Abstract Verification and Debugging of Constraint Logic Programs

Manuel Hermenegildo, *Germán Puebla*,
Francisco Bueno, and *Pedro López-García*

{herme,german,bueno,pedro}@fi.upm.es
Department of Computer Science
Technical University of Madrid (UPM)

(Extended Abstract)

Keywords: Global Analysis, Debugging, Verification, Constraint Logic Programming, Optimization, Parallelization, Abstract Interpretation.

1 Background

The technique of Abstract Interpretation [13] has allowed the development of sophisticated program analyses which are provably correct and practical. The semantic approximations produced by such analyses have been traditionally applied to *optimization* during program compilation. However, recently, novel and promising applications of semantic approximations have been proposed in the more general context of program *verification* and *debugging* [3, 10, 7].

In the case of Constraint Logic Programs (CLP), a comparatively large body of approximation domains, inference techniques, and tools for abstract interpretation-based semantic analysis have been developed to a powerful and mature level (see, e.g., [28, 9, 21, 6, 22, 24] and their references). These systems can approximate at compile-time a wide range of properties, from directional types to variable independence, determinacy or termination, always safely, and with a significant degree of precision.

Our proposed approach takes advantage, within the context of program verification and debugging, of these significant advances in static program analysis techniques and the resulting concrete tools, which have been shown useful for other purposes such as optimization, and are thus likely to be present in compilers. This is in contrast to using traditional proof-based methods (e.g., for the case of CLP, [1, 2, 15, 19, 34]), developing new tools and procedures (such as specific concrete [4, 17, 18] or abstract [10, 11] diagnosers and declarative debuggers), or limiting error detection to run-time checking (e.g., [34]).

2 An Approach Based on Semantic Approximations

We now briefly describe the basis of our approach [7, 25, 31]. We consider the important class of semantics referred to as *fixpoint semantics*. In this setting, a

Property	Definition
P is partially correct w.r.t. $\mathcal{I}$	$[\![P]\!] \subseteq \mathcal{I}$
P is complete w.r.t. $\mathcal{I}$	$\mathcal{I} \subseteq [\![P]\!]$
P is incorrect w.r.t. $\mathcal{I}$	$[\![P]\!] \not\subseteq \mathcal{I}$
P is incomplete w.r.t. $\mathcal{I}$	$\mathcal{I} \not\subseteq [\![P]\!]$

Table 1. Set theoretic formulation of verification problems

(monotonic) semantic operator (which we refer to as S_P) is associated with each program P. This S_P function operates on a semantic domain which is generally assumed to be a complete lattice or, more generally, a chain complete partial order. The meaning of the program (which we refer to as $[\![P]\!]$) is defined as the least fixpoint of the S_P operator, i.e., $[\![P]\!] = \mathrm{lfp}(S_P)$. A well-known result is that if S_P is continuous, the least fixpoint is the limit of an iterative process involving at most ω applications of S_P and starting from the bottom element of the lattice.

Both program verification and debugging compare the *actual semantics* of the program, i.e., $[\![P]\!]$, with an *intended semantics* for the same program, which we denote by $\mathcal{I}$. This intended semantics embodies the user's requirements, i.e., it is an expression of the user's expectations. In Table 1 we define classical verification problems in a set-theoretic formulation as simple relations between $[\![P]\!]$ and $\mathcal{I}$.

Using the exact actual or intended semantics for automatic verification and debugging is in general not realistic, since the exact semantics can be only partially known, infinite, too expensive to compute, etc. An alternative and interesting approach is to approximate the semantics. This is interesting, among other reasons, because a well understood technique already exists, abstract interpretation, which provides *safe* approximations of the program semantics. Our first objective is to present the implications of the use of *approximations* of both the intended and actual semantics in the verification and debugging process.

2.1 Approximating Program Semantics

We start by recalling some basic concepts from abstract interpretation. In this technique, a program is interpreted over a non-standard domain called the *abstract* domain D_α which is simpler than the *concrete* domain D, and the semantics w.r.t. this abstract domain, i.e., the *abstract semantics* of the program is computed (or approximated) by replacing the operators in the program by their abstract counterparts.

