
:- prop repeat_elem_list/1 + regtype.

repeat_elem_list([]).
repeat_elem_list([X,X|Xs]) :-
    num(X),
    repeat_elem_list(Xs).

:- pred diff_elem_list(X) : list(num,X) .

diff_elem_list([]).
diff_elem_list([X,Y|Xs]) :-
    X =\= Y,
    diff_elem_list(Xs).
```

Here `repeat_elem_list/1` is (incorrectly!) declared as a regular type. Checkification detected that, in the analysis output, eterms erroneously inferred that `diff_elem_list/1` produces a `repeat_elem_list/1` upon success:

```
...
diff_elem_list([]).
diff_elem_list([X,Y|Xs]) :-
    check(num(X), num(Y), list(num,Xs)),
    X=\=Y,
    check(num(X), num(Y), list(num,Xs)),
    diff_elem_list(Xs),
    check(num(X), num(Y), repeat_elem_list(Xs)).
```

This inference is obviously incorrect. This allowed us to notice that after some changes the Regular Types Library was failing to check if properties declared to be regular types were actually regular types, which consequently affected the inference of types.

In addition to detecting problems due to the integration with libraries, the checkification approach also detected incorrectness due to the integration with different external or third party solvers which can be used by the analyzer. For example, the *polyhedra* domain uses the *Parma Polyhedra Library* (PPL) as back-end solver for the handling of numeric approximations. Using this domain we can detect errors stemming from the `Ciao`-PPL integration.

Example 4.11
(`Bug #15`: Parma Polyhedra Library). We found that polyhedra did not properly handle some non-numerical parts of programs. A term like `X = Y` is translated into the constraint $1 \times \mathtt{X} - 1 \times \mathtt{Y} = 0$ when analyzing with the abstract domain. If `X` and `Y` are numerical variables, they satisfy the equality dynamically. However, we discovered that the analysis did not take into account the types of `X` and `Y`, that is regardless of whether `X` and `Y` were numerical or non-numerical, the analysis treated them as numerical values, not considering their actual types.

4.3.6 Debugging trust assertions and custom transfer functions

Finally, the approach has also been useful in finding errors in other aspects of the framework. As an example, a feature of `CiaoPP` is that its analyses can be guided by the user by providing the analyzer with information that can be assumed to be true at points where otherwise the analysis would lose precision. We have already mentioned one of these mechanisms, *entry* assertions, which allow providing information on the entry points to the module being analyzed (i.e., on the calls to the predicates exported by the module). Entry assertions are a special case of *trust* assertions. In addition to guiding the analyzer, *trust* assertions are used to define custom abstract transfer functions, similar to those that need to be implemented for abstracting each *built-in* within each domain. *trust* assertions allow the user to do this for any predicate. Checkification can be used to detect errors in these assertions.

Example 4.12
(Defining language semantics). Let us consider this (again, distilled) part of the specification of the exponentiation predicate, where we use the trust assertion in line 1 to state that an integer to the power of an integer yields an integer:

```
1  :- trust pred exp_op(+int,+int,-int).
2
3  exp_op(X,Y,Z) :- Z is ‘**’(X,Y).
4
5  exp_list([],_Y,[]).
6  exp_list([X|Xs],Y,[Z|Zs]) :- exp_op(X,Y,Z), exp_list(Xs,Y,Zs).
```

However, in the ISO-Prolog standard, the outcome of this operation is always a floating-point number. When analyzing this code, `CiaoPP` trusts the assertion previously defined and infers that the result of the operation in the `exp_list/3` predicate is an integer. In the checkified version:

```
...
exp_list([],_Y,[]).
exp_list([X|Xs],Y,[Z|Zs]) :-
    exp_op(X,Y,Z),
    check(int(X), int(Y), int(Z)),
    exp_list(Xs,Y,Zs).
```

when we execute a test case with two integers an error is flagged due to this discrepancy, since the implementation correctly produces a float. This is a practical application of the proposed algorithm since even a completely sound analyzer can produce unsound results if it assumes some assertion to be true when it is not, and thus there will always be the need to test such properties.

In addition to being able to detect errors in *trust* assertions, the approach can also be used to find errors in the mechanisms used internally by the analyzer to apply *trust* assertions, if we assume instead the information provided in the *trust* assertion to be correct.

