VeriFly: On-the-fly Assertion Checking with CiaoPP*
(tool presentation)

Miguel A. Sanchez-Ordaz[1,2] Isabel Garcia-Contreras[1,2] Víctor Pérez[1,2]
Jose F. Morales[1,2] Pedro Lopez-Garcia[1,3] Manuel V. Hermenegildo[1,2]

[1]IMDEA Software Institute [2]Universidad Politécnica de Madrid (UPM)
[3]Spanish Council for Scientific Research (CSIC), Madrid, Spain

{ma.sanchez.ordaz,isabel.garcia,victor.perez,josef.morales,pedro.lopez,manuel.hermenegildo}@imdea.org

Fast response times are essential for the integration of global static analysis tools at early stages of the software development cycle. Triggering a full and precise semantic analysis of a software project every time a change is made can be prohibitively expensive. This is specially the case when complex properties need to be inferred for large, realistic code bases. In the CiaoPP static analysis and verification framework this challenge is addressed through incremental (context- and path-sensitive) analysis that is responsive to program edits, at different levels of granularity. In this tool paper we present how the integration of this framework within an integrated development environment (IDE) takes advantage of such incrementality to achieve a high level of reactivity when reflecting analysis and verification results back as colorings and tooltips directly on the program text –the tool's VeriFly mode. The concrete implementation that we describe is Emacs-based and reuses in part off-the-shelf "on-the-fly" *syntax* checking facilities (`flycheck`). Our initial experience with the tool shows quite promising results, with low latency times that provide early, continuous, and precise assertion checking and other semantic feedback to programmers during the development process.

1 Introduction

Global static analysis and verification tools can greatly help developers detect high-level, semantic errors in programs or certify their absence. Arguably, such tools are more effective the earlier the stage in which they are applied within the software development process. Particularly useful is the application of such tools during code development, simultaneously with the code writing process, alongside the compiler, debugger, etc.

The tight integration of global analysis and verification at such early stages of the software development cycle requires fast response times and source-level presentation of the results within the code development environment, in order to provide timely and useful feedback to the programmer. However, triggering a full and precise semantic analysis of a software project every time a change is made can be expensive and may not be able to meet the requirements of the scenario described. This is specially the case when complex properties need to be inferred for large, realistic code bases.

CiaoPP [14, 6] is a program development framework which performs combined static and dynamic program analysis, assertion checking, and program transformations, based on computing provably safe approximations of properties, generally using the technique of abstract interpretation [3]. CiaoPP can be applied to programs expressed as (constrained) Horn clauses (and in particular, in the Ciao programming language[1]), as well as in other high- and low-level languages, using the well-understood technique of se-

*Partially funded by MICINN PID2019-108528RB-C21 *ProCode* and Madrid P2018/TCS-4339 *BLOQUES-CM*. We would like to thank the anonymous reviewers for their very useful comments.

[1]`https://github.com/ciao-lang/devenv`

Submitted to:
FIDE 2021

mantic translation into intermediate Horn clause-based representation [26, 11, 22, 8, 9, 10, 4, 16, 5].[2] The framework has many uses, but one of the main ones is precisely as an aid for the programmer during program development, since it can capture semantic errors that are significantly higher-level than those detected by classical compilers, as well as produce certificates that programs do not violate their assertions, eliminate run-time assertion tests, etc.

In CiaoPP, the requirements for fast response time and precision stemming from interactive use pointed out above are addressed through a number of techniques, and in particular by an efficient fixpoint engine, which performs context- and path-sensitive inter-procedural analysis *incrementally*, i.e., reactively to fine-grain editions, avoiding reanalyses where possible, both within modules and across the modular organization of the code into separate compilation units. In this tool paper we illustrate how the integration of CiaoPP's static analysis and verification within an integrated development environment (IDE) takes advantage of the incremental and modular features of the framework to achieve a high level of reactivity when reflecting analysis and verification results back as colorings and tooltips directly on the program text –the tool's VeriFly mode. The concrete integration described builds on an existing Emacs-based development environment for the Ciao language. Emacs was chosen because it is a solid platform and preferred by many experienced users. The integration reuses off-the-shelf "on-the-fly" *syntax* checking capabilities offered by the Emacs `flycheck` package.[3] This low-maintanance approach should be easily reproducible in other modern extensible editors.

2 The CiaoPP Framework

We start by providing an informal overview of the components and operation of the Ciao/CiaoPP framework (represented by the yellow shaded part of Fig. 1).

Front end and CHC IR Before getting into the analysis and verification phases, a basic technique used by the framework, in order to support different input languages, is to translate input programs to a language-independent intermediate representation, which in this case is (constrained) Horn clause-based [22] –an approach used nowadays in many analysis and verification tools. We refer to this intermediate representation as the "CHC IR." This CHC IR is handled uniformly by the analyzers, independently of the input language. The translation is performed by the "Front-end" (Fig. 1). Techniques such as partial evaluation and program specialization offer powerful methods to obtain such translations with provable correctness. Using this approach, CiaoPP has been applied to the analysis, verification, and optimization of a number of languages (besides Ciao) ranging from very high-level ones to bytecode and machine code, such as Java, XC (C like) [20], Java bytecode [25, 24], ISA [19], LLVM IR [18], Michelson [27], ...), and properties ranging from pointer aliasing and heap data structure shapes to execution time, energy, or smart contract "gas" consumption [23, 17]. However, for simplicity and concreteness, in our examples herein we will use logic programs, because of their proximity to the CHC IR. In particular, the examples will be written in Ciao's [12] logic programming subsetm i.e., Prolog-style syntax and operational semantics.[4] The framework itself is also written in Ciao.

