Search-Guided Generation of Properties for Program Analyzers*

Daniela Ferreiro[1,2]

[1]Universidad Politécnica de Madrid (UPM) Madrid, Spain
[2]IMDEA Software Institute, Madrid, Spain

Abstract

Static analysis is a fundamental component of modern software tools, yet the increasing complexity of static analyzers makes them difficult to validate. Even software analyzers grounded in formal techniques, such as abstract interpretation, are prone to subtle implementation bugs. In this work, we present *checkification*, a simple, automatic technique for testing static analyzers. The core idea is to check, over a suite of test cases, that the properties inferred statically are satisfied dynamically. Our approach is framed within the `Ciao` assertion-based framework, leveraging its support for static and dynamic checking, test generation, and unit testing. We have implemented this technique in the `CiaoPP` static analyzer, leading to the discovery of several non-trivial bugs with reasonable overhead. In addition, we address a major challenge in test generation: producing diverse, property-guided test cases with strong invariants. To this end, we introduce flexible search strategies for (C)LP programs, enabling expressive and configurable test case generation.

Keywords

Static Analysis, Testing, Run-time Checks, Assertions, Abstract Interpretation, Logic Programming, Constraint Logic Programming

1. Introduction and Problem Description

Since the early years of logic programming, static program analysis has been a fundamental technique used to achieve several objectives ranging from sequential program optimization [1, 2, 3, 4] and parallelization [5, 6] to program verification [7, 8, 9, 10, 11]. However, static analyzers and verifiers are typically large and complex, which can make them prone to bugs. This can limit their applicability in real-life production compilers and development environments, since the tasks they are used in are typically critical and need reassurance about the soundness of the analysis results. At the same time, the validation of static analyzers is a challenging problem, with limited literature and few dedicated tools addressing this issue [12, 13, 14, 15, 16, 17, 18]. 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. This is a clear instance of the classic problem of who checks the checker. As a result, in current practice, extensive testing is the most extended and realistic validation technique, but this poses some significant challenges too. Testing separate components of the analyzer misses integration testing, and designing proper oracles for testing the complete tool is difficult.

A further complication lies in test data generation. Useful test data is often unavailable or too specialized, lacking the configurability needed to explore a wide range of scenarios. Even when data exists, it may fail to trigger corner case behaviors. In other words, achieving good coverage, especially for specific semantic properties, requires not only proper benchmarks but also diverse, high-quality inputs. Additionally, the generation of suitable properties and oracles to assess expected behavior is itself an open problem in many cases.

*Partially funded by MICIU projects CEX2024-001471-M *María de Maeztu* and TED2021-132464B-I00, by EU Grant 101154447 NEAT, and by the Tezos foundation.
ICLP DC 2025: 21st Doctoral Consortium on Logic Programming, September 2025, Rende, Italy.
daniela.ferreiro@imdea.org (D. Ferreiro)
0009-0002-1072-8989 (D. Ferreiro)

2. Research Objectives

In view of the above, our research objectives for the PhD are as follows:

- A first objective is the *development, improvement, and extensive benchmarking* of a practical technique for testing static analyzers. Our approach here is based on exploiting the particular characteristics of the program analyzers, verifiers, and assertion languages developed for (C)LP.
- A second objective involves the development of novel methods to generate relevant test cases, that effectively exercise a wide range of analysis scenarios. Our approach here is also based on exploiting the specific characteristics of (C)LP, and in particular using properties in assertions written as (C)LP programs as generators and flexible search rules to perform constraint-based automatic generation of test cases. This will involve *pushing the development of techniques for supporting flexible search rules in general in (C)LP programs*, which should also be useful beyond test case generation.
- A final objective is to prove practically all the proposed techniques in a realistic analyzer and compiler. To this end, we develop our work in the context of Ciao Prolog [19]. The Ciao programming 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 [20] 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 [21], 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. We thus argue that the Ciao/CiaoPP system is thus a good subject for testing our techniques.