The concrete and abstract domains are related via a pair of monotonic mappings: *abstraction* $\alpha : D \mapsto D_\alpha$, and *concretization* $\gamma : D_\alpha \mapsto D$, which relate the two domains by a Galois insertion (or a Galois connection) [13]. We will denote by $[\![P]\!]_\alpha$ the result of abstract interpretation for a program P. Typically, abstract interpretation guarantees that $[\![P]\!]_\alpha$ is an over-approximation of the abstract semantics of the program itself, $\alpha(\llbracket P \rrbracket)$. Thus, we have that $\llbracket P \rrbracket_\alpha \supseteq \alpha(\llbracket P \rrbracket)$, which we will denote as $\llbracket P \rrbracket_{\alpha^+}$. Alternatively, the analysis can be designed to safely under-approximate the actual semantics, and then we have that $\llbracket P \rrbracket_\alpha \subseteq \alpha(\llbracket P \rrbracket)$, which we denote as $\llbracket P \rrbracket_{\alpha^-}$.

2.2 Abstract Verification and Debugging

The key idea in our approach is to use the abstract approximation $\llbracket P \rrbracket_\alpha$ directly in verification and debugging tasks. As we will see, the possible loss of accuracy due to approximation prevents full verification in general. However, and interestingly, it turns out that in many cases useful verification and debugging conclusions can still be derived by comparing the approximations of the actual semantics of a program to the (also possibly approximated) intended semantics.

A number of approaches have already been proposed which make use to some extent of abstract interpretation in verification and/or debugging tasks. Abstractions were used in the context of algorithmic debugging in [27]. Abstract interpretation for debugging of imperative programs has been studied by Bourdoncle [3], and for the particular case of algorithmic debugging of logic programs by Comini et al. [12] (making use of partial specifications) and [10].

In our approach we actually compute the abstract approximation $\llbracket P \rrbracket_\alpha$ of the actual semantics of the program $\llbracket P \rrbracket$ and compare it directly to the (also approximate) intention (which is given in terms of *assertions* [30]), following almost directly the scheme of Table 1. This approach can be very attractive in programming systems where the compiler already performs such program analysis in order to use the resulting information to, e.g., optimize the generated code. I.e., in these cases the compiler will compute $\llbracket P \rrbracket_\alpha$ anyway.

For now, we assume that the program specification is given as a semantic value $\mathcal{I}_\alpha \in D_\alpha$. Comparison between actual and intended semantics of the program is most easily done in the same domain, since then the operators on the abstract lattice, that are typically already defined in the analyzer, can be used to perform this comparison. Thus, for comparison we need in principle $\alpha(\llbracket P \rrbracket)$. Using abstract interpretation, we can usually only compute instead $\llbracket P \rrbracket_\alpha$, which is an approximation of $\alpha(\llbracket P \rrbracket)$. Thus, it is interesting to study the implications of comparing $\mathcal{I}_\alpha$ and $\llbracket P \rrbracket_\alpha$.

In Table 2 we propose (sufficient) conditions for correctness and completeness w.r.t. $\mathcal{I}_\alpha$, which can be used when $\llbracket P \rrbracket$ is approximated. Several instrumental conclusions can be drawn from these relations.

Analyses which over-approximate the actual semantics (i.e., those denoted as $\llbracket P \rrbracket_{\alpha^+}$), are specially suited for proving partial correctness and incompleteness with respect to the abstract specification $\mathcal{I}_\alpha$. It will also be sometimes possible to prove incorrectness in the extreme case in which the semantics inferred for the program is incompatible with the abstract specification, i.e., when $\llbracket P \rrbracket_{\alpha^+} \cap \mathcal{I}_\alpha = \emptyset$. We also note that it will only be possible to prove completeness if the abstraction is *precise*, i.e., $\llbracket P \rrbracket_\alpha = \alpha(\llbracket P \rrbracket)$. According to Table 2 only $\llbracket P \rrbracket_{\alpha^-}$ can be used to this end, and in the case we are discussing $\llbracket P \rrbracket_{\alpha^+}$ holds. Thus, the only possibility is that the abstraction is precise.

