A Framework for Assertion-based Debugging in Constraint Logic Programming

Germán Puebla, Francisco Bueno, and Manuel Hermenegildo
Department of Computer Science
Technical University of Madrid (UPM)
{german,bueno,herme}@disi.upm.es
(Extended Abstract)

1 Introduction

As (constraint) logic programming (CLP) systems [22] mature and larger applications are built, an increased need arises for advanced development and debugging environments. Such environments will likely comprise a variety of tools ranging from declarative debuggers to execution visualizers (see, for example, [1] for a more comprehensive discussion of tools and possible debugging scenarios). In this paper we concentrate our attention on the particular issue of program validation and debugging via direct static and/or dynamic checking of user-provided assertions.

We assume that a (partial) specification is available with the program and written in terms of assertions [5, 4, 13, 14, 23, 25]. Classical examples of assertions are the type declarations used in languages such as Gödel [21] or Mercury [29] (and in functional languages). However, herein we are interested in supporting a more general setting in which, on one hand assertions can be of a more general nature, including properties which are statically undecidable, and, on the other, only a small number of assertions may be present in the program; i.e., the assertions are optional. In particular, we do not wish to limit the programming language or the language of assertions unnecessarily in order to make the assertions statically decidable.

Consequently, the proposed framework needs to deal throughout with approximations [6, 11, 20]. It is imperative that such approximations be performed in a safe manner, in the sense that if an "error" (more formally, a symptom) is flagged, then it is indeed a violation of the specifications. However, while the system can be complete with respect to statically decidable properties (e.g., certain type systems), it cannot be complete in general, in the sense that when statically undecidable properties are used in assertions, there may be errors in the program with respect to such assertions that are not detected at compile time. This is a tradeoff that we accept in return for the greater flexibility. However, in order to detect as many errors as possible, the framework combines static (i.e., compile-time) and dynamic (i.e., runtime) checking of assertions. In particular, run-time checks will be generated for assertions which cannot be statically determined to hold or not.

Our approach is strongly motivated by the availability of powerful and mature static analyzers for (constraint) logic programs (see, e.g., [8, 16, 17, 24] and their references), generally based on abstract interpretation [11]. These systems can statically infer a wide range of properties (from types to determinacy or termination) accurately and efficiently, for realistic programs. Thus, we would like to take advantage of standard program analysis tools, rather than developing new abstract procedures, such as concrete [4, 13, 14] or abstract [9, 10] diagnosers and debuggers, or using traditional proof-based methods [2, 3, 12, 15, 30].

Figure 1 presents the general architecture of the type of debugging environment that we propose. Hexagons represent the different tools involved and arrows indicate the communication paths among such tools. Most of such communication is performed in terms of assertions. More details on the assertion language can be found in [25].

As mentioned above, we assume that a (partial) specification of the intended meaning or behavior of the (possibly partially developed) program (i.e., the user requirements) is available and written in terms of assertions. Because these assertions are to be checked we will refer to them as "check" assertions. All these assertions (and those which will be mentioned later) are written in the same syntax, with a prefix denoting their status (check, true, ...). The program analyzer generates an approximation of the actual semantics of the program, expressed in the form of true assertions (in the case of CLP programs standard analysis techniques -e.g., [17, 16]- are used for this purpose). The comparator, using the analyzer's abstract operations, compares the user requirements and the information generated by the analysis. This process produces three different kinds of results, which are in turn represented by three different kinds of assertions:

- Verified requirements (represented by checked assertions).
- Requirements identified not to hold (represented by false assertions). In this case an abstract symptom has been found and diagnosis should start.
- None of the above, i.e., the analyzer/comparator pair cannot prove that a requirement holds nor that it does not hold (and some assertions remain in check status). Run-time tests are then introduced to test the requirement (which may produce "concrete" symptoms during program testing). Clearly, this may introduce significant overhead and can be turned off after program testing.

Given this overall design, in this work we concentrate on formally defining a series of assertions and the notions of correctness and errors of a program with respect to these

- The user may optionally provide additional information to the analyzer by means of "try" assertions (which describe the external calls to a module) and "true" assertions (which provide abstract information on a predicate that the analyzer can use even if it cannot prove it) [5, 26].
assertions. We then present techniques for static and dynamic checking of the assertions. More details on the use of assertions and the framework from a user’s perspective can be found in [18].

