TPLP **21** (6): 768–784, 2021. © The Author(s), 2021. Published by Cambridge University Press
doi:10.1017/S1471068421000430

VeriFly: *On-the-fly Assertion Checking via Incrementality*

MIGUEL A. SANCHEZ-ORDAZ, ISABEL GARCIA-CONTRERAS, VICTOR PEREZ and JOSÉ F. MORALES
IMDEA Software Institute and Universidad Politécnica de Madrid (UPM)
(*e-mails:* ma.sanchez.ordaz@imdea.org, isabel.garcia@imdea.org, victor.perez@imdea.org, josef.morales@imdea.org)

PEDRO LOPEZ-GARCIA
IMDEA Software Institute and Spanish Council for Scientific Research (CSIC)
(*e-mail:* pedro.lopez@imdea.org)

MANUEL V. HERMENEGILDO
IMDEA Software Institute and Universidad Politécnica de Madrid (UPM)
(*e-mail:* manuel.hermenegildo@imdea.org)

submitted 1 January 2003 revised 30 August 2021; accepted 31 August 2021

Abstract

Assertion checking is an invaluable programmer's tool for finding many classes of errors or verifying their absence in dynamic languages such as Prolog. For Prolog programmers, this means being able to have relevant properties, such as modes, types, determinacy, nonfailure, sharing, constraints, and cost, checked and errors flagged without having to actually run the program. Such global static analysis tools are arguably most useful the earlier they are used in the software development cycle, and fast response times are essential for interactive use. 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 our static analysis and verification framework, this challenge is addressed through a combination of modular and 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 combination 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). We believe that similar extensions are also reproducible with low effort in other mature development environments. 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. The tool supports Prolog natively, as well as other languages by semantic transformation into Horn clauses.

KEYWORDS: On-the-fly assertion checking, program development environments, static analysis, abstract interpretation, incremental analysis, logic and constraint programming

* Research partially funded by MINECO MICINN PID2019-108528RB-C21 *ProCode* project, FPU grant 16/04811, the Madrid M141047003 *N-GREENS* and P2018/TCS-4339 *BLOQUES-CM* programs, and the Tezos foundation. We are also grateful to the anonymous reviewers for their comments.

https://doi.org/10.1017/S1471068421000430 Published online by Cambridge University Press

1 Introduction

Global static analysis and verification tools can greatly help developers detect high-level, semantic errors in programs or certify their absence. This is specially the case for high-level, dynamic languages such as Prolog, where obtaining information, such as modes, types, determinacy, nonfailure, cardinality, variable sharing in structures, constraints, termination, cost, and checking it against specifications, in the form of annotations or assertions, can be invaluable aids to programmers.

Arguably, such tools are more effective the earlier the stage in which they are applied within the software development process. Particularly useful is their application 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 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.

Our approach builds on the CiaoPP program development framework (Bueno *et al.* 1997; Hermenegildo *et al.* 1999; Puebla *et al.* 2000b; Hermenegildo *et al.* 2005; Garcia-Contreras *et al.* 2021), 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 (Cousot and Cousot 1977). This framework supports natively Prolog and several extensions within the LP and CLP paradigms and can also be applied to other high- and low-level languages, using the now well-understood technique of semantic translation into intermediate Horn clause-based representation. 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 our framework, 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 interprocedural analysis *incrementally*, that is, reactively to fine-grain edits, avoiding re-analyses 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 the static analysis and verification framework within an integrated development environment (IDE) takes advantage of the incremental and modular features 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 and reuses in part off-the-shelf "on-the-fly" *syntax* checking capabilities offered by the Emacs flycheck package. Emacs was chosen because it is a solid platform and preferred by many experienced users. However, this low-maintenance approach should be

https://doi.org/10.1017/S1471068421000430 Published online by Cambridge University Press

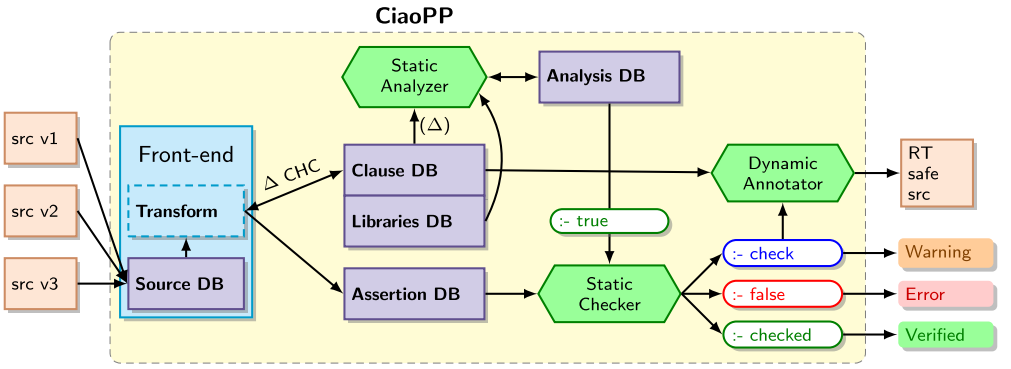


Fig. 1: Architecture of the CiaoPP framework.

easily reproducible in other modern extensible editors and mature program development environments.

2 The CiaoPP framework

We start by providing an informal overview of the components and operation of the CiaoPP framework (Figure 1).

Front end. Before getting into the analysis and verification phases, the tool's front end transforms any syntactic or semantic extensions used in the input program (such as functional notation, or, more specifically for our discussion, the syntactic sugar related to assertions) in order to reduce each module to the intermediate representation that the framework works with. This representation, which we refer to as the HC IR, is fundamentally plain Horn clauses (plain Prolog) extended with the (canonical part) of the assertion language, reviewed in the next section. Although beyond the scope of this paper, as mentioned in the introduction, the framework can also be applied to other input languages, outside (C)LP, which is done by translating such input programs to the same HC IR (Méndez-Lojo *et al.* 2007).[1] However, herein we concentrate on the native use of the tool for (C)LP programs. The examples will be written in Ciao Prolog (Hermenegildo *et al.* 2012; Bueno *et al.* 2021)[2] using its assertions library and some additional syntactic sugar such as functional notation, but the tool can easily be adapted to process source files for other Prolog flavors and systems. The framework itself is also written in Ciao.