The assertion language An important element of the framework is its *assertion language* [1, 13, 28]. Such assertions can express a wide range of properties, including functional (state) properties (e.g., shapes, modes, sharing, aliasing, ...) as well as non-functional (i.e., global, computational) properties such as resource usage (energy, time, memory, ...), determinacy, non-failure, or cardinality. The set of properties

[2] Space in a short tool paper does not permit covering other related work properly (such as, e.g., [31, 32]), but additional references can be found in the CiaoPP overview papers and the other citations provided.

[3] `https://github.com/flycheck/flycheck`

[4] Ciao is a general-purpose programming language that integrates a number of programming paradigms including functional, logic, and constraint programming.

is extensible and new abstract domains (see the later discussion of the analysis) can be defined as "plug-ins" to support them. Assertions associate these properties to different program points, and are used for multiple purposes, including writing specifications, reporting static analysis and verification results to the programmer, providing assumptions, describing unknown code, or generating test cases automatically.

We will use for simplicity (a subset of) the "`pred`"-type assertions, which allow describing sets of *pre-conditions* and *conditional postconditions* on the state for a given procedure (predicate). The syntax that we present below is that of the CHC IR (and the Ciao language); assertions in the different input languages are translated by the front end to and from this format (left of Fig. 1). A `pred` assertion is of the form:

`:- [` *Status* `] pred` *Head* `[:` *Pre* `] [=>` *Post* `].`

where *Head* is a predicate descriptor that denotes the predicate that the assertion applies to, and *Pre* and *Post* are conjunctions of *property literals*, i.e., literals corresponding to predicates meeting certain conditions [28]. There can be multiple `pred` assertions for a predicate. *Pre* expresses properties that hold when *Head* is called, namely, at least one *Pre* must hold for each call to *Head*. *Post* states properties that hold if *Head* is called in a state compatible with *Pre* and the call succeeds. Both *Pre* and *Post* can be empty conjunctions (meaning true), and in that case they can be omitted. *Status* is a qualifier of the meaning of the assertion. Here we consider (in the context of static assertion checking [29]) the following *Status*es:

- `check`: the assertion expresses properties that must hold at run-time, i.e., that the analyzer should prove (or else generate run-time checks for). `check` is the *default* status, and can be omitted.
- `checked`: the analyzer proved that the property holds in all executions.
- `false`: the analyzer proved that the property does not hold in some execution.

For example, the following assertions describe different behaviors of the `pow(X,N,P)` predicate, that computes $P = X^N$: `(1)` is stating that if the exponent of a power is an even number, the result (`P`) is non-negative, `(2)` states that if the base is a non-negative number and the exponent is a natural number the result `P` also is non-negative:

```
1 :- pred pow(X,N,P) : (int(X), even(N)) => P ≥ 0.  % (1)
2 :- pred pow(X,N,P) : ( X ≥ 0,  nat(N)) => P ≥ 0.  % (2)
3 pow(_, 0, 1).
4 pow(X, N, P) :-
5     N > 0,
6     N1 is N - 1,
7     pow(X, N1, P0),
8     P is X * P0.
```

Assertion verification The CiaoPP verification framework uses analyses based on the abstract interpretation technique (the "Static Analyzer" in Fig. 1), in order to statically compute safe approximations of the program semantics at different relevant program points. Given a program, a number of abstract domains are automatically selected, as determined by their relevance to the properties that appear in the assertions written by the programmer or present in libraries and a combined analysis is performed using this set of abstract domains. The resulting safe approximations are compared directly with these assertions (the "Static Checker" in Fig. 1) in order to prove the program correct or incorrect with respect to them. For each assertion originally with status `check`, the result of this process (boxes on the right of Fig. 1) can be: that it is verified (the new status is `checked`), that a violation is detected (the new status is `false`), or that it is not possible to decide either way, in which case the assertion status remains as `check`. In such cases, optionally, a warning may be displayed and/or a run-time test generated for (the part of) the assertion that could not be discharged at compile-time, test cases generated, etc.

Regarding the abstract comparisons, consider a program p, and its *concrete semantics* $[\![p]\!]$. The *abstract interpretation* of the program performed by the static analyzer computes an abstract meaning of the program, $[\![p]\!]_\alpha$ (also represented with assertions), that is guaranteed to be a safe *over*-approximation of the abstraction of the concrete semantics, which we denote as $[\![p]\!]_{\alpha^+}$. Alternatively, the analysis can be designed

Property	**Target**	**Sufficient condition**
p is partially correct w.r.t. I_α	$\alpha([\![p]\!]) \subseteq I_\alpha$	$[\![p]\!]_{\alpha+} \subseteq I_\alpha$
p is complete w.r.t. I_α	$I_\alpha \subseteq \alpha([\![p]\!])$	$I_\alpha \subseteq [\![p]\!]_{\alpha-}$
p is not partially correct w.r.t. I_α	$\alpha([\![p]\!]) \not\subseteq I_\alpha$	$[\![p]\!]_{\alpha-} \not\subseteq I_\alpha$, or $[\![p]\!]_{\alpha+} \cap I_\alpha = \emptyset \wedge [\![p]\!]_{\alpha+} \neq \emptyset \wedge [\![p]\!]_{\alpha-} \neq \emptyset$
p is incomplete w.r.t. I_α	$I_\alpha \not\subseteq \alpha([\![p]\!])$	$I_\alpha \not\subseteq [\![p]\!]_{\alpha+}$

Table 1: Some uses of safe approximations of program semantics for verifying assertions (I_α).

to safely *under*-approximate the abstraction of the concrete semantics of p. In this case, we use $[\![p]\!]_{\alpha^-}$ to represent the result of such an analysis. Table 1 shows (simplified) sufficient conditions for correctness and completeness w.r.t. I_α (the abstract intended semantics), which can be used when $[\![p]\!]$ is safely over- or under-approximated (see [2, 29] for a more detailed discussion).