5 Other related work

In this section we mention other related work in addition to the references interspersed throughout the previous sections. The fact that the reliability of program analyzers has become crucial as they have become increasingly practical and widely adopted in recent years, is now widely recognized (Cadar and Donaldson, 2016). This has led to significant research interest recently.

On the formal verification side, there have been some pen-and-paper proofs, such as that of the Astrée analyzer (Cousot *et al.* 2005), some automatic and interactive proofs, such as Dubois (2000); Shao *et al.* (2002). As mentioned in the introduction, a recent relevant effort has been aimed at verifying the partial correctness of the `CiaoPP` analysis algorithm (also referred to as the "top-down solver"), using the Isabelle prover (Paulson, 1990; Stade *et al.* 2024), but mechanical verification of the actual implementations remains a challenge. An approximation to this problem is Blazy *et al.* (2013) and Jourdan *et al.* (2015) who have developed and proved the soundness of a static analyzer, extracting several abstract domains and a fixed-point verifier directly from Coq formalizations.

In the context of testing static analyzers significant work has been done. Midtgaard and Møller (2017) and Bugariu *et al.* (2018) use mathematical properties of abstract domains as test oracles. Specifically, Bugariu *et al.* (2018) employ such properties to validate the soundness and precision of numerical abstract domains, while Midtgaard and Møller (2017) apply that approach to checking type analysis using QuickCheck (Claessen and Hughes 2000).

The closest works to ours are those that cross-check dynamically observed and statically inferred properties (Cuoq *et al.* 2012; Wu *et al.* 2013; Andreasen *et al.* 2017; Zhang *et al.* 2019). In Wu *et al.* (2013) the actual pointer aliasing in concrete executions is cross-checked with the pointer aliasing inferred by an aliasing analyzer. They are also able to deal with multi-variance and path-sensitivity. 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 Zhang *et al*. (2019) 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 Cuoq *et al.* (2012) a wide range of static analysis tests are performed over randomly generated programs as input. Among others, they check dynamically, at the end of the program, one assertion inferred statically, and they perform the sanity check to ensure that the analyzer behaves as an interpreter when run from a singleton set of initial states. Our approach differs from these previous works by identifying issues throughout the entire static analyzer framework, rather than focusing on specific components. In this setting, Klinger *et al.* (2019) propose an automatic technique to evaluate soundness of program analyzers based on differential testing. From seed programs, they generate program analysis benchmarks and compare the overall frameworks of software model checkers. Differential analysis presents a significant challenge, requiring multiple analyzers with identical input/output behavior to be applied. A potential problem arises if one or more analyzers behave differently, making it difficult to determine which is correct. He *et al.* (2024) apply two oracles to compare static analyzers. The first oracle is constructed using dynamic program executions. A random program generated by Csmith Lidbury *et al.* (2015), a generation-based C fuzzer, serves as test input, and its dynamic execution result (e.g., a run-time error on some program path) is used as a test oracle to validate the static analyzer. The second oracle is a static oracle based on metamorphic relations. The approach involves selecting a conditional expression of some statement in the program as a target and generating an equivalent boolean expression. The static analyzer should produce identical truth values for these equivalent expressions, any discrepancy indicates a potential defect in the analyzer.

There is work on testing program analyzers in other domains. For instance, Brummayer and Biere (2009) introduce a grammar-based fuzzer to identify crashes in SMT solvers, using delta debugging to minimize generated instances. Additionally, Kapus and Cadar (2017) combine random program generation with differential testing to evaluate symbolic execution engines. Daniel *et al.* (2007) implement Java programs as test oracles for abstract syntax trees for testing refactoring engines. There is also considerable work in testing compilers. Le et al. (2014); Le *et al.* (2015); Sun *et al.* (2016) apply Equivalence Modulo Inputs testing, which involves mutating unexecuted statements of an existing program under certain inputs to produce new equivalent test programs. Regehr *et al.* (2012) propose a test-case reduction technique to construct minimal inputs that trigger compiler bugs. Finally, Building on Csmith Yang *et al.* (2011), Lidbury *et al.* (2015) introduce CLsmith, which designs six modes to generate test programs.

We argue that, in comparison with these related proposals, our approach provides a solution that is simpler, elegant, more general, and easier to implement, specially when the different system components are integrated as in the Ciao assertion model.