The assertion language is a fundamental element of our framework (Bueno *et al.* 1996; Hermenegildo *et al.* 1999; Puebla *et al.* 2000a). Such assertions can express a wide

[1] This approach is used nowadays in many analysis and verification tools (Peralta *et al.* 1998; Henriksen and Gallagher 2006; Méndez-Lojo *et al.* 2007; Gómez-Zamalloa *et al.* 2009; Grebenshchikov *et al.* 2012; Gurfinkel *et al.* 2015; De Angelis *et al.* 2017; Lopez-Garcia *et al.* 2018; Perez-Carrasco *et al.* 2020; Gallagher *et al.* 2020). The front end is also in charge of these translations, as well as of translating the analysis and verification results back to the source language. Techniques such as partial evaluation and program specialization offer powerful methods to obtain such translations with provable correctness – see the work by De Angelis *et al.* (2021) for a recent survey.

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

https://doi.org/10.1017/S1471068421000430 Published online by Cambridge University Press

range of properties, including functional (state) properties, such as modes, types, sharing/aliasing, constraints, etc., as well as non-functional (i.e., global, computational) properties, such as determinacy, non-failure, cardinality, or resource usage (energy, time, memory, ...). The set of properties 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, other parts of the framework, or other tools), providing assumptions, describing unknown code, generating test cases automatically, or producing documentation.

PREDICATE-LEVEL ASSERTIONS allow describing *sets* of *preconditions* and *conditional postconditions* on the state for a given predicate (as well as global properties). In particular, `pred` assertions are of the form:

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

where *Head* is a predicate descriptor that denotes the predicate that the assertion applies to, and *Pre* and *Post* are conjunctions of *state property* literals (the modes, types, constraints, etc.), which can include arguments, and these can also be variables from *Head*. Such properties are predicates, typically written in (subsets of) the source language, and thus runnable, so that they can be used as run-time checks, and which (see "Assertion verification" later in this section) may be abstracted and inferred by some domain in CiaoPP. They can be imported from system libraries or user-defined. *Pre* expresses properties that hold when *Head* is called. *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. *Comp* describes properties of the whole computation, such as determinism, nonfailure, resource usage,and termination and they also apply to calls to the predicate that meet *Pre*. Assertions are optional, and there can be multiple `pred` assertions for a predicate. This allows describing different behaviors for the same predicate for different call substitutions, that is, different *modes*. If multiple `pred` assertions are present for a given predicate, their *Pre* fields then describe the set of all admissible calls to the predicate, that is, at least one *Pre* must hold for each call to *Head*.

Status is a qualifier of the meaning of the assertion. Here we consider (in the context of static assertion checking) the following *Status*es (cf. Figure 1):

- `check`: the assertion expresses properties that must hold at run-time, that is, 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.

Additionally, assertions with `trust` status can be used to provide the analyzer with information that it should *assume*. This can be useful for, for example, describing external or unknown code or improving analysis precision.

Figure 2 provides some examples of assertions. The syntax that allows including the `pred` assertions is made available through the `assertions` package (packages are similar to modules but oriented towards providing syntactic and semantic extensions).

https://doi.org/10.1017/S1471068421000430 Published online by Cambridge University Press

```
1  :- module(_, [nrev/2], [assertions, nativeprops, functional]).
2  :- entry nrev/2 : {list, ground} * var.
3  :- use_module(someprops).
4
5  :- pred nrev(A, B) : list(A) => list(B)
6     + (not_fails,is_det,terminates,steps_o(exp(length(A),2))).
7  nrev([])    := [].
8  nrev([H|L]) := ~conc(nrev(L),[H]).
9
10 :- pred conc(A,B,C) : list(A) => size_ub(C,length(A)+length(B)) + steps_o(length(A)).
11 conc([],    L) := L.
12 conc([H|L], K) := [ H | conc(L,K) ].
```

Fig. 2: Naive reverse with some assertions.

```
1  :- module(someprops, _, [functional, hiord, assertions]).
2
3  :- prop color/1.
4  color := red | blue | green. % Equivalent to: color(red). color(blue). color(green).
5
6  :- prop list/1.
7  list := [] | [_ | list].     % Equivalent to: list([]). list([_|T]) :- list(T).
8
9  :- prop list/2.
10 list(T) := [] | [~T | list(T)].
11
12 :- prop sorted/1.
13 sorted := [] | [_].
14 sorted([X,Y|Z]) :- X @< Y, sorted([Y|Z]).
```

Fig. 3: Examples of state **prop**erty definitions.

Predicates `nrev/2` and `conc/2` define naive reverse and are written using the `functional` notation package. The first assertion (lines 5 and 6) expresses that calls to `nrev/2` with the first argument bound to a list are admissible, and that if such calls succeed, then the second argument should also be bound to a list (as well as being non-failing, deterministic, terminating, and quadratic). Thus, such an assertion provides information on (a generalization of) types and modes, as well as determinism, failure, cost, etc.

Figure 3 provides some examples of state properties (**prop**s, for short), which, as mentioned before, are defined and exported/imported as normal predicates: for example, in Figure 2 `list/1`, `list/2`, or `color/1` are imported from the `someprops` user module while others (e.g., `size_ub/2`) from the system's `nativeprops` library. Properties need to be marked explicitly as such with a `:- prop` declaration, and this flags that they need to meet some restrictions (Puebla *et al.* 2000a; Bueno *et al.* 2021). For example, their execution should terminate for any possible call since, as discussed later, **prop**s will not only be checked at compile time, but may also be involved in run-time checks. Types are just a particular case (further restriction) of state properties. Different type systems are provided as libraries. Properties like `list/1` that are in addition regular types can be flagged as such: `:- prop list/1 + regtype.` or, more compactly, `:- regtype list/2.`

The combination of multiple assertions $\{a_1, \ldots, a_n\}$ for a given predicate, where $a_i =$ "`:- pred` *Head* `:` Pre_i `=>` $Post_i$`.`" is defined more precisely as follows: the set of *assertion conditions* for *Head* is $\{C_0, C_1, \ldots, C_n\}$, with:

$$C_0 = \mathsf{calls}(Head, \vee_{j=1}^{n} Pre_j), \text{ and}$$
$$C_i = \mathsf{success}(Head, Pre_i, Post_i), i = 1, \ldots, n$$

https://doi.org/10.1017/S1471068421000430 Published online by Cambridge University Press

where calls($Head, Pre$) states conditions on all concrete calls to the predicate described by *Head*, and success($Head, Pre_i, Post_i$) the conditions on the success constraints produced by calls to *Head* if Pre_i is satisfied. If the assertions a_i above, $i = 1, \ldots, n$, include a `+ Comp` field, then the set of *assertion conditions* also includes conditions of the form: comp($Head, Pre_i, Comp_i$), for $i = 1, \ldots, n$, that express properties of the whole computation for calls to *Head* if Pre_i is satisfied.