Property	Definition	Sufficient condition
P is partially correct w.r.t. $\mathcal{I}_\alpha$	$\alpha([\![P]\!]) \subseteq \mathcal{I}_\alpha$	$[\![P]\!]_{\alpha^+} \subseteq \mathcal{I}_\alpha$
P is complete w.r.t. $\mathcal{I}_\alpha$	$\mathcal{I}_\alpha \subseteq \alpha([\![P]\!])$	$\mathcal{I}_\alpha \subseteq [\![P]\!]_{\alpha^-}$
P is incorrect w.r.t. $\mathcal{I}_\alpha$	$\alpha([\![P]\!]) \not\subseteq \mathcal{I}_\alpha$	$[\![P]\!]_{\alpha^-} \not\subseteq \mathcal{I}_\alpha$, or $[\![P]\!]_{\alpha^+} \cap \mathcal{I}_\alpha = \emptyset \wedge [\![P]\!]_\alpha \neq \emptyset$
P is incomplete w.r.t. $\mathcal{I}_\alpha$	$\mathcal{I}_\alpha \not\subseteq \alpha([\![P]\!])$	$\mathcal{I}_\alpha \not\subseteq [\![P]\!]_{\alpha^+}$

Table 2. Validation problems using approximations

On the other hand, if analysis under-approximates the actual semantics, i.e., in the case denoted $[\![P]\!]_{\alpha^-}$, it will be possible to prove completeness and incorrectness. In this case, partial correctness and incompleteness can only be proved if the analysis is precise.

If analysis information allows us to conclude that the program is incorrect or incomplete w.r.t. $\mathcal{I}_\alpha$, an (abstract) symptom has been found which ensures that the program does not satisfy the requirement. Thus, debugging should be initiated to locate the program construct responsible for the symptom.

More details about the theoretical foundation of our approach can be found in [7, 31].

3 A Practical Framework and its Implementation

Using the ideas outlined above, we have developed a framework [25, 29] capable of combined static and dynamic validation, and debugging for CLP programs, using semantic approximations, and which can be integrated in an advanced program development environment comprising a variety of co-existing tools [16].

This framework has been implemented as a generic preprocessor composed of several tools. Figure 1 depicts the overall architecture of the system. Hexagons represent the different tools involved and arrows indicate the communication paths among the different tools.

Program verification and detection of errors is first performed at compile-time by using the sufficient conditions shown in Table 2. I.e., by inferring properties of the program via abstract interpretation-based static analysis and comparing this information against (partial) specifications written in terms of assertions. Such assertions are linguistic constructions which allow expressing properties of programs.

Classical examples of assertions are type declarations (e.g., in the context of (C)LP those used by [26, 32, 5]). However, herein we are interested in supporting a much more powerful setting in which assertions can be of a much more general nature, stating additionally other properties, some of which cannot always be determined statically for all programs. These properties may include properties defined by means of user programs and extend beyond the predefined set which may be natively understandable by the available static analyzers. Also, in the proposed framework only a small number of (even zero) assertions may be present