As to performing the static analysis, CiaoPP builds an analysis result in the shape of an *abstract graph*, representing how the clauses are executed. Nodes in this graph abstract how predicates are called (i.e., how they are used in the program). A predicate may have several nodes if there are different calling situations (also known as context-sensitivity). For each calling situation, properties that hold if the predicate succeeds are also inferred; these are similar to *contracts*, and can be represented by (true) assertions. The values for the call and success properties are abstractions of the state in the concrete execution of the program and are defined in terms of the abstract domain(s) selected.[5] The edges in the graph capture how predicates call each other. Hence this analysis graph also provides an abstraction of the paths explored by the concrete executions through the program (also known as path-sensitivity). The analysis graph thus embodies two different abstractions (two different abstract domains): the graph itself is a *regular approximation* of the paths through the program, using a domain of regular structures. Separately, the abstract values (call and success patterns) contained in the graph nodes are finite representations of the states occurring at each point in these program paths, by means of one or more data-related abstract domains.

Supporting incrementality In order to support incrementality, the analysis graph produced by the static analyzer is made persistent ("Analysis DB" in Fig. 1), storing an abstraction of the behavior of each predicate and predicate (abstract) call dependencies. In turn, the "Front-end" (Fig. 1) keeps track of and translates source code changes into, e.g., clause and assertion additions, deletions, and changes in the intermediate representation (Δ CHC in the figure). These changes are transmitted to the static analyzer, which performs incremental fixpoint computation. This process consists in finding the parts of the graph that need to be deleted or recomputed, following their dependencies, and updating the fixpoint [6, 7, 30, 15]. The key point here is that a tight relation between the analysis results and the predicates in the program is kept, allowing reducing the re-computation to the part of the analysis that corresponds to the affected predicates, and only propagating it to the rest of the analysis graph if necessary.

3 VeriFly: The On-the-fly IDE Integration

Fig. 1 shows the overall architecture of the CiaoPP verification framework integrated with the new IDE components, represented by the new box to the left, and the communication to and from CiaoPP. As mentioned before, the tool interface is implemented within Emacs and the on-the-fly support is provided by the Emacs "flycheck" package. Flycheck is an extension developed for GNU Emacs originally designed for on-the-fly *syntax* checking, but we use it here in a semantic context. However, as also mentioned before, a similar integration is possible with any reasonably extensible IDE.

[5] As mentioned before, abstract domains are defined as plug-ins which provide the basic abstract domain lattice operations and transfer functions, and are made accessible to the generic fixpoint computation component.

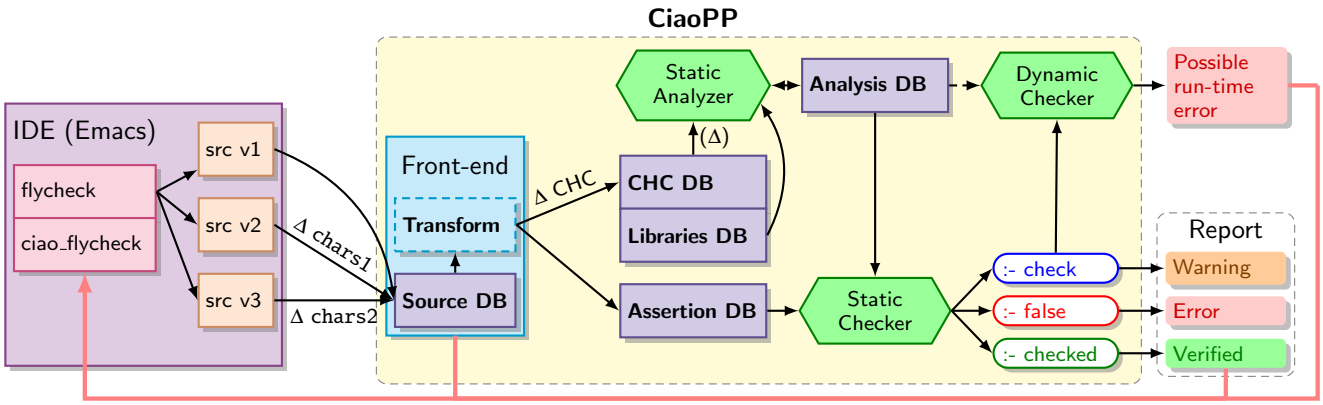

Figure 1: Integration of the CiaoPP framework in the Emacs-based IDE.

The CiaoPP framework runs in the background in daemon mode. When a file is opened, modified (and after some small period of inactivity), or saved, an editor checking event is triggered. Edit events notify CiaoPP (via a lightweight client) about program changes. The CiaoPP daemon will receive these changes, and, behind the scenes, transform them into changes at the CHC IR level (also checking for syntactic errors), and then incrementally (re-)analyze the code and (re-)check any reachable assertions. This can be code and assertions in libraries, other modules, language built-in specifications, or user-provided assertions. The results (errors, verifications, and warnings), from static (and possibly also dynamic checks) are returned to the IDE and presented as colorings and tooltips directly on the program text. This overall behavior is what we have called in our tool the "VeriFly" mode of operation.