In addition to predicate-level assertions, PROGRAM-POINT ASSERTIONS are of the form:

Status (*StateFormula*) ,

and they can be placed at program locations where a new literal may be added. For example, for status `check` they should be interpreted as "whenever computation reaches a state corresponding to this program point, *StateFormula* should hold." For example,

```
check((list(color, A), var(B))),
```

is a program-point assertion that checks whether `A` is instantiated to a list of colors and `B` is a variable, where `A` and `B` are variables of the clause where the assertion appears.

Assertion-related syntactic sugar. In order to facilitate writing assertions, as well as compatibility with different systems, the assertion language also provides syntactic sugar such as *modes*, Cartesian product notation, markdown syntax, etc. For example, the following set of `pred` assertions use Cartesian product notation to provide information on a reversible sorting predicate:

```
:- pred sort/2 : list(num) * var => list(num)
                          * list(num) + is_det.
:- pred sort/2 : var * list(num) => list(num)
                     * list(num) + non_det.
```

(in addition, curly brackets can be used to group properties – see Figure 2). The assertion language also allows defining *modes*, which are macros that can be placed in argument positions of *Head* and expand to properties in different assertion fields. For example, using the `isomodes` library the assertions above can also be expressed as:

```
:- pred sort(+list(num), -list(num)) + is_det.
:- pred sort(-list(num), +list(num)) + non_det.
```

Or, if no types and only modes are used:

```
:- pred sort(+, -).
:- pred sort(-, +).
```

There is also an alternative encoding via "markdown-style" comments (provided by libraries `doccomments` and `markdown`) that allows writing, for example:

```
%! sort(+list(num),-list(num)): This predicate sorts.
```

Thanks to these libraries and the underlying syntactic expansion mechanisms, and their modular nature, it is easy to adapt the tool to other syntactic forms, on a per-module basis, such as for example supporting "Quintus-style" modes:

```
:- mode sort(+, -).
:- mode sort(-, +).
```

or the SWI-Prolog `pldoc`-style annotations for documentation, making all of them machine checkable.

Static program analysis. In order to compute safe over-approximations of the program semantics, the CiaoPP verification framework uses analyses based on abstract interpretation (the "Static Analyzer" in Figure 1). Given a program P and a set of queries $\mathcal{Q}$, an

https://doi.org/10.1017/S1471068421000430 Published online by Cambridge University Press

analysis graph is inferred that abstracts the (possibly infinite) set of (possibly infinite) execution trees. The analysis result is denoted by $[\![P]\!]^{\alpha}_{\mathcal{Q}}$, where α is the abstraction(s) performed. By default, the abstract domains to be used are automatically selected, depending on the properties that appear in the assertions, but manual selection is also possible. 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, and can be represented by (`true`) assertions. For our purposes, and without loss of generality, we treat $[\![P]\!]^{\alpha}_{\mathcal{Q}}$ as a set of tuples (one per node): $\{\langle L_1, \lambda^c_1, \lambda^s_1\rangle, \ldots, \langle L_n, \lambda^c_n, \lambda^s_n\rangle\}$. In each $\langle L_i, \lambda^c_i, \lambda^s_i\rangle$ triple, L_i is a predicate descriptor, and λ^c_i and λ^s_i are, respectively, the abstract call and success substitutions, elements of abstract domain D_α.[3] 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.

Assertion verification. To verify a program, the behaviors abstracted in $[\![P]\!]^{\alpha}_{\mathcal{Q}}$ are compared with program assertions (the "Static Checker" in Figure 1). The verification results are reported as changes in the status and transformations of the assertions: `checked`, if the properties are satisfied; `false` if some property was proved not to hold; or `check` if it was not possible to determine any the first two, in which case run-time checks will be included in the program to make it run-time safe (Figure 1). Since in our framework the intended semantics is also specified by using predicates, some of the properties may be undecidable and also not exactly representable in the abstract domains. However, it is possible to under- and over-approximate them, denoted by $I^{\alpha-}$ and $I^{\alpha+}$. Such approximations are always computable by choosing the closest element in the abstract domain. At the limit $\bot$ and $\top$ are, respectively, an under-approximation and an over-approximation of any specification. Let $Ren(p(\bar{v}), [\![P]\!]^{\alpha}_{\mathcal{Q}})$, where $p(\bar{v})$ is a predicate descriptor, denote the set of tuples resulting from the static analysis for predicate p *in* P w.r.t. $\mathcal{Q}$ renamed to variables $\bar{v}$, that is: $Ren(p(\bar{v}), [\![P]\!]^{\alpha}_{\mathcal{Q}}) = \{\langle p(\bar{v}), \lambda^c\sigma, \lambda^s\sigma\rangle \mid \exists \langle p(\bar{v}'), \lambda^c, \lambda^s\rangle \in [\![P]\!]^{\alpha}_{\mathcal{Q}}$ and σ is a renaming substitution s.t. $p(\bar{v}) = p(\bar{v}')\sigma\}$.

The following are some sufficient conditions, adapted from the works of Puebla *et al.* (2000b) and Pietrzak *et al.* (2006), which allow deciding whether assertions are `checked` or `false` with respect to the concrete semantics, based on the abstract semantics:

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

https://doi.org/10.1017/S1471068421000430 Published online by Cambridge University Press

- calls$(p(\bar{v}), Pre)$ is *checked* (resp. *false*) for predicate $p \in P$ w.r.t. $\mathcal{Q}$ if $\forall \langle p(\bar{v}), \lambda^c, \lambda^s \rangle \in Ren(p(\bar{v}), [\![P]\!]^{\alpha}_{\mathcal{Q}}) : \ \lambda^c \sqsubseteq Pre^{\alpha-}$ (resp. $\lambda^c \sqcap Pre^{\alpha+} = \bot$).
- success$(p(\bar{v}), Pre, Post)$ is *checked* (resp. *false*) for predicate $p \in P$ w.r.t. $\mathcal{Q}$ if $\forall \langle p(\bar{v}), \lambda^c, \lambda^s \rangle \in Ren(p(\bar{v}), [\![P]\!]^{\alpha}_{\mathcal{Q}}) : \ ((\lambda^c \sqcap Pre^{\alpha+} = \bot) \vee (\lambda^s \sqsubseteq Post^{\alpha-}))$ (resp. $\lambda^c \sqsubseteq Pre^{\alpha-} \wedge (\lambda^s \sqcap Post^{\alpha+} = \bot)$ and there is at least one actual call to predicate p that succeeds).