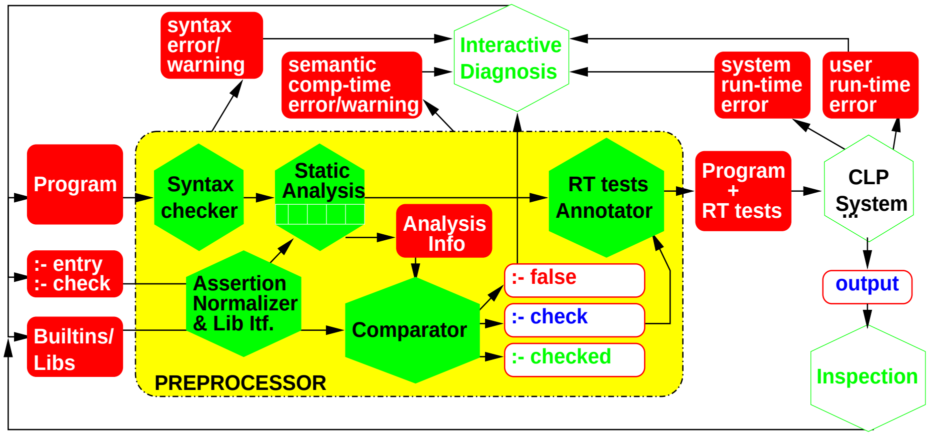

Fig. 1. Architecture of the Preprocessor

in the program, i.e., the assertions are *optional*. In general, we do not wish to limit the programming language or the language of assertions unnecessarily in order to make the validity of the assertions statically decidable (and, consequently, the proposed framework needs to deal throughout with *approximations*). We also propose a concrete language of assertions which allows writing this kind of (partial) specifications for CLP [30].

The assertion language is also used by the preprocessor to express both the information inferred by the analysis and the results of the comparisons performed against the specifications.[1] As can be derived from Table 2, these comparisons can result in proving statically (i.e., at compile-time) that the assertions hold (i.e., they are validated) or that they are violated, and thus bugs have been detected. User-provided assertions (or *parts* of assertions) which cannot be statically proved nor disproved are optionally translated into run-time tests. Both the static and the dynamic checking are provably *safe* in the sense that all errors flagged are definite violations of the specifications.

The practical usefulness of the framework is illustrated by what is arguably the first and most complete implementation of these ideas: CiaoPP,[2] the Ciao system preprocessor [29, 24]. Ciao is a public-domain, next-generation constraint logic programming system, which supports ISO-Prolog, but also, selectively for each module, extensions and restrictions such as, for example, pure logic programming, constraints, functions, objects, or higher-order. Ciao is specifically designed to a) be highly extensible and b) support modular program analysis, debugging, and optimization. The latter tasks are performed in an integrated fashion by CiaoPP.

[1] Interestingly, the assertions are also quite useful for generating documentation automatically (see [23]).

[2] A demonstration of the system was performed at the meeting.

CiaoPP, which incorporates analyses developed by several groups in the LP and CLP communities, uses abstract interpretation to infer properties of program predicates and literals, including types, modes and other variable instantiation properties, constraint independence, non-failure, determinacy, bounds on computational cost, bounds on sizes of terms in the program, etc. It processes modules separately, performing incremental analysis. CiaoPP can find errors at compile-time (or perform partial verification) by checking how programs call system libraries. This is possible since the expected behaviour of system predicates is also given in terms of assertions This allows detecting errors in user programs even if they contain no assertions. Also, the preprocessor can detect errors as well by checking the assertions present in the program or in other modules used by the program. As already mentioned, assertions are completely optional. Nevertheless, if the program is not correct, the more assertions are present in the program the more likely it is for errors to be automatically detected. Thus, for those parts of the program which are potentially buggy or for parts whose correctness is crucial, the programmer may decide to invest more time in writing assertions than for other parts of the program which are more stable. In addition, CiaoPP also performs program transformations and optimizations such as multiple abstract specialization, parallelization (including granularity control), and inclusion of run-time tests for assertions which cannot be checked completely at compile-time.

Finally, the implementation of the preprocessor is generic in that it can be easily customized to different CLP systems and dialects and in that it is designed to allow the integration of additional analyses in a simple way. As a particularly interesting example, the preprocessor has been adapted for use with the CHIP CLP(FD) system. This has resulted in CHIPRE, a preprocessor for CHIP which has been shown to detect non-trivial programming errors in CHIP programs. In the next section we show an example of a debugging session with CHIPRE. More information on the system can be found in [29].

4 A Sample Debugging Session with CHIPRE

In this section we will show some of the capabilities of our debugging framework through a sample session with CHIPRE, an implemented instance of the framework. Consider Figure 2, which contains a tentative version of a CHIP program for solving the *ship* scheduling problem, a typical CLP(FD) benchmark.