In general, CiaoPP can be run fully automatically and does not *require* the user to change the configuration or provide invariants or assertions, and, for example, selects automatically abstract domains, as mentioned before. However, the system does have a configuration interface that allows manual selection of abstract domains, and of many parameters such as whether passes are incremental or not, the level of context sensitivity, error and warning levels, the type of output, etc. Appendix A shows the option browsing interface of the tool, as well as some options (abstract domain selections) in the menus for the cost analysis, shape and type analysis, pointer (logic variable) aliasing analysis, and numeric analysis.

4 Some Examples

We now show some simple examples of the system in action.[6] Figure 2 shows an assertion being verified within a medium-sized program implementing an automatic program parallelizer. The `add_annotations` loop traverses recursively a list of blocks and transforms sequential sections into parallel expressions. Upon opening the file the assertion is underlined in green, meaning that it has been verified (`checked` status). This ensures that upon entering the procedure there is no variable (pointer) sharing between `Info` (the input) and `Phrase`, i.e., `indep(Info,Phrase)`; that `Phrase` will arrive always as a free variable; and that on output from the procedure, `Ind` and `Gnd` will be ground terms (i.e., will contain no null pointers). Furthermore, this procedure is guaranteed not to fail. The corresponding information is also highlighted in the tool-tip (in yellow). In Appendix B we provide additional examples: two cases of a program that does quick-sort using open ended lists to construct the output lists (i.e., using pointers to append in constant time). In the first case the assertions are checked, and in the second one a warning is issued because an erroneous change eliminated a binding (i.e., a pointer assignment, shown commented out). In this case, a definite error is not proved, but the fact that the assertion cannot be discharged points at the cause. Examples 7, 8, and 9 show static detection of, respectively, a property incompatibility bug, an illegal call to a library predicate, and a simple non-termination.

[6]The system is demonstrated in the workshop talk, showing additional examples.

```
rewrite( clause(H,B), clause(H,P),I,G,Info) :-
    numbervars_2(H,0,Lhv),
    collect_info(B,Info,Lhv,_X,_Y),
    add_annotations(Info,P,I,G),!.

:- pred add_annotations(Info,Phrase,Ind,Gnd)
   :  (var(Phrase),indep(Info,Phrase))
   => (ground(Ind),ground(Gnd))
   +  not_fails.

add_annotations([],[],_,_).
add_annotations([I|Is],[P|Ps],Indep,Gnd) :-
    add_annotations(I,P,Indep,Gnd),
    add_annotations(Is,Ps,Indep,Gnd).
add_annotations(Info,Phrase,I,G) :- !,
    para_phrase(Info,Code,Type,Vars,I,G),
    make_CGE_phrase( Type,Code,Vars,PCode,I
    (       var(Code),!,
            Phrase = PCode
    ;       Vars = [],!,
            Phrase = Code
    ;       Phrase = (PCode,Code)
    ).
```

```
➢ Verified assertion:
:- check calls add_annotations(Info,Phrase,Ind,Gnd)
   : ( var(Phrase), indep(Info,Phrase) ).
➢ Verified assertion:
:- check comp add_annotations(Info,Phrase,Ind,Gnd)
   : ( var(Phrase), indep(Info,Phrase) )
   + not_fails.
➢ Verified assertion:
:- check success add_annotations(Info,Phrase,Ind,Gnd)
   : ( var(Phrase), indep(Info,Phrase) )
   => ( ground(Ind), ground(Gnd) ).
```

Figure 2: An assertion within a parallelizer (`ann`).

5 Some Performance Results

We conducted an initial evaluation of our tool with a classic benchmark application (`chat-80`,[7] 5.2k LOC across 27 files), written in Ciao Prolog. The experiments consisted in loading a specific module, turning on the checking of assertions in the IDE, selecting global analysis, i.e., analyzing the whole application, and performing a series of small edits. Concretely, we selected the modules `aggreg`, `readin`, and `talkr`, and performed clause and assertion edits, which we mark with the suffixes `-cls` and `-asr`, respectively in Table 2. To study whether incrementality improves response times significantly, we included experiments enabling and disabling it. The experiments were performed in a MacBook Air with the Apple M1 chip and 16 GB of RAM. The domain chosen for the experiment is a classic pair sharing+groundness domain [21], which was precise enough to prove many (existing) assertions in the application. Table 2 shows the time taken to check assertions in the different experiments with the incremental (`inc`) and non incremental (`non-inc`) analysis settings, splitting the time of the phases in which incrementality makes a difference. The last line of each setting reports the average *total* roundtrip assertion checking time, *measured from the IDE*, that is, what the programmer actually perceives. In the `non-inc` setting, the first row shows the time spent analyzing, which is almost $2/3$ of the total time. In the `inc` setting, we have split the total time into *diff*, which consists on determining which parts of the analysis result needs to be recomputed; *restore*, which restores a previous analysis (note that this is necessary in case different files are open at the same time in the IDE); and *(re)analysis*, which is the time required to recompute the analysis results. Note that this analysis time is down to $1/3$ of the total time (vs. $2/3$ in the `non-inc` setting). Aside from the data in the table, we observed a constant overhead of 0.4s for loading the code—parsing and prior transformations—in the tool.[8] This is currently not incremental and could be optimized to load only the parts that change. Verification times are negligible w.r.t. analysis times and are approx. 0.1s; this is also non incremental, since we found that it is not currently a bottleneck. Lastly, note that when only assertions are modified and only properties covered by the already run domains are introduced (last three columns), the analysis does not need to be recomputed, and restoring a previous analysis is enough ($< 1/4$ of the total time).