Supporting incrementality. In order to support incrementality, the analysis graph produced by the static analyzer is made persistent ("Analysis DB" in Figure 1), storing an abstraction of the behavior of each predicate and predicate (abstract) call dependencies. In turn, the "Front-end" (Figure 1) keeps track of and translates source code changes into, for example, 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 (Garcia-Contreras *et al.* 2021; 2020; Puebla and Hermenegildo 1996; Hermenegildo *et al.* 2000). 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

Figure 4 shows the overall architecture of VeriFly, the integration of the CiaoPP framework 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.[4] 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.

The overall architecture consists of a flycheck adaptor (implementing different Ciao-based checkers, from syntactic to full analysis), a CiaoPP process that runs in the background in daemon mode, and a lightweight client to CiaoPP. When a file is opened, modified, or saved, as well as after some small period of inactivity, an editor checking event is triggered. Edit events notify CiaoPP (via the lightweight client) about program changes, which can be both in code and assertions. The CiaoPP daemon receives these changes, and, behind the scenes, transforms them into changes at the HC IR level (also checking for syntactic errors), and then incrementally (re-)analyzes the code and (re-)checks any reachable assertions. The latter can be in libraries, other modules, language built-in specifications, or of course (but not necessarily) in user code. The results (errors, verifications, and warnings) from static (and possibly also dynamic checks) are

[4] https://github.com/flycheck/flycheck

https://doi.org/10.1017/S1471068421000430 Published online by Cambridge University Press

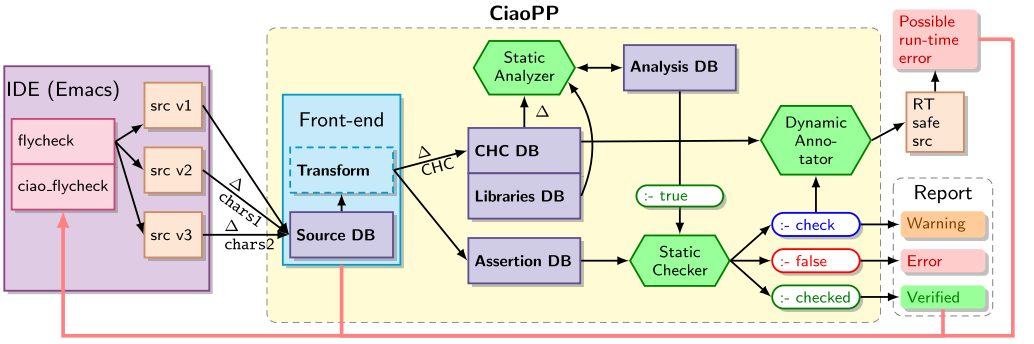


Fig. 4: Integration of the CiaoPP framework in the Emacs-based IDE.

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.

Details on the architecture. Currently, flycheck requires saving the contents of the source being edited (Emacs *buffer*) into a temporary file and then invoking an external command. In our case, the external command is a lightweight client that communicates with a running CiaoPP process, executing as a an *active module*, a daemon process that executes in the background and reacts to simple JSON-encoded queries via a socket.[5] This CiaoPP process is started once and kept alive for future analyses, ensuring that no time is unnecessarily wasted in startup and cleanups, as well as allowing caching some common analysis data, etc. for libraries. The approach is similar to LSP (Language Server Protocol). Finally, Ciao implements a "shadow module" mechanism that allows the compiler to read alternative versions for some given modules. We use this mechanism to make CiaoPP (and other Ciao-based checkers) read the contents of temporary Emacs buffers during edition (saved as temporary files with *shadowed* names). This is specially useful to work in (inter-)modular analysis where the analysis root is not necessarily the active buffer.

Customizing the analysis. 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, and the type of output. Figure 5 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.

VeriFly *in action.* We now show some simple examples of the system in action. Figure 6 shows an assertion being verified within a medium-sized program implementing an au-

[5] JSON-encoded interaction capabilities have been added recently to support tool interoperability and browser-based interactions and to simplify future extensions.

https://doi.org/10.1017/S1471068421000430 Published online by Cambridge University Press

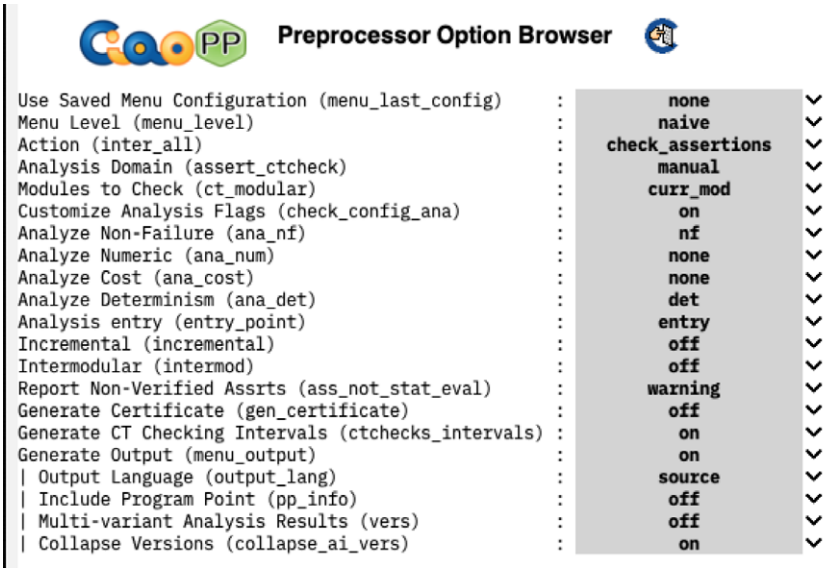


Fig. 5: The CiaoPP option browser.