Often, the results of static analysis are good indicators of bugs, even if no assertion is given. This is because "strange" results often correspond to bugs. An important observation is that plenty of static analyses, such as modes and regular types, compute over-approximations of the success sets of predicates. Then, if such an over-approximation corresponds to the empty set then this implies that such predicate never succeeds. Thus, unless the predicate is dead-code, this often indicates that the code for the predicate is erroneous since every call either fails finitely (or raises an error) or loops. If analysis is goal-dependent and thus also computes an over-approximation of the calling states to the predicate, predicates

```
solve(Upper,Last,N,Dis,Mis,L,Sis):-
        length(Sis, N),
        Sis :: 0..Last,
        Limit :: 0..Upper,
        End :: 0..Last,
        set_precedences(L, Sis, Dis),
        cumulative(Sis, Dis, Mis, unused, unsed, Limit, End, unused),
        min_max(labeling(Sis), End).

labeling([]).
labeling([H|T]):-
        delete(X,[H|T],R,0,most_constrained),
        indomain(X),
        labeling(R).

set_precedences(L, Sis, Dis):-
        Array_starts=..[starts|Sis], % starts(S1,S2,S3,...)
        Array_durations=..[durations|Dis], % durations(D1,D2,D3,...)
        initialize_prec(L,Array_starts),
        set_pre_lp(l, array_starts, Array_durations).

set_pre_lp([], _, _).
set_pre_lp([After#>=Before|R], Array_starts, Array_durations):-
        arg(After, Array_starts, S2),
        arg(Before, Array_starts, S1),
        arg(Before, Array_durations, D1),
        S2 #>= S1 + D1,
        set_pre_lp(R, Array_starts, array_durations).

initialize_prec(_,_).
```

Fig. 2. A tentative *ship* program in CHIP

which are dead-code can often be identified by having an over-approximation of the calling states which corresponds to the empty set.

We now preprocess the current version of our example program using *regular type* [35, 14, 21, 20, 33] analysis. Our implementation of regular types is goal-dependent and thus computes over-approximations of both the success set and calling states of all predicates. In addition, our analysis also computes over-approximations of the values of variables at each program point. Once analysis information is available, the preprocessor automatically checks the consistency of the analysis results and we get the following messages:

```
WARNING: Literal set_precedences(L, Sis, Dis)
           at solve/7/1/5 does not succeed!
WARNING: Literal set_pre_lp(l, array_starts, Array_duration)
           at set_precedences/3/1/4 does not succeed!
```

The first warning message refers to a literal (in particular, the 5th literal in the 1st clause of `solve/7`) which calls the predicate `set_precedences/3`, whose success type is empty. Also, even if the success type of a predicate is not empty, i.e., there may be some calls which succeed, it may be possible to detect that at a certain program point the given call to the predicate cannot succeed because the type of the particular call is incompatible with the success type of the predicate. This is the reason for the second warning message. Note that this kind of reasoning can only be made if (1) the static analysis used infers properties which are *downwards closed*, i.e., once they hold they keep on being valid during forward execution and (2) analysis computes descriptions at each program point which, as already mentioned, is the case with our regular type analysis. Note that the predicate `set_pre_lp/3` can only succeed if the value at the first argument is compatible with a list. However, the call `set_pre_lp(l, array_starts, Array_duration)` has the constant `l` at the first argument position. This is actually a bug, as the constant `l` should instead be the variable `L`. Once we correct this bug, in subsequent preprocessing of the program both warning messages disappear. In fact, the first one was also a consequence of the same bug which propagated to the calling predicates of `set_precedences/3`.

4.1 Aiding the Analyzer

In the ship program, all initial queries to the program are intended to be to the `solve` predicate. However, the compiler has no way to automatically determine this. Thus, in the absence of more information, the most general possible calls have to be assumed for *all* predicates in the program.[3] One way to alleviate this is to provide *entry* assertion(s) which are assumed to cover all possible initial calls to the program. Even the simplest entry declaration which can be given for predicate `solve`, i.e., '`:- entry solve/7.`', is very useful for goal-dependent static analysis. Since it is the only `entry` assertion, the only calls to the rest of the predicates in the program are those generated during computations of `solve/7`. This allows analysis to start from the predicate `solve/7` only, instead of from all predicates. Reducing the number (and generality) of starting points for goal-dependent analysis by means of `entry` declarations often leads to increased precision and reduced analysis times. However, analysis will still make no assumptions regarding the arguments of the calls to `solve/7` since there is no further information available. This could be improved using a more accurate entry declaration such as the following:

```
:- entry solve/7 : int * int * int * list(int) * list(int) * list * term.
```