In the following sections, we explain further our progress so far in the thesis objectives and the remaining challenges towards these objectives.

3. Current Progress and Results

3.1. The Checkification Algorithm

The first component of our work is the *checkification* algorithm and system [26]: a simple, automatic, technique for testing static analyzers. This work is based on 2preliminary work by I. Casso and other members of the group on the topic [27], but we have extended it with new ideas and techniques, as well as implemented it and fully benchmarked it. *Checkification* combines four basic components (all present in the Ciao system):

- A static analyzer (the PLAI *static analyzer* [28, 29, 30]),
- An assertion *run-time checking framework* [31, 32],
- A *(random) test case* generation framework [33],
- A *unit-testing framework* [34]

These elements are combined in a novel way that allows testing the static analyzer almost for free. Intuitively, the idea consists in checking, over a suite of benchmarks (or generated benchmarks), that the properties inferred statically are satisfied dynamically. The overall testing process (see Figure 1), 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

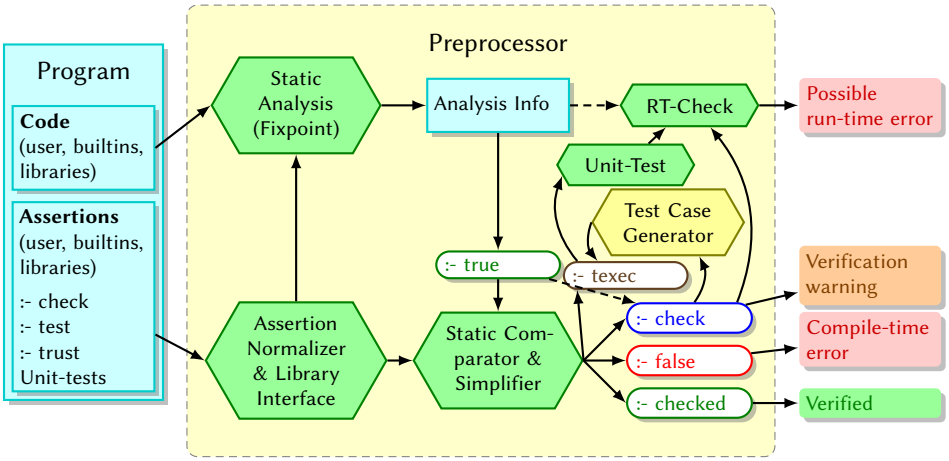

Figure 1: The Ciao assertion framework (CiaoPP's verification/testing architecture).

execution time. Finally, 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 the 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, in [14], this technique was applied 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.

Illustrative example. Let us illustrate the main idea of the approach with an example. Assume we have the following simple Prolog program, where we use an 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.

```
1  :- entry prepend(_,+list,-).
2
3  prepend(X,Xs,Ys) :-
4      Ys=[X|Rest],
5      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` literal, the abstract value for `Ys` is not modified at all. The result of the analysis will be represented as a new source file with interspersed assertions, as shown in Fig. 2 (lines 3-5, 8, 10, and 12). Note that the correct result, if the analysis considered aliasing, would be that there is no groundness information for `Ys` at the end of the clause (line 12), since there is none for `X` or `Xs` at the beginning either. `Ys` could only be inferred to be `nonvar`, but instead is incorrectly inferred to be `nonground` too (line 10). Normally `any/1` properties (i.e., top, or unknown) would not actually be included in the analysis output for conciseness, but are included in Fig. 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 contains an error. The insight is that, thanks to the different capabilities of the Ciao model this can be achieved by (**1**) *turning the status of the* `true` *assertions produced by the analyzer into* `check`, as shown in Fig. 3. This would normally not make any sense since these `true` assertions have been proved by the analyzer. But that is exactly what we want to check, i.e., whether the information inferred is incorrect. To do this, (**2**) we run the transformed program (Fig. 3) again through CiaoPP (Fig. 1) but *without performing any analysis*.

In that case, the `check` literals (stemming from the `true` literals of the previous run) will not be simplified in the comparator (since there is no abstract information to compare against) and instead will be converted directly to run-time tests. 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, already be available as test assertions in the program or, alternatively, the random test case generator can be used to generate them. E.g., 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 a run-time error in the `check` in line 12, letting us know that the third program point, and thus the analysis, is incorrect.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