[7] `https://github.com/ciao-lang/chat80`

[8] We expect those numbers to be highly reduced with a finely-tuned parser and database implementation. Nevertheless, this work shows that the integration is feasible in practice.

		aggreg-cls	readin-cls	talkr-cls	aggreg-asr	readin-asr	talkr-asr
non-inc	analysis	2.0	2.0	1.8	2.1	2.1	2.1
	total	**3.0**	**2.4**	**3.2**	**2.9**	**2.4**	**2.9**
inc	diff	0.1	0.1	0.1	0.1	0.1	0.1
	restore	0.2	0.2	0.2	0.2	0.2	0.2
	(re)analysis	0.4	0.4	0.3	–	–	–
	total	**1.7**	**1.6**	**1.7**	**1.3**	**1.2**	**1.3**

Table 2: Average assertion checking time (seconds) for the experiments split by action.

6 Conclusions

We have shown how the integration of the CiaoPP static analysis and verification framework within an integrated development environment (IDE) can take advantage of incrementality to achieve a high level of reactivity at different levels of granularity. Our initial experience with the integrated tool shows quite promising results, with low latency times that provide early, continuous, and precise "on-the-fly" semantic feedback to programmers during the development process. This allows detecting many types of errors including swapped variables, property incompatibilities, illegal calls to library predicates, violated numeric constraints, unintended behaviour w.r.t. termination, resource usage, determinism, covering and failure (exceptions), etc. While presented using the Emacs and the `flycheck` package, our VeriFly techniques and results should be applicable to any integration into a modern extensible IDE. We plan to continue to work to achieve further reactivity and scalability improvements, enhanced presentations of verification results, and improved diagnosis.

References

[1] F. Bueno, D. Cabeza, M. V. Hermenegildo & G. Puebla (1996): *Global Analysis of Standard Prolog Programs*. In: *ESOP*.

[2] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. V. Hermenegildo, J. Maluszynski & G. Puebla (1997): *On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs*. In: *Proc. of the 3rd Int'l. Workshop on Automated Debugging–AADEBUG'97*, U. of Linköping Press, Linköping, Sweden, pp. 155–170. Available at `ftp://cliplab.org/pub/papers/aadebug_discipldeliv.ps.gz`.

[3] P. Cousot & R. Cousot (1977): *Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In: *ACM Symposium on Principles of Programming Languages (POPL'77)*, ACM Press, pp. 238–252.

[4] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi & Maurizio Proietti (2015): *Semantics-based generation of verification conditions by program specialization*. In: *17th International Symposium on Principles and Practice of Declarative Programming*, ACM, pp. 91–102, doi:10.1145/2790449.2790529.

[5] J. Gallagher, M. V. Hermenegildo, B. Kafle, M. Klemen, P. Lopez-Garcia & J.F. Morales (2020): *From big-step to small-step semantics and back with interpreter specialization (invited paper)*. In: *International WS on Verification and Program Transformation (VPT 2020)*, EPTCS, Open Publishing Association, pp. 50–65.

[6] I. Garcia-Contreras, J. F. Morales & M. V. Hermenegildo (2021): *Incremental and Modular Context-sensitive Analysis*. *Theory and Practice of Logic Programming*, pp. 1–48, doi:10.1017/S1471068420000496. Available at `https://arxiv.org/abs/1804.01839`. ArXiv:1804.01839.

[7] I. Garcia-Contreras, J.F. Morales & M. V. Hermenegildo (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)*, LNCS, Springer-Verlag, pp. 36–56, doi:10.1007/978-3-030-45260-5_3.

[8] M. Gómez-Zamalloa, E. Albert & G. Puebla (2009): *Decompilation of Java Bytecode to Prolog by Partial Evaluation*. *JIST* 51, pp. 1409–1427.

[9] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea & Andrey Rybalchenko (2012): *Synthesizing software verifiers from proof rules*. In Jan Vitek, Haibo Lin & Frank Tip, editors: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*, ACM, pp. 405–416, doi:10.1145/2254064.2254112.

[10] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli & Jorge A. Navas (2015): *The SeaHorn Verification Framework*. In: *International Conference on Computer Aided Verification, CAV 2015*, *LNCS* 9206, Springer, pp. 343–361.

[11] Kim S. Henriksen & John P. Gallagher (2006): *Abstract Interpretation of PIC Programs through Logic Programming*. In: *SCAM '06*, IEEE Computer Society, pp. 184–196.

[12] M. V. Hermenegildo, F. Bueno, M. Carro, P. Lopez-Garcia, E. Mera, J.F. Morales & G. Puebla (2012): *An Overview of Ciao and its Design Philosophy*. *Theory and Practice of Logic Programming* 12(1–2), pp. 219–252, doi:doi:10.1017/S1471068411000457. Available at `http://arxiv.org/abs/1102.5497`.

[13] M. V. Hermenegildo, G. Puebla & F. Bueno (1999): *Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging*. In: *The Logic Programming Paradigm*, Springer, pp. 161–192.

[14] M. V. Hermenegildo, G. Puebla, F. Bueno & P. Lopez-Garcia (2005): *Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor)*. *Science of Computer Programming* 58(1–2), pp. 115–140, doi:10.1016/j.scico.2005.02.006.

[15] M. V. Hermenegildo, G. Puebla, K. Marriott & P. Stuckey (2000): *Incremental Analysis of Constraint Logic Programs*. *ACM TOPLAS* 22(2), pp. 187–223.