It gives the types of the seven arguments, and describes more precisely the valid input data. Note that the assertion above also specifies a *mode* for the calling patterns. The first three arguments are *required* to be instantiated to integers.

[3] Note that this can be partly alleviated with a strict module system such as that of Ciao [8], in which only exported predicates of a module can be subject to initial queries.

The forth and fifth must be fully instantiated to lists of integers. The sixth argument is (only) required to be instantiated to a list skeleton. Finally, the seventh argument can be any possible term. Note that, by default, our assertion language interprets properties in assertions as *instantiation* properties. However, the assertion language also allows the use of *compatibility* properties if so desired [30].

4.2 Assertions for System Predicates

Consider a new version of the *ship* program, after correcting the typo involving `L` and introducing the (simple) entry declaration '`:- entry solve/7.`'. When preprocessing the program the following messages are issued:

```
ERROR: Builtin predicate
        cumulative(Sis,Dis,Mis,unused,unsed,Limit,End,unused)
        at solve/7/1/6 is not called as expected (argument 5):
         Called:    ^unsed
         Expected:  intlist_or_unused

ERROR: Builtin predicate arg(After,Array_starts,S2)
        at set_pre_lp/3/2/1 is not called as expected (argument 2):
         Called:    ^array_starts
         Expected:  struct
```

Which indicate that the program is still definitely incorrect. Note that the preprocessor could not detect this without the extra precision allowed by the `entry` assertion. In error messages involving regular types, one important issue is not to confuse term constructors with type constructors. In order to improve the readability and conciseness of the error messages, the marker `^` is used to distinguish terms (constants) from regular types (which represent regular sets of terms). By default, values represent regular types. However, if they are marked with `^` they represent constants. In our example, `intlist_or_unused` is a type since it is not marked with `^` whereas `^unsed` is a constant. Note that though it is always possible to define a regular type which contains a single constant such as `unsed` and distinguish terms from types by the context in which the value appears, we opt by introducing the marker `^` ("quote") since in our experience this improves readability of error messages. Note that defining such type explicitly instead would require inventing a new name for it and providing the definition of the type together with the error message.

Coming back to the pending error messages, the first message is due to the fact that the constant `unused` has been mistakingly typed as `unsed` in the fifth argument of the call to the CHIP builtin predicate `cumulative/8`. As indicated in the error message, this predicate requires the fifth argument to be of type `intlist_or_unused` which was defined when writing assertions for the system predicates in CHIP and which indicates that such argument must be either the constant `unused` or a list of integers.

The automatic detection of this error at compile-time has been possible because the CHIP builtins have been provided with assertions that describe their intended use. Though system predicates are in principle considered correct under the assumption that they are called with valid input data, it is often useful to check that they are indeed called with valid input data. In fact, existing CLP systems perform this checking at run-time. The existence of such assertions allows checking the calls to system predicates at compile-time in addition to run-time in CLP systems which originally do not perform compile-time checking.

In the second message we have detected that we call the CHIP builtin predicate `arg/3` with the second argument bound to `array_starts` which is a constant (as indicated by the marker `^`) and thus of arity zero. This is incompatible with the expected call type `struct`, i.e., a structure with arity strictly greater than zero. In the current version of CHIP, this will generate a run-time error, whereas in other systems such as Ciao and SICStus, this call would fail but would not raise an error. Though we know the program is incorrect, the literal where the error is flagged, `arg(After, Array_starts, S2)` is apparently correct. We correct the first error and leave detection of the cause for the second error for later.