6 Conclusions

We have proposed and studied *checkification*, an automatic method for testing static analysis tools by checking that the properties inferred statically are satisfied dynamically. A fundamental strength of our approach lies in its simplicity, which stems from framing it within the `Ciao` assertion model and using its assertion language. We have shown how checkification can be implemented with comparatively little effort by combining the static analyzer, the run-time checking framework, the random test case generator, and the unit-test framework, together with a reduced amount of glue code. This code implements the proposed algorithm, and pilots the combination and interplay of the intervening components to effectively implement the overall approach.

Following this approach, we have constructed a quite complete implementation of the checkification method and applied it to testing a large number of the abstract interpretation-based analyses in `CiaoPP`, representing different levels of code maturity, as well as to the framework itself, the interaction with libraries, the interaction with third-party code, etc. In the study we analyzed both standard benchmarks and real-world tools. We applied our technique not only to state properties such as variable sharing/aliasing, modes, linearity, numerical properties, types, and term structure, but also to computational properties such as determinacy and (non)failure. As we discussed, the technique should also be applicable to other, more complex, computational properties such as cost and even termination, although they obviously bring in specific challenges and theoretical limits for run-time checking.

The experimental results show that our tool can effectively discover and locate interesting, non-trivial and previously undetected bugs, with reasonable overhead, not only in the less-developed parts of the system but also in corner cases of the more mature components, such as the handling of *built-ins*, run-time checking instrumentation, etc. The approach has also proven useful for detecting issues in auxiliary stages of analysis and verification, including assertion simplification, pretty printing, abstract program optimizations and transformations, etc. We have also observed that it is generally not too hard to locate the source of the errors from the information produced by the tests, and as a result the vast majority of the detected issues were in fact fixed during the experimental study. While testing approaches are obviously ultimately insufficient for proving the correctness of analyzers, and thus it is clearly worthwhile to also pursue the avenue of code verification, we believe that our results show that the checkification approach can be a practical and effective technique for detecting errors in large and complex analysis tools.

Acknowledgements

Partially funded by MICINN projects PID2019-108528RB-C21 *ProCode*, TED2021-132464B-I00 *PRODIGY*, and FJC2021-047102-I, and by the Tezos foundation. We would also like to thank the anonymous reviewers for their very useful feedback. This paper is an extended version of our previous work published in LOPSTR'20.

References

Andreasen, E. S., Møller, A. and Nielsen, B. B. 2017. 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, 2017, SOAP, 2017. Association for Computing Machinery, New York, NY, USA, 31–36.

Arias, J., Carro, M., Salazar, E., Marple, K. and Gupta, G. 2018. Constraint answer set programming without grounding. *Theory and Practice of Logic Programming* 18, 3–4, 337–354.

Bagnara, R., Ricci, E., Zaffanella, E. and Hill, P. M. 2002. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In Static Analysis: Proceedings of the 9th International Symposium, 2002. M. V. Hermenegildo and G. Puebla, Eds. volume 2477 of Lecture Notes in Computer Science, Springer-Verlag, Madrid, Spain, Berlin, 213–229.

Blazy, S., Laporte, V., Maroneze, A. and Pichardie, D. (2013) Formal verification of a C value analysis based on abstract interpretation. In *Static Analysis*, F. Logozzo and M. Fähndrich, Eds. Berlin, Heidelberg, Berlin Heidelberg, Springer, 324–344.

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

Bueno, F., Lopez-Garcia, P. and Hermenegildo, M. V. 2004. Multivariant non-failure analysis via standard abstract interpretation. In 7th Int'l. Symposium on Functional and Logic Programming 2004, volume 2998 of LNCS, Springer-Verlag, 100–116.

Bueno, F., Lopez-Garcia, P., Puebla, G. and Hermenegildo, M. V. (2006). 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.

Bugariu, A., Wüstholz, V., Christakis, M. and Müller, P. 2018. Automatically testing implementations of numerical abstract domains. In Proceedings of the 33rd ACM/IEEE

International Conference on Automated Software Engineering 2018, ASE 2018. Association for Computing Machinery, New York, NY, USA, 768–778.

Cadar, C. and Donaldson, A. 2016. Analysing the program analyser. In International Conference on Software Engineering, Visions of 2025 and Beyond Track (ICSE V2025), Association for Computing Machinery. 2016, 765–768.

Casso, I., Morales, J. F., Lopez-Garcia, P. and Hermenegildo, M. V. 2020. An integrated approach to assertion-based random testing in prolog. In Post-Proceedings of the 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'19), M. Gabbrielli, Ed. volume 12042 of LNCS, Springer-Verlag, 159–176.