[16] Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez & Martin Schäf (2016): *JayHorn: A Framework for Verifying Java Programs*. In Swarat Chaudhuri & Azadeh Farzan, editors: *Computer Aided Verification - 28th International Conference, CAV 2016*, *LNCS* 9779, Springer, pp. 352–358, doi:10.1007/978-3-319-41528-4_19.

[17] U. Liqat, Z. Banković, P. Lopez-Garcia & M. V. Hermenegildo (2018): *Inferring Energy Bounds via Static Program Analysis and Evolutionary Modeling of Basic Blocks*. In: *Logic-Based Program Synthesis and Transformation - 27th International Symposium*, *LNCS* 10855, Springer.

[18] U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, M. V. Hermenegildo, J. P. Gallagher & K. Eder (2016): *Inferring Parametric Energy Consumption Functions at Different Software Levels: ISA vs. LLVM IR*. In: *Proc. of FOPARA*, *LNCS* 9964, Springer, pp. 81–100, doi:10.1007/978-3-319-46559-3_5.

[19] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. V. Hermenegildo & K. Eder (2014): *Energy Consumption Analysis of Programs based on XMOS ISA-Level Models*. In: *Proceedings of LOPSTR'13*, *LNCS* 8901, Springer, pp. 72–90, doi:10.1007/978-3-319-14125-1_5.

[20] P. Lopez-Garcia, L. Darmawan, M. Klemen, U. Liqat, F. Bueno & M. V. Hermenegildo (2018): *Interval-based Resource Usage Verification by Translation into Horn Clauses and an Application to Energy Consumption*. *Theory and Practice of Logic Programming, Special Issue on Computational Logic for Verification* 18(2), pp. 167–223. Available at `https://arxiv.org/abs/1803.04451`.

[21] K. Marriott & H. Søndergaard (1993): *Precise and Efficient Groundness Analysis for Logic Programs*. Technical Report 93/7, Univ. of Melbourne.

[22] M. Méndez-Lojo, J. Navas & M. Hermenegildo (2007): *A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs*. In: *LOPSTR*, *LNCS* 4915, Springer-Verlag, pp. 154–168, doi:10.1007/978-3-540-78769-3_11.

[23] E. Mera, P. Lopez-Garcia, M. Carro & M. V. Hermenegildo (2008): *Towards Execution Time Estimation in Abstract Machine-Based Languages*. In: *PPDP'08*, ACM Press, pp. 174–184, doi:10.1145/1389449.1389471.

[24] J. Navas, M. Méndez-Lojo & M. Hermenegildo (2008): *Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications*. In: *The Sixth NASA Langley Formal Methods Workshop (LFM 08)*, pp. 29–32. Extended Abstract.

[25] J. Navas, M. Méndez-Lojo & M. V. Hermenegildo (2009): *User-Definable Resource Usage Bounds Analysis for Java Bytecode*. In: *BYTECODE'09*, *ENTCS* 253, Elsevier, pp. 6–86. Available at `http://cliplab.org/papers/resources-bytecode09.pdf`.

[26] J.C. Peralta, J. Gallagher & H. Sağlam (1998): *Analysis of Imperative Programs through Analysis of Constraint Logic Programs*. In G. Levi, editor: *Static Analysis. 5th International Symposium, SAS'98, Pisa*, *LNCS* 1503, pp. 246–261.

[27] V. Perez-Carrasco, M. Klemen, P. Lopez-Garcia, J.F. Morales & M. V. Hermenegildo (2020): *Cost Analysis of Smart Contracts via Parametric Resource Analysis*. In: *Static Aanalysis Simposium (SAS'20)*, *LNCS* 12389, Springer, pp. 7–31.

[28] G. Puebla, F. Bueno & M. V. Hermenegildo (2000): *An Assertion Language for Constraint Logic Programs*. In: *Analysis and Visualization Tools for Constraint Programming*, *LNCS* 1870, Springer-Verlag, pp. 23–61.

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

[30] G. Puebla & M. V. Hermenegildo (1996): *Optimized Algorithms for the Incremental Analysis of Logic Programs*. In: *SAS'96*, Springer LNCS 1145, pp. 270–284.

[31] K. Rustan, M. Leino & Valentin Wüstholz (2015): *Fine-Grained Caching of Verification Results*. In Daniel Kroening & Corina S. Pasareanu, editors: *Proc. of the 27th International Conference on Computer Aided Verification, CAV 2015*, *LNCS* 9206, Springer, pp. 380–397, doi:10.1007/978-3-319-21690-4_22.

[32] Julian Tschannen, Carlo A. Furia, Martin Nordio & Bertrand Meyer (2011): *Usable Verification of Object-Oriented Programs by Combining Static and Dynamic Techniques*. In Gilles Barthe, Alberto Pardo & Gerardo Schneider, editors: *Proc. of the 9th International Conference on Software Engineering and Formal Methods, SEFM 2011*, *LNCS* 7041, Springer, pp. 382–398, doi:10.1007/978-3-642-24690-6_26.