The different behaviour of seemingly identical builtin predicates (such as `arg/3` in the example above) in different systems further emphasizes the benefits of describing builtin predicates by means of assertions. They can be used for easily customizing static analysis for different systems, as assertions are easier to understand by naive users of the analysis than the hardwired internal representation used in ad-hoc analyzers for a particular system.

4.3 Assertions for User-Defined Predicates

Up to now we have seen that the preprocessor is capable of issuing a good number of error and warning messages even if the user does not provide any `check` assertions (assertions that the system should check to hold). We believe that this is very important in practice. However, adding assertions to programs can be useful for several reasons. One is that they allow further semantic checking of the programs, since the assertions provided encode a partial specification of the user's intention, against which the analysis results can be compared. Another one is that they also allow a form of diagnosis of the error symptoms detected, so that in some cases it is possible to automatically locate the program construct responsible for the error.

Consider again the pending error message from the previous iteration over the ship program. We know that the program is incorrect because (global) type analysis tells us that the variable `Array_starts` will be bound at run-time to the constant `array_starts`. However, by just looking at the definition of predicate `set_pre_lp` it is not clear where this constant comes from. This is because the cause of this problem is not in the definition of `set_pre_lp` but rather in that the predicate is being used incorrectly (i.e., its precondition is violated). We thus introduce the following `calls` assertion, which describes the expected calls to the predicate:

```
:- calls set_pre_lp(A,B,C): (struct(B),struct(C)).
```

In this assertion we require that both the second and third parameters of the predicate, i.e., B and C are structures with arity greater than zero, since in the program we are going to access the arguments in the structure of B and C with the builtin predicate arg/3.

The next time our ship program is preprocessed, having added the calls assertion, besides the pending error message of above regarding arg/3, we also get the following one:

```
ERROR: false assertion at set_precedences/3/1/4
       unexpected call (argument 2):
         Called:   ^array_starts
         Expected: struct
```

This message tells us the exact location of the bug, the fourth literal of the first clause for predicate set_precedences/3. This is because we have typed the constant array_starts instead of the variable Array_starts in such literal.

Thus, as shown in the example above, user-provided check assertions may help in locating the actual cause for an error. Also, as already mentioned, and maybe more obvious, user-provided assertions may allow detecting errors which are not easy to detect automatically otherwise.

After correcting the bug located in the previous example, preprocessing the program once again produces the following error message:

```
ERROR: false assertion at set_pre_lp/3/2/5
       unexpected call (argument 3):
         Called:   ^array_durations
         Expected: struct
```

which would not be automatically detected by the preprocessor without user-provided assertions. The obvious correction is to replace array_durations in the recursive call to set_pre_lp in its second clause with Array_durations.

After correcting this bug, preprocessing the program with the given assertions does not generate any more messages. Besides, the user provided calls assertion would have been proved by analysis.

Additionally, if some part of an assertion for a user-defined predicate has not been proved nor disproved during compile-time checking, it can be checked at run-time in the classical way, i.e., run-time tests are added to the program which encode in some way the given assertions. Introducing run-time tests by hand into a program is a tedious task and may introduce additional bugs in the program. In the preprocessor, this is performed automatically upon user's request.

Compile-time checking of assertions is conceptually more powerful than run-time checking. However, it is also more complex. Since the results of compile-time checking are valid for *any* query which satisfies the existing entry declarations, compile-time checking can be used both to detect that an assertion is violated and to prove that an assertion holds for any valid query, i.e., the assertion is validated. The main problem with compile-time checking is that it requires the existence of suitable static analyses which are capable of proving the properties of interest. For conciseness, we have shown the possibilities of our system using only a (regular) type analysis. However, the system is generic in that any program property (for which a suitable analysis exists in the system) can be used for debugging. As mentioned before, currently CiaoPP can infer types, modes and other variable instantiation properties, constraint independence, non-failure of predicates, determinacy, bounds on computational cost, bounds on sizes of terms in the program, and other properties.

More info: For more information, full versions of selected papers and technical reports, and/or to download Ciao and other related systems please access `http://www.clip.dia.fi.upm.es/`.