Claessen, K. and Hughes, J. 2000. QuickCheck: A lightweight tool for random testing of Haskell programs. In Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming 2000, ICFP'00, ACM, New York, NY, USA, 268–279.

Codish, M., Mulkers, A., Bruynooghe, M., García de la Banda, M. and Hermenegildo, M. 1993. Improving abstract interpretations by combining domains. In Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation 1993, ACM, 194–206.

Codish, M., Mulkers, A., Bruynooghe, M., García de la Banda, M. and Hermenegildo, M. 1995. Improving abstract interpretations by combining domains. *ACM Transactions on Programming Languages and Systems* 17, 1, 28–44.

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D. and Rival, X. 2005. The ASTRéE analyzer. In 14th European Symposium on Programming, ESOP 2005, held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Vol. 3444, 21–30, 2005; Springer-Verlag. Conference date: 04-04-2005 Through 08-04-2005.

Cuoq, P., Monate, B., Pacalet, A., Prevosto, V., Regehr, J., Yakobowski, B. and Yang, X. 2012. Testing static analyzers with randomly generated programs. In *NASA Formal Methods*, A. E. Goodloe and S. Person, Eds. Berlin, Heidelberg, Berlin Heidelberg, Springer-Verlag, 120–125.

Daniel, B., DIG, D., Garcia, K. And Marinov, D. 2007. 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 2007, ESEC-FSE '07, New York, NY, USA, Association for Computing Machinery, 185–194.

Debray, S., Lopez-Garcia, P. and Hermenegildo, M. V. 1997. Non-failure analysis for logic programs. In 1997 International Conference on Logic Programming 1997, MIT Press, Cambridge, MA, Cambridge, MA, 48–62.

Dubois, C. 2000. Proving ML type soundness within coq. In *Theorem Proving in Higher Order Logics*, M. Aagaard and J. Harrison, Eds. Berlin, Heidelberg, Springer, Berlin Heidelberg, 126–144.

Ferreiro, D., Morales, J., Abreu, S. and Hermenegildo, M. 2023. Demonstrating (Hybrid) active logic documents and the Ciao Prolog playground, and an application to verification tutorials. In Technical Communications of the 39th International Conference on Logic Programming (ICLP 2023) 2023, volume 385 of Electronic Proceedings in Theoretical Computer Science (EPTCS), Open Publishing Association (OPA), 324–330, See also associated poster at https://cliplab.org/papers/hald-poster-iclp.pdf

Fortz, S., Mesnard, F., Payet, é., Perrouin, G., Vanhoof, W. and Vidal, G. 2020. An SMT-based concolic testing tool for logic programs. In Functional and Logic Programming - 15th International Symposium, FLOPS 2020, K. Nakano and K. Sagonas, Eds. volume 12073 of Lecture Notes in Computer Science, Akita, Japan, Springer, 215–219, September 14-16, 2020, Proceedings 2020

García de la Banda, M., Hermenegildo, M., Bruynooghe, M., Dumortier, V., Janssens, G. and Simoens, W. 1996. Global analysis of constraint logic programs. *ACM Transactions on Programming Languages and Systems* 18, 5, 564–615.

Garcia-Contreras, I., Morales, J. F. and Hermenegildo, M. V. 2016. Semantic code browsing. In Theory and practice of logic programming, 32nd Int'l. Conference on Logic Programming (ICLP'16), Cambridge University Press, Vol. 16, Special Issue, 5-6, 721–737.

Garcia-Contreras, I., Morales, J. F. and Hermenegildo, M. V. 2021. Incremental and modular context-sensitive analysis. *Theory and Practice of Logic Programming* 21, 2, 196–243.

Garcia-Contreras, I., Morales, J. and Hermenegildo, M. V. 2019. Multivariant assertion-based guidance in abstract interpretation. In Post-Proceedings of the 28th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'18) number 11408 in LNCS, Springer-Verlag, 184–201.

Garcia-Contreras, I., Morales, J. and Hermenegildo, M. V. 2020. 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) 202, volume 12042 of LNCS, Springer, 36–56,

Guarnieri, M., Köpf, B., Morales, J. F., Reineke, J. and Sánchez, A. 2020. Spectector: Principled detection of speculative information flows. In 2020 IEEE Symposium on Security and Privacy (SP) 2020, 1–19.

He, W., Di, P., Ming, M., Zhang, C., Su, T., Li, S. and Sui, Y. 2024. Finding and understanding defects in static analyzers by constructing automated oracles. In ACM International Conference on the Foundations of Software Engineering, 2024.