```
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)).
```

Figure 2: An incorrect simple mode analysis.

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

:- check pred prepend(X,Xs,Ys)
    :  (any(X), nonvar(Xs), var(Ys))
    => (any(X), nonvar(Xs), nonground(Ys), nonvar(Ys)).

prepend(X,Xs,Ys) :-
    check(any(X), nonvar(Xs), var(Ys), var(Rest)),
    Ys=[X|Rest],
    check(any(X), nonvar(Xs), nonground(Ys), nonvar(Ys), var(Rest)),
    Rest=Xs,
    check(any(X), nonvar(Xs), nonground(Ys), nonvar(Ys), nonvar(Rest)).
```

Figure 3: The instrumented program.

imprecise (although of course, we cannot have actual correctness in general by testing).

We have carried out a full implementation of the approach and completed a substantial experimental evaluation [26]. The results show that *checkification* 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. The results and more details about the algorithm are provided in [26].

3.2. Search Rules and Generation of Test Cases from Properties

As mentioned before, a second challenge that we are addressing is that the test case generation process that is necessary during the validation of static analyzers using *checkification*, and, in fact, while testing logic programs in general, becomes progressively difficult when dealing with complex data structures and programs whose correctness depends on complex, hard-to-test conditions. Prolog provides a unique advantage in this context, as its declarative nature allows the properties to be proved to be expressed as predicates, which can then be leveraged to generate test cases [35, 36, 26, 37, 38].

Prolog systems traditionally employ depth-first search as their execution strategy. While this choice is well-justified for efficiency reasons, and generally accepted, it is well known that this can lead to incompleteness when evaluating programs over infinite search spaces. This second thesis objective motivates us to revisit the role of the search rule in Prolog and explore the challenges involved in running Prolog predicates with alternative search strategies in the real world. The hypothesis is that flexible and customizable control mechanisms can be very useful in uncovering edge cases in testing, particularly when generating data structures with strong invariants. At the same time, our objective is to *push the development of techniques for supporting flexible search rules in general in (C)LP programs*, to be useful beyond test case generation.

Writing Prolog code that implements a search with a particular strategy from scratch is not particularly difficult and it is also not difficult to run Prolog predicates using other search rules using variations of the standard meta-interpreter. However, our challenge is to be able to run standard Prolog predicates with different search strategies while maintaining compatibility with modules, built-ins, and other libraries and features. Straightforward implementation approaches typically limit the use of alternative search rules to a subset of Prolog and thus do not support the full expressiveness of the language.

In the context of test case generation, our work leverages assertion preconditions as generators: since these preconditions are conjunctions of literals, the corresponding predicates can be used to systematically produce valid inputs. The key innovation lies in executing standard predicates under non-standard search rules, enabling either fully automatic or user-guided generation. Figure 3.2 demonstrates this approach with a binary tree library.