Acknowledgments: This work has been funded in part by MCYT projects CUBICO (TIC02-0055), EDIPIA (TIC99-1151) and ADELA (HI2000-0043), and by EU IST FET project ASAP (IST-2001-38059).

References

1. K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6):743–765, 1994.
2. K. R. Apt and D. Pedreschi. Reasoning about termination of pure PROLOG programs. *Information and Computation*, 1(106):109–157, 1993.
3. F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Programming Languages Design and Implementation'93*, pages 46–55, 1993.
4. J. Boye, W. Drabent, and J. Małuszyński. Declarative diagnosis of constraint programs: an assertion-based approach. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 123–141, Linköping, Sweden, May 1997. U. of Linköping Press.
5. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997. System and on-line version of the manual available at `http://clip.dia.fi.upm.es/Software/Ciao/`.
6. F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.
7. F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.
8. D. Cabeza and M. Hermenegildo. The Ciao Module System: A New Module System for Prolog. In *Special Issue on Parallelism and Implementation of (C)LP Systems*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.
9. B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.

10. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Proving properties of logic programs by abstract diagnosis. In M. Dams, editor, *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop*, number 1192 in Lecture Notes in Computer Science, pages 22–50. Springer-Verlag, 1996.
11. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1–3):43–93, 1999.
12. M. Comini, G. Levi, and G. Vitiello. Declarative diagnosis revisited. In *1995 International Logic Programming Symposium*, pages 275–287, Portland, Oregon, December 1995. MIT Press, Cambridge, MA.
13. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
14. P.W. Dart and J. Zobel. A Regular Type Language for Logic Programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–187. MIT Press, 1992.
15. P. Deransart. Proof methods of declarative properties of definite programs. *Theoretical Computer Science*, 118:99–166, 1993.
16. P. Deransart, M. Hermenegildo, and J. Maluszynski. *Analysis and Visualization Tools for Constraint Programming.* Number 1870 in LNCS. Springer-Verlag, September 2000.
17. W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. The Use of Assertions in Algorithmic Debugging. In *Proceedings of the Intl. Conf. on Fifth Generation Computer Systems*, pages 573–581, 1988.
18. W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. Algorithmic debugging with assertions. In H. Abramson and M.H.Rogers, editors, *Meta-programming in Logic Programming*, pages 501–522. MIT Press, 1989.
19. G. Ferrand. Error diagnosis in logic programming. *J. Logic Programming*, 4:177–198, 1987.
20. J. Gallagher and G. Puebla. Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In *Fourth International Symposium on Practical Aspects of Declarative Languages*, number 2257 in LNCS, pages 243–261. Springer-Verlag, January 2002.
21. J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In Pascal Van Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.
22. M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–615, September 1996.
23. M. Hermenegildo. A Documentation Generator for (C)LP Systems. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 1345–1361. Springer-Verlag, July 2000.
24. M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 International Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.
25. M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

26. P. Hill and J. Lloyd. *The Goedel Programming Language*. MIT Press, Cambridge MA, 1994.
27. Y. Lichtenstein and E. Y. Shapiro. Abstract algorithmic debugging. In R. A. Kowalski and K. A. Bowen, editors, *Fifth International Conference and Symposium on Logic Programming*, pages 512–531, Seattle, Washington, August 1988. MIT.
28. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.
29. G. Puebla, F. Bueno, and M. Hermenegildo. A Generic Preprocessor for Program Validation and Debugging. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 63–107. Springer-Verlag, September 2000.
30. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.
31. G. Puebla, F. Bueno, and M. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag, 2000.
32. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *JLP*, 29(1–3), October 1996.
33. C. Vaucheret and F. Bueno. More precise yet efficient type inference for logic programs. In *International Static Analysis Symposium*, number 2477 in LNCS, pages 102–116. Springer-Verlag, September 2002.
34. E. Vetillard. *Utilisation de Declarations en Programmation Logique avec Contraintes*. PhD thesis, U. of Aix-Marseilles II, 1994.
35. E. Yardeni and E. Shapiro. A Type System for Logic Programs. *Concurrent Prolog: Collected Papers*, pages 211–244, 1987.