Hermenegildo, M. V. (2000) A system for automatically generating documentation for (C)LP programs. In *Special Issue on Parallelism and Implementation of (C)LP Systems 2000*, volume 30 of Electronic Notes in Theoretical Computer Science, Association for Computing Machinery.

Hermenegildo, M. V., Bueno, F., Carro, M., Lopez-Garcia, P., Mera, E., Morales, J. and Puebla, G. 2012. An overview of ciao and its design philosophy. *Theory and Practice of Logic Programming* 12, 1–2, 219–252.

Hermenegildo, M. V., Puebla, G. and Bueno, F. 1999. Using global analysis, partial specifications, and an extensible assertion language for program validation and debugging, *The Logic Programming Paradigm: A 25–Year Perspective 1999*, K. R. Apt, V. Marek, M. Truszczynski and D. S. Warren, Eds. Springer-Verlag, 161–192.

Hermenegildo, M. V., Puebla, G., Bueno, F. and Lopez-Garcia, P. 2003. Program development using abstract interpretation (and The Ciao System Preprocessor). In 10th International Static Analysis Symposium (SAS'03) 2003, number 2694 in LNCS, Springer-Verlag, 127–152.

Hermenegildo, M. V., Puebla, G., Bueno, F. and Lopez-Garcia, P. 2005. Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). *Science of Computer Programming* 58, 1–2, 115–140.

Hermenegildo, M. V., Puebla, G., Marriott, K. and Stuckey, P. 2000. Incremental analysis of constraint logic programs. *ACM Transactions on Programming Languages and Systems* 22, 2, 187–223.

Hermenegildo, M. V. and CLIP Group, T 1997. An Automatic Documentation Generator for (C)LP – Reference Manual, The Ciao System Documentation Series–TR CLIP5/97.3, Facultad de Informática, UPM. Online at https://ciao-lang.org

Jourdan, J.-H., Laporte, V., Blazy, S., Leroy, X. and Pichardie, D. 2015. A formally-verified C Static analyzer. In Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT

Symposium on Principles of Programming Languages, 2015, POPL '15, Association for Computing Machinery, New York, NY, USA, 247–259.

Jurjo, D., Morales, J. F., Lopez-Garcia, P. and Hermenegildo, M. V. 2024. Abstract environment trimming. Theory and Practice of Logic Programming,. Special Issue on ICLP'24

Kapus, T. and Cadar, C. 2017. Automatic testing of symbolic execution engines via program generation and differential testing. In IEEE/ACM International Conference on Automated Software Engineering (ASE 2017), IEEE Press. 2017, 590–600.

Klinger, C., Christakis, M. and Wüstholz, V. 2019. Differentially testing soundness and precision of program analyzers. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019, ISSTA 2019, Association for Computing Machinery, New York, NY, USA, 239–250.

Koukoutos, E. and Kuncak, V. 2014. Checking data structure properties orders of magnitude faster. In *Runtime Verification*, B. Bonakdarpour and S. A. Smolka, Eds. volume 8734 of Lecture Notes in Computer Science, Springer International Publishing, 263–268.

Le, V., Afshari, M. and Su, Z. 2014. Compiler validation via equivalence modulo inputs. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation 2014, PLDI '14, Association for Computing Machinery, New York, NY, USA, 216–226.

Le, V., Sun, C. and Su, Z. 2015. 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 2015, OOPSLA 2015, Association for Computing Machinery, New York, NY, USA, 386–399.

Lidbury, C., Lascu, A., Chong, N. and Donaldson, A. F. 2015. Many-core compiler fuzzing. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation 2015, PLDI '15,, Association for Computing Machinery, New York, NY, USA, 65–76.

Lopez-Garcia, P., Bueno, F. and Hermenegildo, M. V. 2005. 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) 2005, number 3573 in LNCS, Springer-Verlag, 19–35,

Lopez-Garcia, P., Bueno, F. and Hermenegildo, M. V. 2010. Automatic inference of Determinacy and mutual exclusion for logic programs using mode and type analyses. *New Generation Computing* 28, 2, 117–206.

Mera, E., Lopez-Garcia, P. and Hermenegildo, M. V. 2009. Integrating software testing and run-time checking in an assertion verification framework. In 25th Int'l. Conference on Logic Programming (ICLP'09) 2009, volume 5649 of LNCS, Springer-Verlag, 281–295,

Midtgaard, J. and MØller, A. 2017. QuickChecking static analysis properties. *Software Testing, Verification and Reliability* 27, 6, 6.