A CiaoPP option browser and some abstract domains

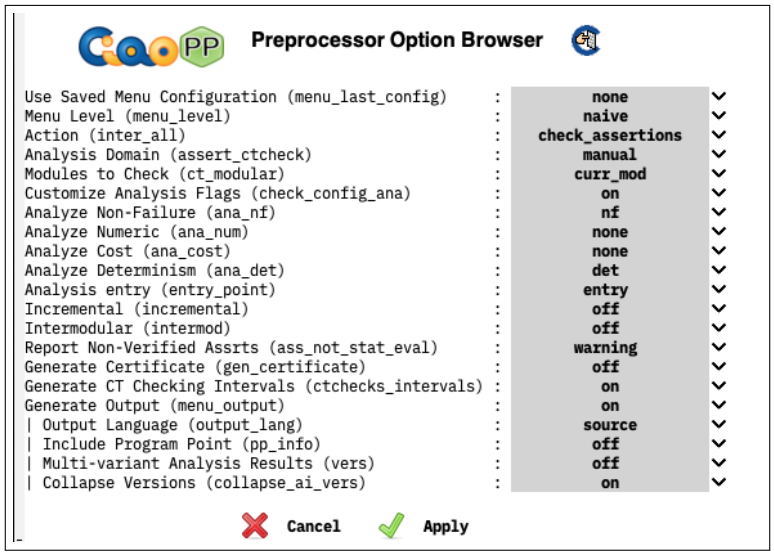

Figure 3: The CiaoPP option browser.

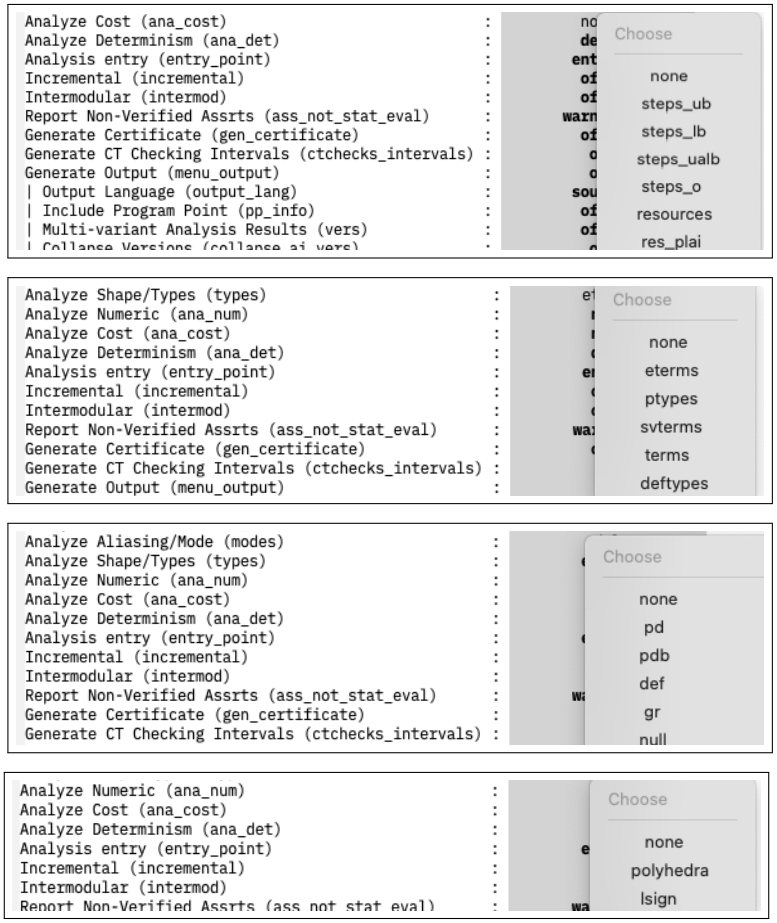

Figure 4: Some options (abstract domains) in the menus for the cost analysis, shape and type analysis, pointer (logic variable) aliasing analysis, and numeric analysis.

B Some additional examples

```
:- module(_,[qsort/2],[assertions,nativeprops]).

:- entry qsort(X,Y) : (ground(X),list(X),var(Y)).


:- pred qsort(X,Y) => ground(Y).

qsort(X,Y) :- qsort_(X,Y,T), T=[].


:- pred qsort_(X,Y,Z) : (list(X),var(Y),var(Z),indep(Y,Z)) => ground(X).

qsort_([], Result, Result).
qsort_([First|Rest],ResultB,ResultE) :-
    partition(Rest,First,Sm,Lg),
    qsort_(Sm,SmB,SmE),
    qsort_(Lg,LgB,LgE),
    ResultB=SmB,
    SmE=[First|LgB],
    ResultE=LgE.


:- pred partition(L,P,Lg,Sm) => (list(Lg), list(Sm), ground(Lg), ground(Sm)).

partition([],_,[],[]).
partition([X|Y],F,[X|Y1],Y2) :-
    X @=< F,
    partition(Y,F,Y1,Y2).
partition([X|Y],F,Y1,[X|Y2]) :-
    X @> F,
    partition(Y,F,Y1,Y2).
```

```
➤ Verified assertion:
:- check calls qsort_(X,Y,Z)
   : ( list(X), var(Y), var(Z), indep(Y,Z) ).
➤ Verified assertion:
:- check success qsort_(X,Y,Z)
   : ( list(X), var(Y), var(Z), indep(Y,Z) )
   => ground(X).
```

Figure 5: Sorting with incomplete data structures – assertions checked.

```
:- module(_,[qsort/2],[assertions,nativeprops]).

:- entry qsort(X,Y) : (ground(X),list(X),var(Y)).


:- pred qsort(X,Y) => ground(Y).

qsort(X,Y) :- qsort_(X,Y,T), T=[].


:- pred qsort_(X,Y,Z) : (list(X

qsort_([], Result, Result).
qsort_([First|Rest],ResultB,Res
    partition(Rest,First,Sm,Lg),
    qsort_(Sm,SmB,SmE),
    qsort_(Lg,LgB,LgE),
    ResultB=SmB,
%    SmE=[First|LgB],
    ResultE=LgE.


:- pred partition(L,P,Lg,Sm) => (list(Lg), list(Sm), ground(Lg), ground(Sm)).

partition([],_,[],[]).
partition([X|Y],F,[X|Y1],Y2) :-
    X @=< F,
    partition(Y,F,Y1,Y2).
partition([X|Y],F,Y1,[X|Y2]) :-
    X @> F,
    partition(Y,F,Y1,Y2).
```

```
➤ Could not verify assertion:
:- check success qsort(X,Y)
   => ground(Y).
because
    term_typing:ground(Y)
could not be derived from inferred success:
[shfr]   native_props:mshare([[Y]]),term_typing:ground([X])
[eterms] basic_props:list(X),basic_props:term(Y)
➤ Verified assertion:
:- check calls qsort(X,Y).
```

Figure 6: Sorting with incomplete data structures – bug found (missing binding / pointer assignment).