The `tree/1` property describes the shape of trees, allowing either an empty node or a compound term

```
:- search_rule(tree/1,bf).

:- prop tree/1.
tree(empty).
tree(t(L,N,R)) :-
    tree(L),
    gen([sr(rnd)], (nnegint(X), X =< 15)),
    tree(R).

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

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

:- pred insert(X,T0,T1) : tree(T0) => tree(T1) + gen([sorted_tree(T0), tsum(T0,10)]).
insert(X,T0,T1) :- ...

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

with left and right subtrees. The use of `gen([sr(rnd)], (nnegint(X), X =< 15))` in the recursive clause shows how value generation can be customized: in this case, generating non-negative integers bounded by 15. Moreover, the declaration `:- search_rule(tree/1, bf)` specifies that breadth-first search should be used to generate trees, which helps avoid unbalanced growth and improves coverage by generating smaller and more diverse trees earlier. In the `insert/3` specification, the generator is further guided by auxiliary properties such as `sorted_tree/1` and `tsum/2`, which restrict the structure and content of the input trees.

Our thesis is that flexible search rules can significantly enhance test case generation, enabling users to customize and implement their own generators, including those producing data structures with complex invariants.

4. Conclusions and Future Work

Our starting point has been *checkification*, an automatic method for testing static analysis tools by checking that the properties inferred statically are satisfied dynamically. We have shown how *checkification* can be implemented effectively in practice, improving several aspects of the approach, and performed a full benchmarking of the technique showing its usefulness in practice for discovering and locating interesting, non-trivial and previously undetected bugs, with reasonable overhead.

During this first phase of the work, we have identified test case generation as an important remaining challenge, and as a result, as our second thesis objective, we are developing methods for specifying search procedures in a concise and elegant way for realistic Prolog programs. Our work here allows users to switch search rules at any point in the program, and to combine simple primitives to build useful search heuristics for generating complex data structures for test cases. Future work in the thesis involves improving the generation process by supporting more configurable search parameters, including termination conditions, solution limits, and other customizable aspects. We will apply these techniques to the generation of programs from grammars for creating benchmarks which can serve as sophisticated automatic inputs to further automate the *checkification* algorithm.

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 the approaches addressed in the thesis will together offer a practical and effective technique for detecting errors in large and complex analysis tools, as well as in logic programs in general. In addition, we believe that the availability of sophisticated and easy to use search rules in Prolog can also be useful in both teaching the language [39] and in practical applications.

Declaration on Generative AI

The author has not employed any Generative AI tools.

References

[1] P. Van Roy, 1983-1993: The Wonder Years of Sequential Prolog Implementation, Journal of Logic Programming 19/20 (1994) 385–441.

[2] P. Van Roy, A. Despain, High-Performance Logic Programming with the Aquarius Prolog Compiler, IEEE Computer Magazine (1992) 54–68.

[3] J. Morales, M. Carro, M. Hermenegildo, Improving the Compilation of Prolog to C Using Moded Types and Determinism Information, in: Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages, volume 3057 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, 2004, pp. 86–103.

[4] M. Carro, J. Morales, H. Muller, G. Puebla, M. Hermenegildo, High-Level Languages for Small Devices: A Case Study, in: K. Flautner, T. Kim (Eds.), Compilers, Architecture, and Synthesis for Embedded Systems, ACM Press / Sheridan, 2006, pp. 271–281.