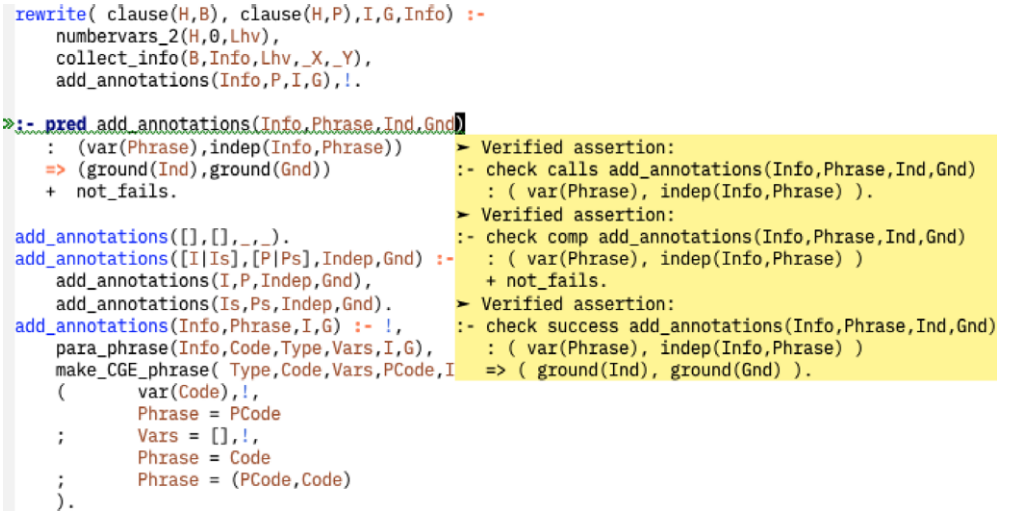


Fig. 6: An assertion within a parallelizer (`ann`).

tomatic 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`, that is, `indep(Info,Phrase)` holds; 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).

https://doi.org/10.1017/S1471068421000430 Published online by Cambridge University Press

In Figure 7, we show an implementation of quick-sort using open-ended ("difference") lists to construct the output lists (i.e., using pointers to append in constant time).

Figures 8, 9a, and 9b show static detection of, respectively, a property incompatibility bug (note that the concrete property `sorted/1` is approximated by the regtypes domain as `list/1` and an incompatibility is found with `rt0`, i.e., `red`), an illegal call to a library predicate, and a simple non-termination.

Figure 10 shows again the naive reverse example of Figure 2, modified to illustrate the detection of an error regarding unintended behavior w.r.t. cost. The assertion in lines 5 and 6 of Figure 10 (left) states that predicate `nrev` should be linear in the length of the (input) argument `A`. This is expressed by the property `steps_o(length(A))`, meaning that the cost, in terms of resolution steps, of any call to `nrev(A, B)` with `A` bound to a list and `B` a free variable, is in $O($`length(A)`$)$. However, this worst case asymptotic complexity stated in the user-provided assertion is incompatible with a safe, quadratic lower bound on the cost of such calls ($\frac{1}{2}\ length(A)^2 + \frac{3}{2}\ length(A) + 1$) inferred by the static analyzer. In contrast, assertion in lines 5 and 6 of Figure 11 (left) states that predicate `nrev` should have a quadratic worst case asymptotic complexity in the length of `A`, which is proved to hold by means of the upper-bound cost function inferred by analysis (which coincides with the lower bound above). Figure 11 also shows the verification of determinacy, non-failure, and termination properties.

4 Some performance results

We provide some performance results from our tool using the well-known `chat-80` program (https://github.com/ciao-lang/chat80) which contains 5.2k lines of Prolog code across 27 files and uses a number of system libraries containing different Prolog built-ins and library predicates. The experiments consisted in opening a specific module in the IDE, and activating the checking of assertions with global analysis, that is, analyzing the whole application as well as the libraries, and then performing a series of small edits, observing the total response time, that is, the time from edit to graphical update of assertion coloring in the IDE. Concretely, we performed two kinds of edits, predicate and assertions (**E1**), and only assertions (**E2**). The edits were performed on three selected files: `aggreg`, `readin`, and `talkr`. To study whether incrementality improves response times significantly, we included experiments enabling and disabling it. The experiments were performed using Ciao 1.20 on a MacBook Air with the Apple M1 chip and 16 GB of RAM. We evaluated the tool with three well-known abstract domains: a classic pair sharing (Marriott and Søndergaard 1993) (`pairSh`), a dependency tracking via propositional clauses domain (Dumortier *et al.* 1993) (`def`), and sharing+groundness with clique-based widening (Navas *et al.* 2006) (`ShGrC`). The latter is the most precise, and, hence, the most expensive. We used sharing/groundness domains because they are known to be costly and at the same time, beyond mode inference, necessary to ensure correctness of most other analyses in (C)LP systems that support logical variables, and furthermore in any language that has pointers (aliasing).

Tables 1 and 2 show the response times of analyzing and checking assertions in experiments **E1** and **E2**, respectively. For each of the files that were modified the table shows three columns: *noinc* is the response time in the nonincremental analysis setting, *inc* is the response time in the incremental setting, and *speedup* is the speedup of *inc* vs. *noinc*.

https://doi.org/10.1017/S1471068421000430 Published online by Cambridge University Press

```
1  :- module(_,[qsort/2],[assertions,nativeprops]).
2
3  :- entry qsort(X,Y) : (ground(X),list(X),var(Y)).
4
5  :- pred qsort(X,Y) => ground(Y).
6  qsort(X,Y) :- qsort_(X,Y,T), T=[].
7
8  :- pred qsort_(X,Y,Z) : (list(X),var(Y),var(Z),indep(Y,Z)) => ground(X).
9  qsort_([], Result, Result).
10 qsort_([First|Rest],ResultB,ResultE) :-
11     partition(Rest,First,Sm,Lg),
12     qsort_(Sm,SmB,SmE),
13     qsort_(Lg,LgB,LgE),
14     ResultB=SmB,
15     SmE=[First|LgB],
16     ResultE=LgE.
17
18 :- pred partition(L,P,Lg,Sm)
19    => (list(Lg), list(Sm), ground(Lg), ground(Sm)).
20 partition([],_,[],[]).
21 partition([X|Y],F,[X|Y1],Y2) :-
22     X @=< F,
23     partition(Y,F,Y1,Y2).
24 partition([X|Y],F,Y1,[X|Y2]) :-
25     X @> F,
26     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) )
```

