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.
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
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 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

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
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:

\[ \texttt{:- [ Status ] pred Head [: Pre ] [=> Post ] [+ Comp ]}. \]

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 Statuses (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).
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 (props, 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, props 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: :- `regtype list/2`. or, more compactly, :- `regtype list/2`.

The combination of multiple assertions \( \{a_1, \ldots, a_n\} \) for a given predicate, where \( a_i = \text{:-} \text{pred Head : Pre}_i \Rightarrow \text{Post}_i \) is defined more precisely as follows: the set of assertion conditions for `Head` is \( \{C_0, C_1, \ldots, C_n\} \), with:

\[
C_0 = \text{calls}(\text{Head}, \bigvee_{j=1}^n \text{Pre}_j), \quad \text{and} \\
C_i = \text{success}(\text{Head}, \text{Pre}_i, \text{Post}_i), \quad i = 1, \ldots, n
\]
where \( \text{calls}(\text{Head}, \text{Pre}) \) states conditions on all concrete calls to the predicate described by \( \text{Head} \), and \( \text{success}(\text{Head}, \text{Pre}_i, \text{Post}_i) \) the conditions on the success constraints produced by calls to \( \text{Head} \) if \( \text{Pre}_i \) is satisfied. If the assertions \( a_i \) above, \( i = 1, \ldots, n \), include a \( + \text{Comp} \) field, then the set of \emph{assertion conditions} also includes conditions of the form: \( \text{comp}(\text{Head}, \text{Pre}_i, \text{Comp}_i) \), for \( i = 1, \ldots, n \), that express properties of the whole computation for calls to \( \text{Head} \) if \( \text{Pre}_i \) is satisfied.

In addition to predicate-level assertions, PROGRAM-POINT ASSERTIONS are of the form:
\[
\text{Status}(\text{StateFormula}),
\]
and they can be placed at program locations where a new literal may be added. For example, for status \text{check} they should be interpreted as “whenever computation reaches a state corresponding to this program point, \text{StateFormula} should hold.” For example,
\[
\text{check}((\text{list}(\text{color}, A), \text{var}(B))),
\]
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.

\textit{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 \textit{modes}, Cartesian product notation, markdown syntax, etc. For example, the following set of \textit{pred} assertions use Cartesian product notation to provide information on a reversible sorting predicate:
\[
\begin{align*}
\text{:- pred sort/2 : list(num) * var} & \Rightarrow \text{list(num)} \quad \text{* list(num) + is_det.} \\
\text{:- pred sort/2 : var * list(num)} & \Rightarrow \text{list(num)} \quad \text{* list(num) + non_det.}
\end{align*}
\]
(in addition, curly brackets can be used to group properties – see Figure 2). The assertion language also allows defining \textit{modes}, which are macros that can be placed in argument positions of \( \text{Head} \) and expand to properties in different assertion fields. For example, using the \textit{isomodes} library the assertions above can also be expressed as:
\[
\begin{align*}
\text{:- pred sort(+list(num), -list(num)) + is_det.} \\
\text{:- pred sort(-list(num), +list(num)) + non_det.}
\end{align*}
\]
Or, if no types and only modes are used:
\[
\begin{align*}
\text{:- pred sort(+, -).} \\
\text{:- pred sort(-, +).}
\end{align*}
\]
There is also an alternative encoding via “markdown-style” comments (provided by libraries \textit{doccomments} and \textit{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:
\[
\begin{align*}
\text{:- mode sort(+, -).} \\
\text{:- mode sort(-, +).}
\end{align*}
\]
or the SWI-Prolog \texttt{pldoc}-style annotations for documentation, making all of them machine checkable.

\textit{Static program analysis.} In order to compute safe over-approximations of the program semantics, the \texttt{CiaoPP} verification framework uses analyses based on abstract interpretation (the “Static Analyzer” in Figure 1). Given a program \( P \) and a set of queries \( Q \), an
analysis graph is inferred that abstracts the (possibly infinite) set of (possibly infinite) execution trees. The analysis result is denoted by $[P]_Q^\alpha$, where $\alpha$ 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 $[P]_Q^\alpha$ as a set of tuples (one per node): 
\[ \{ (L_1, \lambda_c^1, \lambda_s^1), \ldots, (L_n, \lambda_c^n, \lambda_s^n) \} \].