Muthukumar, K. and Hermenegildo, M. (1989). Determination of variable dependence information at compile-time through abstract interpretation. Technical Report ACA-ST-232-89.Austin, TX 78759, Microelectronics and Computer Technology Corporation (MCC),

Muthukumar, K. and Hermenegildo, M. (1990). Deriving a fixpoint computation algorithm for top-down abstract interpretation of logic programs. Technical Report ACT-DC-153-90.Austin, TX 78759, Microelectronics and Computer Technology Corporation (MCC),

Muthukumar, K. and Hermenegildo, M. 1991. Combined determination of sharing and freeness of program variables through abstract interpretation. In International Conference on Logic Programming (ICLP 1991) 1991, MIT Press, 49–63.

Muthukumar, K. and Hermenegildo, M. 1992. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming* 13, 2-3, 315–347.

Navas, J., Bueno, F. and Hermenegildo, M. V. 2006. Efficient top-down set-sharing analysis using cliques. In 8th International Symposium on Practical Aspects of Declarative Languages (PADL'06) 2006, number 2819 in LNCS, Springer-Verlag, 183–198,

Paulson, L. (1990) Isabelle: The next 700 theorem provers. In P. Odifreddi, Ed. Logic and Computer Science, Academic Press, 361–386.

Puebla, G., Bueno, F. and Hermenegildo, M. V. 2000. An assertion language for constraint logic programs. In *Analysis and Visualization Tools for Constraint Programming 2000*, P. Deransart, M. V. Hermenegildo and J. Maluszynski, Eds. Springer-Verlag, 23–61.

Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C. and Yang, X. 2012. Test-case reduction for C compiler bugs. In Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation 2012, PLDI '12, Association for Computing Machinery, New York, NY, USA, 335–346.

Sato, T. and Tamaki, H. 1984. Enumeration of success patterns in logic programs. *Theoretical Computer Science* 34, 1-2, 227–240.

Shao, Z., Saha, B., Trifonov, V. and Papaspyrou, N. 2002. A type system for certified binaries. *ACM SIGPLAN Notices* 37, 1, 217–232.

Søndergaard, H. 1986. An application of abstract interpretation of logic programs: occur check reduction. In European Symposium on Programming, LNCS 123 1986, Springer, 327–338.

Stade, Y., Tilscher, S. and Seidl, H. 2024. Partial correctness of the top-down solver. Archive of Formal Proofs, Formal proof development, https://isa-afp.org/entries/Top_Down_Solver.html

Stulova, N., Morales, J. F. and Hermenegildo, M. V. 2015. Practical run-time checking via unobtrusive property caching. In Theory and Practice of Logic Programming, 31st Int'l. Conference on Logic Programming (ICLP'15), Vol. 15, 726–741, Special Issue 04-05, https://arxiv.org/abs/1507.05986

Stulova, N., Morales, J. F. and Hermenegildo, M. V. 2016. 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) 2016, ACM Press, 90–103.

Sun, C., Le, V. and Su, Z. 2016. Finding compiler bugs via live code mutation. In Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications 2016, OOPSLA 2016, Association for Computing Machinery, New York, NY, USA, 849–863.

Vaucheret, C. and Bueno, F. 2002. More precise yet efficient type inference for logic programs. In 9th International Static Analysis Symposium (SAS'02) 2002 volume 2477 of Lecture Notes in Computer Science, Springer-Verlag, 102–116,

Warren, D. and Pereira, F. C. N. 1982. An efficient, easily adaptable system for interpreting natural language queries. *American Journal of Computational Linguistics* 8, 3-4, 110–122.

Wu, J., Hu, G., Tang, Y. and Yang, J. 2013. Effective dynamic detection of alias analysis errors. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering 2013, ESEC/FSE 2013, Association for Computing Machinery, New York, NY, USA, 279–289.

Yang, X., Chen, Y., Eide, E. and Regehr, J. 2011. Finding and understanding bugs in C compilers. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation 2011, PLDI '11, Association for Computing Machinery, New York, NY, USA, 283–294.

Zhang, C., Su, T., Yan, Y., Zhang, F., Pu, G. and Su, Z. 2019. 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 2019, Association for Computing Machinery, 763–773.