2 Preliminaries and Notation

A constraint is essentially a conjunction of predefined predicates (such as term equations or inequalities over the reals) whose arguments are constructed using predefined functions (such as real addition). We let \( \exists \psi \theta \) be constraint \( \theta \) restricted to the variables \( W \).

An atom has the form \( p(t_1, \ldots, t_n) \) where \( p \) is a predicate symbol and the \( t_i \) are terms. A literal is either an atom or a primitive constraint. A goal is a finite sequence of literals. A rule is of the form \( H \leftarrow B \) where \( H \), the head, is an atom with distinct variables as arguments and \( B \), the body, is a possibly empty finite sequence of literals. A constraint logic program, or program, is a finite set of rules. The definition of an atom \( A \) in program \( P \), \( \text{def}_{\psi}(A) \), is the set of variable renamings of rules in \( P \) such that each renaming has \( A \) as a head and has distinct new local variables.

We assume that all rule heads are normalized, i.e., \( H \) is of the form \( p(X_1, \ldots, X_n) \) where \( X_1, \ldots, X_n \) are distinct free variables. This is not restrictive since programs can always be normalized.

The operational semantics of a program is in terms of its “derivations” which are sequences of reductions between “states”. A state \( \langle G \mid \theta \rangle \) consists of the current goal \( G \) and the current constraint store (or store for short) \( \theta \). A state \( \langle L :: G \mid \theta \rangle \) where \( L \) is a literal can be reduced as follows:

1. If \( L \) is a primitive constraint and \( \theta \wedge L \) is satisfiable, it is reduced to \( \langle G \mid \theta \wedge L \rangle \).
2. If \( L \) is an atom, it is reduced to \( \langle B :: G \mid \theta \rangle \) for some rule \( \langle L \leftarrow B \rangle \in \text{def}_{\psi}(L) \).

where :: denotes concatenation of sequences and we assume for simplicity that the underlying constraint solver is complete. A derivation from state \( S \) for program \( P \) is a sequence of states \( S_0 \rightsquigarrow P S_1 \rightsquigarrow P \ldots \rightsquigarrow P S_n \), where \( S_0 = S \) and there is a reduction from each \( S_i \) to \( S_{i+1} \). Given a non-empty derivation \( D \), we denote by \( \text{curr-state}(D) \) and \( \text{curr-store}(D) \) the last state in the derivation, and the store in such last state, respectively. Eg., if \( D \) is the derivation \( S_0 \rightsquigarrow P S_n \) with \( S_n = \langle G \mid \theta \rangle \) then \( \text{curr-state}(D) = S_n \) and \( \text{curr-store}(D) = \theta \).

A query is a pair \( \langle L, \theta \rangle \) where \( L \) is a literal and \( \theta \) a store for which the CLP systems starts a computation from state \( \langle L \mid \theta \rangle \). The set of all derivations from \( \theta \) is denoted \( \text{derivations}(P, \theta) \). We will denote sets of queries by \( Q \). We extend \( \text{derivations} \) to operate on sets of queries as follows: \( \text{derivations}(P, Q) = \bigcup_{\psi \theta \in Q} \text{derivations}(P, \theta) \).

The observational behavior of a program is given by its “answers” to queries. A finite derivation from a state \( S \) for program \( P \) is finished if the last state in the derivation cannot be reduced. A finished derivation from a state \( S \) is successful if the last state has form \( \langle \text{nil} \mid \theta \rangle \), where \( \text{nil} \) denotes the empty sequence. The constraint \( \exists \psi_{\text{def}(S)} \theta \) is an answer to \( S \). A finished derivation is failed if the last state is not of the form \( \langle \text{nil} \mid \theta \rangle \).

3 Assertions

Assertions are linguistic constructions which allow expressing properties of programs. The properties can relate to the program execution, particular derivations, or execution states.

Definition 3.1 [Assertion] An assertion \( A \) for a program \( P \) is a pair \( \langle \text{app}_A, \text{val}_A \rangle \) s.t. both \( \text{app}_A \) and \( \text{val}_A \) are first-order logic formulae and \( \text{app}_A(D) \) and \( \text{val}_A(\text{curr-store}(D)) \) are decidable for any derivation \( D \) for \( P \).

As an intuition, the role of \( \text{app}_A \) is to indicate the execution states in which \( A \) is applicable. For the assertion to hold, \( \text{val}_A \) should take the value \( \text{true} \) for the corresponding constraint store in any applicable state of \( A \).