[5] K. Muthukumar, M. Hermenegildo, Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation, in: 1989 North American Conference on Logic Programming, MIT Press, 1989, pp. 166–189.

[6] F. Bueno, M. García de la Banda, M. Hermenegildo, Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming, ACM TOPLAS 21 (1999) 189–238.

[7] M. Hermenegildo, G. Puebla, F. Bueno, Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging, in: The Logic Programming Paradigm: a 25–Year Perspective, Springer-Verlag, 1999, pp. 161–192.

[8] G. Puebla, F. Bueno, M. Hermenegildo, A Generic Preprocessor for Program Validation and Debugging, in: Analysis and Visualization Tools for Constraint Programming, number 1870 in LNCS, Springer-Verlag, 2000, pp. 63–107.

[9] M. Hermenegildo, G. Puebla, F. Bueno, P. L. Garcia, Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor), Science of Computer Programming 58 (2005) 115–140.

[10] M. Sanchez-Ordaz, I. Garcia-Contreras, V. Perez-Carrasco, J. F. Morales, P. Lopez-Garcia, M. Hermenegildo, VeriFly: On-the-fly Assertion Checking via Incrementality, Theory and Practice of Logic Programming 21 (2021) 768–784.

[11] M. Hermenegildo, J. Morales, P. Lopez-Garcia, M. Carro, Types, modes and so much more – the Prolog way, in: D. S. Warren, V. Dahl, T. Eiter, M. V. Hermenegildo, R. Kowalski, F. Rossi (Eds.), Prolog - The Next 50 Years, number 13900 in LNCS, Springer, 2023, pp. 23–37. URL: https://cliplab.org/papers/AssertionsAndOther-PrologBook.pdf.

[12] A. Bugariu, V. Wüstholz, M. Christakis, P. Müller, Automatically testing implementations of numerical abstract domains, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 768–778. URL: https://doi.org/10.1145/3238147.3240464. doi:10.1145/3238147.3240464.

[13] J. Midtgaard, A. Møller, QuickChecking Static Analysis Properties, Softw. Test., Verif. Reliab. 27 (2017). URL: https://doi.org/10.1002/stvr.1640. doi:10.1002/stvr.1640.

[14] J. Wu, G. Hu, Y. Tang, J. Yang, Effective dynamic detection of alias analysis errors, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, Association for Computing Machinery, New York, NY, USA, 2013, p. 279–289. URL: https://doi.org/10.1145/2491411.2491439. doi:10.1145/2491411.2491439.

[15] C. Zhang, T. Su, Y. Yan, F. Zhang, G. Pu, Z. Su, Finding and understanding bugs in software model checkers, in: Proceedings of the 13th Joint Meeting of the 18th European Software Engineering

Conference and the 27th Symposium on the Foundations of Software Engineering, 2019, pp. 763–773. doi:10.1145/3338906.3338932.

[16] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, X. Yang, Testing static analyzers with randomly generated programs, in: A. E. Goodloe, S. Person (Eds.), NASA Formal Methods, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 120–125.

[17] C. Klinger, M. Christakis, V. Wüstholz, Differentially testing soundness and precision of program analyzers, in: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Association for Computing Machinery, New York, NY, USA, 2019, p. 239–250. URL: https://doi.org/10.1145/3293882.3330553. doi:10.1145/3293882.3330553.

[18] W. He, P. Di, M. Ming, C. Zhang, T. Su, S. Li, Y. Sui, Finding and understanding defects in static analyzers by constructing automated oracles, in: ACM International Conference on the Foundations of Software Engineering, 2024. doi:10.1145/3660781.

[19] M. V. Hermenegildo, F. Bueno, M. Carro, P. Lopez-Garcia, E. Mera, J. Morales, G. Puebla, An Overview of Ciao and its Design Philosophy, Theory and Practice of Logic Programming 12 (2012) 219–252. URL: https://arxiv.org/abs/1102.5497. doi:10.1017/S1471068411000457.

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

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

[22] D. Ferreiro de Aguiar, Automatic Analysis of Code Examples, Master's thesis, Universidad Politécnica de Madrid, ETSIInf, E-28660, Boadilla del Monte, Madrid, Spain, 2021. BSc Thesis.