Fig. 7: Sorting with incomplete data structures.

```
1  :- module(_,[p/1],[assertions,regtypes,functional,nativeprops]).
2
3  :- pred p(X) => sorted(X) + (is_det).
4
5  p(X) :-
6      q(X).
7
8  :- pred q(X) => color(X) + (is_det).
9
10 q(M) :-
11     M = red.
12
13 :- regtype color/1.
14 color := red | green | blue.
15
16 :- regtype sorted/1.
17 sorted := [] | [_].
18 sorted([X,Y|T]) :-  X>Y, sorted([Y|T]).
19
```

```
Verified assertion:
:- check calls p(X).
False assertion:
:- check success p(X)
   => sorted(X).
because the success field is incompatible with inferred succes
:
[eterms] rt0(X)
with:

:- regtype rt0/1.
rt0(red).
Verified assertion:
:- check comp p(X)
```

Fig. 8: A property incompatibility bug detected statically.

(a)

```
1  :- module(_,[top/0],[assertions]).
2
3  top:-
4      input_data(X),
5      compute(X,Y),
6      show_results(Y).
7
8  input_data(a).
9
10 compute(X,Y):-
11     X2 is 3,
12     X1 is X+X2,
13     Y is X1+1.
14
15 show_results(_X):-
16     % ...
17     true.
```

```
goal arithmetic:is(X1,X+X2) at literal 2 does not succeed
```

Static detection of illegal call to lib predicate.

(b)

```
2
3  top:-
4      input_data(X),
5      compute(X,Y),
6      show_results(Y).
7
8  input_data(5).
9
10 compute(X,Y):-
11     X1 is X-1,
12     compute(X1,Y1),
13     Y is Y1*X1.
14
15 show_results(_X):-
16     % ...
17     true.
```

```
goal always_fails:compute(X1,Y1) at literal 2 does not succeed
```

Static detection of simple non-termination.

Fig. 9: Detection of errors at program points.

https://doi.org/10.1017/S1471068421000430 Published online by Cambridge University Press

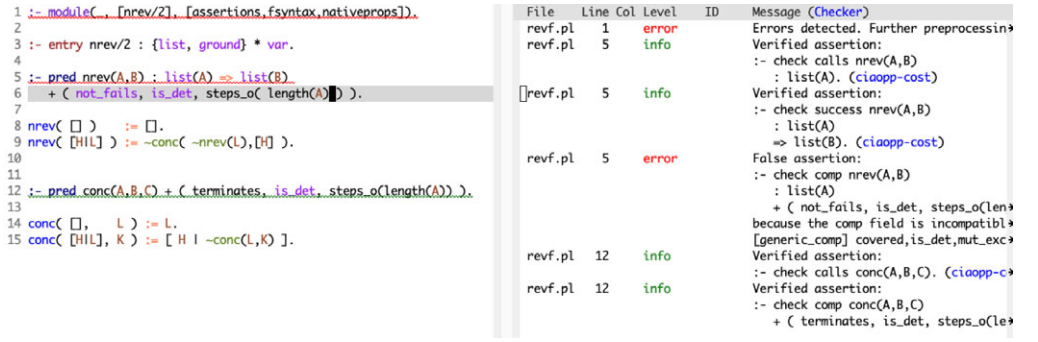


Fig. 10: Static verification of determinacy, termination, and cost (errors detected).

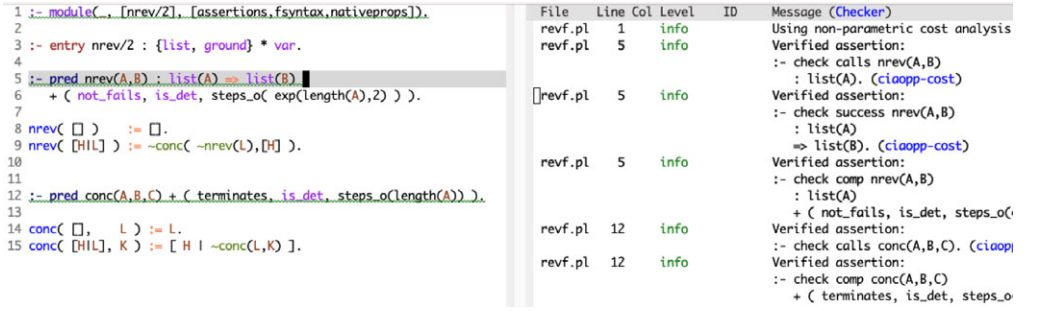


Fig. 11: Static verification of determinacy, termination, and cost (verified).

Each of the rows in the table corresponds to each of the abstract domains. The reported time is the average of *total* roundtrip assertion checking time, *measured from the IDE*, that is, what the programmer actually perceives.

In all of the experiments incrementality provides significant speedups. In the first experiment, **E1** (Table 1), for the `pairSh` and `def` domains, it allows keeping the response time, crucial for the interactive use case of the analyzer that we propose, under 2 s. For `ShGrC`, a significant speedup is also achieved (more than $\times 3.5$). The response time is borderline at 5 *s*. The reason for this is, as mentioned, that the domain is more precise, an thus more computationally expensive, which in fact allows (dis)proving more assertions. For the **E2** experiment (Table 2), the tool detects that no changes are required in the analysis results, and the assertions can be rechecked w.r.t. the available (previous) analysis. The performance analyzing with `ShGrC` is improved significantly (more than $\times 9$) but, more importantly, the response times are around 2 *s*. All in all, the incremental features allow using many domains and, at least in some cases, even the most expensive domains.