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 $[P]_Q^\alpha$ 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 \( \text{Ren}(p(\bar{v}), [P]_Q^\alpha) \), 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. $Q$ renamed to variables $\bar{v}$, that is: 
\[ \text{Ren}(p(\bar{v}), [P]_Q^\alpha) = \{ (p(\bar{v}), \lambda_c, \lambda_s) | \exists (p(\bar{v}'), \lambda_c', \lambda_s') \in [P]_Q^\alpha \text{ and } \sigma \text{ is a renaming substitution s.t. } p(\bar{v}) = p(\bar{v}')\sigma \}. \]

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

- **calls**($p(\bar{v}), Pre$) is checked (resp. false) for predicate $p \in P$ w.r.t. $Q$ if $
exists (\forall p(\bar{v}), \lambda_c, \lambda_s) \in Ren(p(\bar{v}), [P]^\alpha_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. $Q$ if $
exists (\forall p(\bar{v}), \lambda_c, \lambda_s) \in Ren(p(\bar{v}), [P]^\alpha_Q): ((\lambda_c \sqcap Pre^\alpha + = \bot) \lor (\lambda_s \sqsubseteq Post^\alpha -))$ (resp. $\lambda_c \sqsubseteq Pre^\alpha - \land (\lambda_s \sqcap Post^\alpha + = \bot)$ 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 ($\Delta$ CCH 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 (García-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. 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
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 an active module, a daemon process that executes in the background and reacts to simple JSON-encoded queries via a socket. 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-
VeriFly: On-the-fly Assertion Checking via Incrementality

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).
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 \texttt{sorted/1} is approximated by the regtypes domain as \texttt{list/1} and an incompatibility is found with \texttt{rt0}, i.e., \texttt{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 \texttt{nrev} should be linear in the length of the (input) argument \texttt{A}. This is expressed by the property \texttt{steps0(length(A))}, meaning that the cost, in terms of resolution steps, of any call to \texttt{nrev(A, B)} with \texttt{A} bound to a list and \texttt{B} a free variable, is in \texttt{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 ($\frac{1}{2} \text{length}(A)^2 + \frac{3}{2} \text{length}(A) + 1$) inferred by the static analyzer. In contrast, assertion in lines 5 and 6 of Figure 11 (left) states that predicate \texttt{nrev} should have a quadratic worst case asymptotic complexity in the length of \texttt{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 \texttt{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 (\texttt{E1}), and only assertions (\texttt{E2}). The edits were performed on three selected files: \texttt{aggreg}, \texttt{readin}, and \texttt{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) (\texttt{pairSh}), a dependency tracking via propositional clauses domain (Dumortier et al. 1993) (\texttt{def}), and sharing+groundness with clique-based widening (Navas et al. 2006) (\texttt{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 \texttt{E1} and \texttt{E2}, respectively. For each of the files that were modified the table shows three columns: \texttt{noinc} is the response time in the nonincremental analysis setting, \texttt{inc} is the response time in the incremental setting, and \texttt{speedup} is the speedup of \texttt{inc} vs. \texttt{noinc}.
VeriFly: On-the-fly Assertion Checking via Incrementality

Fig. 7: Sorting with incomplete data structures.

Fig. 8: A property incompatibility bug detected statically.

Fig. 9: Detection of errors at program points.
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, $E_1$ (Table 1), for the $\text{pairSh}$ and $\text{def}$ domains, it allows keeping the response time, crucial for the interactive use case of the analyzer that we propose, under 2 s. For $\text{ShGrC}$, 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 $E_2$ 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 $\text{ShGrC}$ 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 $\text{ShGrC}$. 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.
Table 1: Average response time (seconds) for the experiments with any program edit.

<table>
<thead>
<tr>
<th>domain</th>
<th>aggreg</th>
<th></th>
<th>readin</th>
<th></th>
<th>talkr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>noinc</td>
<td>inc</td>
<td>speedup</td>
<td>noinc</td>
<td>inc</td>
</tr>
<tr>
<td>pairSh</td>
<td>2.8</td>
<td>1.6</td>
<td>×1.8</td>
<td>2.9</td>
<td>1.5</td>
</tr>
<tr>
<td>def</td>
<td>3.0</td>
<td>1.6</td>
<td>×1.9</td>
<td>2.7</td>
<td>1.5</td>
</tr>
<tr>
<td>ShGrC</td>
<td>18.1</td>
<td>5.1</td>
<td>×3.5</td>
<td>18.3</td>
<td>5.1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>domain</th>
<th>aggreg</th>
<th></th>
<th>readin</th>
<th></th>
<th>talkr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>noinc</td>
<td>inc</td>
<td>speedup</td>
<td>noinc</td>
<td>inc</td>
</tr>
<tr>
<td>pairSh</td>
<td>2.8</td>
<td>1.7</td>
<td>×1.6</td>
<td>2.7</td>
<td>1.6</td>
</tr>
<tr>
<td>def</td>
<td>3.1</td>
<td>1.5</td>
<td>×2.0</td>
<td>2.9</td>
<td>1.4</td>
</tr>
<tr>
<td>ShGrC</td>
<td>18.2</td>
<td>2.0</td>
<td>×9.1</td>
<td>18.1</td>
<td>1.9</td>
</tr>
</tbody>
</table>

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
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