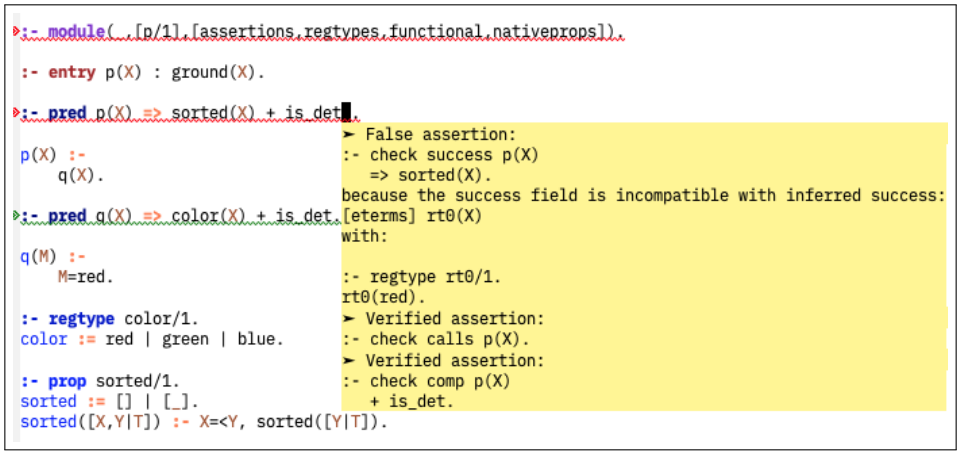

Figure 7: A property incompatibility bug detected statically.

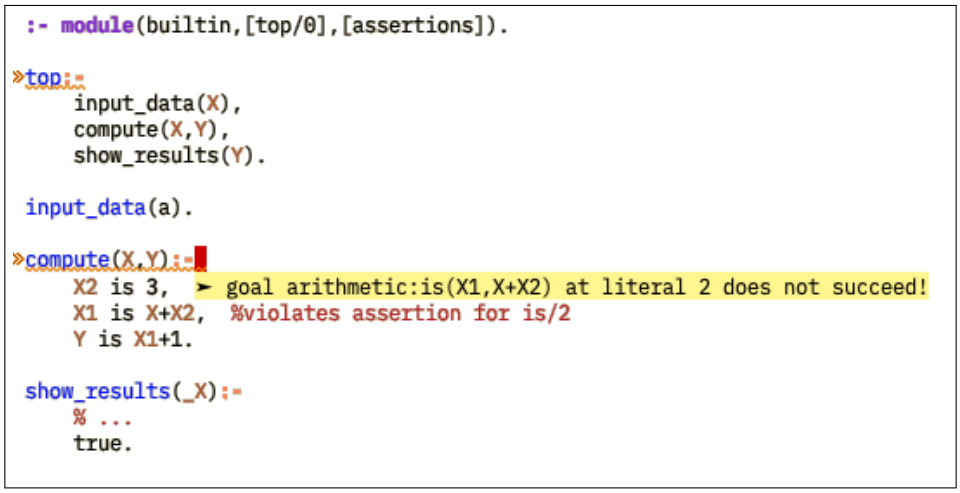

Figure 8: Statically detect illegal call to library predicate.

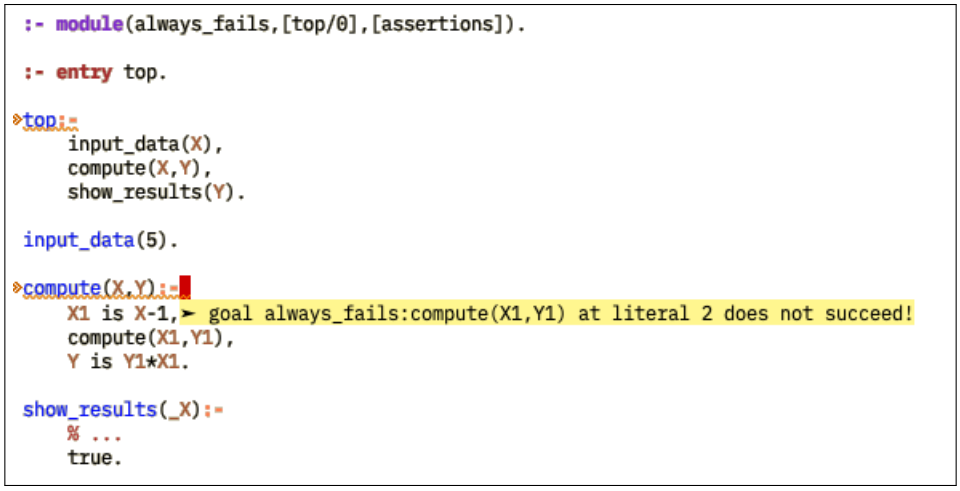

Figure 9: Static detection of simple non-termination.