[23] D. Ferreiro de Aguiar, A System for generating interactive tutorials for CiaoPP, Master's thesis, Universidad Politécnica de Madrid, ETSIInf, E-28660, Boadilla del Monte, Madrid, Spain, 2023. MSc Thesis.

[24] J. Morales, S. Abreu, D. Ferreiro, M. Hermenegildo, Teaching Prolog with Active Logic Documents, Technical Report CLIP-1/2022.0, Technical University of Madrid (UPM) and IMDEA Software Institute, 2022.

[25] D. Ferreiro, J. Morales, S. Abreu, M. Hermenegildo, 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), volume 385 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, Open Publishing Association (OPA), 2023, pp. 324–330. URL: https://cliplab.org/papers/hald-demo-iclp-tc.pdf, see also associated poster at https://cliplab.org/papers/hald-poster-iclp.pdf.

[26] D. Ferreiro, I. Casso, P. Lopez-Garcia, J. F. Morales, M. V. Hermenegildo, Checkification: A Practical Approach for Testing Static Analysis Truths, Theory and Practice of Logic Programming (2025). URL: https://arxiv.org/abs/2501.12093.

[27] I. Casso, J. F. Morales, P. López-García, M. V. Hermenegildo, Testing Your (Static Analysis) Truths, in: M. Fernández (Ed.), Logic-Based Program Synthesis and Transformation - 30th International Symposium, Post-Proceedings, volume 12561 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 271–292. doi:10.1007/978-3-030-68446-4_14.

[28] K. Muthukumar, M. Hermenegildo, Compile-time Derivation of Variable Dependency Using Abstract Interpretation, Journal of Logic Programming 13 (1992) 315–347.

[29] M. Hermenegildo, G. Puebla, K. Marriott, P. Stuckey, Incremental Analysis of Constraint Logic Programs, ACM Transactions on Programming Languages and Systems 22 (2000) 187–223.

[30] I. Garcia-Contreras, J. Morales, M. Hermenegildo, 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), volume 12042 of *LNCS*, Springer, 2020, pp. 36–56. doi:10.1007/978-3-030-45260-5_3.

[31] N. Stulova, J. F. Morales, M. Hermenegildo, Practical Run-time Checking via Unobtrusive Property Caching, Theory and Practice of Logic Programming, 31st Int'l. Conference on Logic Programming (ICLP'15) Special Issue 15 (2015) 726–741. doi:10.1017/S1471068415000344, https://arxiv.org/abs/1507.05986.

[32] N. Stulova, J. F. Morales, M. Hermenegildo, 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), ACM Press, 2016, pp. 90–103.

[33] I. Casso, J. F. Morales, P. Lopez-Garcia, M. Hermenegildo, An Integrated Approach to Assertion-Based Random Testing in Prolog, in: M. Gabbrielli (Ed.), Post-Proceedings of the 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'19), volume 12042 of *LNCS*, Springer-Verlag, 2020, pp. 159–176. doi:10.1007/978-3-030-45260-5_10.

[34] E. Mera, P. Lopez-Garcia, M. Hermenegildo, Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework, in: 25th Int'l. Conference on Logic Programming (ICLP'09), volume 5649 of *LNCS*, Springer-Verlag, 2009, pp. 281–295.

[35] I. Casso, J. F. Morales, P. Lopez-Garcia, M. Hermenegildo, An Integrated Approach to Assertion-Based Random Testing in Prolog, in: M. Gabbrielli (Ed.), 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'19), volume 12042 of *LNCS*, Springer-Verlag, 2020, pp. 159–176. doi:10.1007/978-3-030-45260-5_10.

[36] S. Fortz, F. Mesnard, É. Payet, G. Perrouin, W. Vanhoof, G. Vidal, An SMT-Based Concolic Testing Tool for Logic Programs, in: K. Nakano, K. Sagonas (Eds.), Functional and Logic Programming - 15th International Symposium, FLOPS 2020, Akita, Japan, September 14-16, 2020, Proceedings, volume 12073 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 215–219. doi:10.1007/978-3-030-59025-3_13.

[37] R. Denney, Test-case generation from Prolog-based specifications, IEEE Software 8 (1991) 49–57.

[38] N. Mweze, W. Vanhoof, Automatic generation of test inputs for mercury programs, in: Pre-proceedings of LOPSTR 2006, July 2006. Extended abstract.

[39] M. V. Hermenegildo, J. F. Morales, P. Lopez-Garcia, Teaching Pure LP with Prolog and a Fair Search Rule, in: Proceedings of the 40th ICLP Workshops, volume 3799, CEUR-WS.org, 2024. URL: https://ceur-ws.org/Vol-3799/paper2PEG2.0.pdf.