Note that the experiments reveal also that an interesting configuration of this tool is to run different analyses in a portfolio, where which analyses to run is decided depending on the kind of change occurred. If only assertions have changed, it is enough to recheck only with `ShGrC`. However if both the code and the assertions changed, analysis for all domains can be run in parallel giving fast, less precise feedback to the programmer as the results are available, and then refine the results when the more precise results are ready.

https://doi.org/10.1017/S1471068421000430 Published online by Cambridge University Press

Table 1: Average response time (seconds) for the experiments with any program edit.

	aggreg			readin			talkr		
domain	*noinc*	*inc*	*speedup*	*noinc*	*inc*	*speedup*	*noinc*	*inc*	*speedup*
pairSh	2.8	1.6	×1.8	2.9	1.5	×1.9	2.8	1.6	×1.8
def	3.0	1.6	×1.9	2.7	1.5	×1.8	2.9	1.7	×1.7
ShGrC	18.1	5.1	×**3.5**	18.3	5.1	×**3.6**	18.1	4.5	×**4.0**

Table 2: Average response time (seconds) for the experiments only changing assertions.

	aggreg			readin			talkr		
domain	*noinc*	*inc*	*speedup*	*noinc*	*inc*	*speedup*	*noinc*	*inc*	*speedup*
pairSh	2.8	1.7	×1.6	2.7	1.6	×1.7	2.9	1.7	×1.7
def	3.1	1.5	×2.0	2.9	1.4	×2.0	3.0	1.6	×1.9
ShGrC	18.2	2.0	×**9.1**	18.1	1.9	×**9.6**	18.2	1.9	×**9.6**

Aside from the data in the tables, we observed a constant overhead of 0.4 s for loading the code – parsing and prior transformations – in the tool. This is currently still not fully incremental and has not been optimized yet to load only the parts that change. Verification times are negligible w.r.t. analysis times and are approximately 0.1 s; this is also nonincremental, since we found that it is not currently a bottleneck, although we plan to make it incremental in the future to push the limits of optimizations forward.

5 Related work

The topic of assertion checking in logic programming, and in Prolog in particular, has received considerable attention. A family of approaches involves defining static-type systems for logic programs (Mycroft and O'Keefe 1984; Lakshman and Reddy 1991; Pfenning 1992), and several strongly typed logic programming systems have been proposed, notable examples being Mercury (Somogyi *et al.* 1996) and Gödel (Hill and Lloyd 1994). Approaches for combining strongly typed and untyped Prolog modules were proposed by Schrijvers *et al.* (2008). Most of these approaches impose a number of restrictions that make them less appropriate for dynamic languages like Prolog. The Ciao model introduced an alternative for writing safe programs without relying on full static typing, but based instead on the notions of safe approximations and abstract interpretation, providing a more general and flexible approach than in previous work, since assertions are optional and can contain any abstract property. This approach is specially useful for dynamic languages – see, for example, the paper by Hermenegildo *et al.* (2011) for a discussion of this topic. Some aspects of the Ciao model have been adopted or applied in other recent Prolog-based approaches, such as, for example, the papers by Schrijvers *et al.* (2008) and Wingen and Körner (2020), or the library for run-time checking of assertions in SWI-Prolog. The Ciao model can be considered an antecedent of the now popular *gradual-* and *hybrid-typing* approaches (Flanagan 2006; Siek and Taha 2006; Rastogi *et al.* 2015). Moreover, recent work by Stulova *et al.* (2018a; 2018b) illustrates further that these techniques and the exploitation of local data invariants allow maintaining

https://doi.org/10.1017/S1471068421000430 Published online by Cambridge University Press

strong safety guarantees with zero or very low (and potentially guaranteed (Klemen *et al.* 2018)) run-time checking overheads, even in a reusable-library context, without requiring switching to strong typing. This approach is also more natural in Prolog-style languages that distinguish between erroneous and failed (backtracking) executions. All these techniques taken together strongly suggest that using the Ciao approach comparable safety guarantees to those of strong typing can be achieved without their requirements. There has been work on incrementality within theorem proving-based verification approaches, such as, for example, the papers by Rustan *et al.* (2015) and Tschannen *et al.* (2015). Additional references can be found in the CiaoPP overview papers and the other citations provided.

6 Conclusions

We have shown how a combination 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 when reflecting analysis and verification results back as colorings and tooltips directly on the program text. We have termed this mode of operation "VeriFly mode." Our initial experience with this 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 behavior w.r.t. termination, resource usage, determinism, covering and failure, etc. While presented using the Emacs and the flycheck package, we argue that our techniques and results should be applicable to any VeriFly-style integration into a modern extensible IDE. We plan to continue our work to achieve further reactivity and scalability improvements, enhanced presentations of verification results, and improved diagnosis, contributing to further improve the programming environments available to the (C)LP programmer.

References

Bueno, F., Cabeza, D., Hermenegildo, M. V. and Puebla, G. 1996. Global analysis of standard prolog programs. In *ESOP*.

Bueno, F., Carro, M., Hermenegildo, M. V., Lopez-Garcia, P. and (Eds.), J. M. 2021. The Ciao System. Ref. Manual (v1.20). Tech. rep. April. URL: http://ciao-lang.org.

Bueno, F., Deransart, P., Drabent, W., Ferrand, G., Hermenegildo, M. V., Maluszynski, J. and Puebla, G. 1997. On the role of semantic approximations in validation and diagnosis of constraint logic programs. In *Proc. of the 3rd Int'l. WS on Automated Debugging–AADEBUG*. U. Linköping Press, 155–170.

Cousot, P. and Cousot, R. 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, 238–252.

De Angelis, E., Fioravanti, F., Gallagher, J. P., Hermenegildo, M. V., Pettorossi, A. and Proietti, M. 2021. Analysis and transformation of constrained Horn clauses for program verification. *Theory and Practice of Logic Programming*.

De Angelis, E., Fioravanti, F., Pettorossi, A. and Proietti, M. 2017. Semantics-based

https://doi.org/10.1017/S1471068421000430 Published online by Cambridge University Press

generation of verification conditions via program specialization. *Science of Computer Programming 147*, 78–108.

Dumortier, V., Janssens, G., Simoens, W. and García de la Banda, M. 1993. Combining a definiteness and a freeness abstraction for CLP languages. In *Workshop on LP Synthesis and Transformation*.

Flanagan, C. 2006. Hybrid type checking. In *33rd ACM POPL*, 245–256.

Gallagher, J., Hermenegildo, M. V., Kafle, B., Klemen, M., Lopez-Garcia, P. and Morales, J. 2020. From big-step to small-step semantics and back with interpreter specialization. In *VPT 2020*. EPTCS. Open Publishing Association, 50–65.

Garcia-Contreras, I., Morales, J. and Hermenegildo, M. V. 2020. Incremental analysis of logic programs with assertions and open predicates. In *LOPSTR'19*. LNCS, vol. 12042. Springer, 36–56.

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

Gómez-Zamalloa, M., Albert, E. and Puebla, G. 2009. Decompilation of java bytecode to Prolog by partial evaluation. *JIST 51*, 1409–1427.

Grebenshchikov, S., Gupta, A., Lopes, N. P., Popeea, C. and Rybalchenko, A. 2012. HSF(C): a software verifier based on Horn clauses. In *TACAS*, 549–551.

Gurfinkel, A., Kahsai, T., Komuravelli, A. and Navas, J. A. 2015. The SeaHorn verification framework. In *CAV*, 343–361.

Henriksen, K. S. and Gallagher, J. P. 2006. Abstract interpretation of PIC programs through logic programming. In *SCAM'06*. IEEE Computer Society, 184–196.

Hermenegildo, M., Puebla, G., Bueno, F. and Garcia, P. L. 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., Bueno, F., Carro, M., Lopez-Garcia, P., Mera, E., Morales, J. and Puebla, G. 2011. The Ciao approach to the dynamic vs. static language dilemma. In *Proc. Int'l. WS on Scripts to Programs, STOP'11*. ACM.

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. In *The Logic Programming Paradigm: A 25–Year Perspective*. Springer-Verlag, 161–192.

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

Hill, P. and Lloyd, J. 1994. *The Goedel Programming Language*. MIT Press.

Klemen, M., Stulova, N., Lopez-Garcia, P., Morales, J. F. and Hermenegildo, M. V. 2018. Static performance guarantees for programs with run-time checks. In *PPDP*. ACM.

Lakshman, T. and Reddy, U. 1991. Typed Prolog: a semantic reconstruction of the Mycroft-O'Keefe type system. In *International Logic Programming Symposium*. MIT Press.

Lopez-Garcia, P., Darmawan, L., Klemen, M., Liqat, U., Bueno, F. and Hermenegildo, M. V. 2018. Interval-based resource usage verification by translation into Horn clauses and an application to energy consumption. *TPLP 18,* 2, 167–223.

Marriott, K. and Søndergaard, H. 1993. Precise and efficient groundness analysis for logic programs. Technical report 93/7, University of Melbourne.

Méndez-Lojo, M., Navas, J. and Hermenegildo, M. 2007. A flexible (C)LP-based approach to the analysis of object-oriented programs. In *LOPSTR*. LNCS, vol. 4915. Springer-Verlag, 154–168.

https://doi.org/10.1017/S1471068421000430 Published online by Cambridge University Press

Mycroft, A. and O'Keefe, R. A. 1984. A polymorphic type system for Prolog. *Artificial Intelligence 23,* 3, 295–307.

Navas, J., Bueno, F. and Hermenegildo, M. V. 2006. Efficient top-down set-sharing analysis using cliques. In *8th PADL*. LNCS, vol. 2819. Springer, 183–198.

Peralta, J., Gallagher, J. and Sağlam, H. 1998. Analysis of imperative programs through analysis of constraint logic programs. In *Static Analysis. 5th International Symposium, SAS'98, Pisa*, G. Levi, Ed. LNCS, vol. 1503, 246–261.

Perez-Carrasco, V., Klemen, M., Lopez-Garcia, P., Morales, J. and Hermenegildo, M. V. 2020. Cost analysis of smart contracts via parametric resource analysis. In *Static Aanalysis Simposium (SAS'20)*. LNCS, vol. 12389. Springer, 7–31.

Pfenning, F., Ed. 1992. *Types in Logic Programming*. MIT Press.

Pietrzak, P., Correas, J., Puebla, G. and Hermenegildo, M. V. 2006. Context-sensitive multivariant assertion checking in modular programs. In *LPAR'06*. LNCS, vol. 4246. Springer-Verlag, 392–406.

Puebla, G., Bueno, F. and Hermenegildo, M. V. 2000a. An assertion language for constraint logic programs. In *Analysis and Visualization Tools for Constraint Programming*. LNCS, vol. 1870. Springer-Verlag, 23–61.

Puebla, G., Bueno, F. and Hermenegildo, M. V. 2000b. Combined static and dynamic assertion-based debugging of constraint logic programs. In *Proc. of LOPSTR'99*. LNCS, vol. 1817. Springer-Verlag, 273–292.

Puebla, G. and Hermenegildo, M. V. 1996. Optimized algorithms for the incremental analysis of logic programs. In *SAS'96*. LNCS, vol. 1145. Springer, 270–284.

Rastogi, A., Swamy, N., Fournet, C., Bierman, G. and Vekris, P. 2015. Safe & efficient gradual typing for TypeScript. In *42nd POPL*. ACM, 167–180.

Rustan, K., Leino, M. and Wüstholz, V. 2015. Fine-grained caching of verification results. In *CAV*, D. Kroening and C. S. Pasareanu, Eds. LNCS, vol. 9206. Springer, 380–397.

Schrijvers, T., Santos Costa, V., Wielemaker, J. and Demoen, B. 2008. Towards typed Prolog. In *ICLP'08*. LNCS, vol. 5366. Springer, 693–697.

Siek, J. G. and Taha, W. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 81–92.

Somogyi, Z., Henderson, F. and Conway, T. 1996. The execution algorithm of mercury: an efficient purely declarative logic programming language. *JLP 29,* 1–3, 17–64.

Stulova, N., Morales, J. F. and Hermenegildo, M. V. 2018a. Exploiting term hiding to reduce run-time checking overhead. In *Int'l. Symp. on Practical Aspects of Declarative Languages (PADL)*. LNCS, vol. 10702. Springer-Verlag, 99–115.

Stulova, N., Morales, J. F. and Hermenegildo, M. V. 2018b. Some trade-offs in reducing the overhead of assertion run-time checks via static analysis. *Science of Computer Programming 155*, 3–26. Highest ranked paper.

Tschannen, J., Furia, C. A., Nordio, M. and Meyer, B. 2011. Usable verification of object-oriented programs by combining static and dynamic techniques. In *SEFM*. LNCS, vol. 7041. Springer, 382–398.

Wingen, I. and Körner, P. 2020. Effectiveness of annotation-based static type inference. In *WS on Functional and Constraint Logic Programming*. LNCS, vol. 12560. Springer, 74–93.

https://doi.org/10.1017/S1471068421000430 Published online by Cambridge University Press