Note that this is a very open definition of assertions. In the following we provide some more specific schemas for assertions which correspond to the assertions traditionally used: i.e., \( \text{pre} \) and \( \text{post} \) conditions. For each of these schemas we provide the meaning of the logic formulae associated to \( \text{app} \) and \( \text{val} \) corresponding to the assertion. An example program annotated with assertions of this kind is shown in Figure 2, where two assertions \( A_1 \) and \( A_2 \) are provided in the schema oriented syntax that we use herein, as well as in the program oriented syntax of [25]. In the figure, \( A_1 \) expresses that if \( \text{query} \) is called with its first argument being a list then upon success (if it succeeds) its second argument is a sorted list, and \( A_2 \) expresses that \( \text{partition} \) is expected to be called with its first argument a list. These assertions refer...
:- success qsort(A,B) : list(A) => (list(B), sorted(B)). % A1
% A1: { success( qsort(A,B) , list(A) , (list(B) and sorted(B)) ) }
qsort([X|L],R) :-
  partition(L,X,L1,L2),
  qsort(L2,R2), qsort(L1,R1),
  append(R1,[X|R2],R).
qsort([],[]).
qsort([X|Y|Z|X],[]) :- X =< Y, sorted([Y,Z|X]).
qsort([X|Y|Z|X],[W]) :- X =< Y, sorted([Y,Z|X]), list([W]).
qsort([X|Y|Z|X],[W],[V]) :- X =< Y, sorted([Y,Z|X]), list([V,W]).

Figure 2: An Example Program Annotated with Assertions

to particular execution states in derivations in which qsort (resp. partition) are involved. We say that these assertions are "evaluable" only in such states.

Definition 3.2 [Evaluation of an Assertion for a Derivation] Given an assertion A = (appA, valA) for program P, the evaluation of A for a derivation D is solve(A,D,P) =

\[ \forall r : appA(r(D)) \rightarrow valA(curr_store(r(D))) \]

where r is a variable renaming which relates the variable names in A with the variables in a concrete derivation D.

3.1 Assertion Schemas

Assertion Schemas are expressions which given a syntactic object AS produce an assertion A = (appA, valA) by syntactic manipulation only. In other words, assertion schemas are syntactic sugar for writing certain kinds of assertions which are used very often. Assertions described using the given assertion schemas will be denoted as AS in order to distinguish them from the actual assertion (i.e., a pair of logic formulae) A = (appA, valA).

Calls Assertions: This assertion schema is used to describe execution states of the calls possible to a predicate. Given an expression AS = calls(p, Precond), we obtain an assertion A whose appAS and valAS are defined as follows:

\[ app_{calls}(p, Precond)(D) = \begin{cases} true & \text{if current state}(D) = (p::G I \theta) \\ false & \text{otherwise} \end{cases} \]

\[ val_{calls}(p, Precond)(\theta) = Precond(\bar{\exists}_{vars(p)}(\theta)) \]

Clearly, there is no way an assertion calls(p, Precond) can be violated unless the next predicate to be executed, i.e., the leftmost literal in the goal of the current state, is p.

Success Assertions: Success assertions are used in order to express postconditions of predicates. These postconditions may be required to hold on success of the predicate for any call to the predicate, i.e., the precondition is true, or only for calls satisfying certain preconditions. Given an expression AS = success(p, Pre, Post), we obtain an assertion A whose appAS and valAS are defined as follows:

\[ app_{success}(p, Pre, Post)(D) = \begin{cases} true & \text{if current state}(D) = (G I \theta) \text{ and } \exists(p::G I \theta) \in D \text{ and } Pre(\bar{\exists}_{vars(p)}(\theta)) \\ false & \text{otherwise} \end{cases} \]

\[ val_{success}(p, Pre, Post)(\theta) = Post(\bar{\exists}_{vars(p)}(\theta)) \]

Note that, for a given assertion A and derivation D, several states of the form (p::G I \theta) may exist in D. As a result, the assertion A will have to be checked several times with different renamings so that the variables of the assertion are related to different states in D.

3.2 Assertions and Debugging

Assertions have often been used for performing debugging with respect to partial correctness, i.e., to ensure that the program does not produce unexpected results for valid queries. The framework allows restricting correctness checking to those queries which are "expected". The set of valid queries to the program are represented by Q. In this section we provide several simple definitions which will be instrumental.

Definition 3.3 [Error Set] Given an assertion A, the error set of A in a program P for a set of queries Q is E(A,P,Q) = \{ D \in \text{derivations}(P,Q) | \neg \text{solve}(A,D,P) \}.

Definition 3.4 [False Assertion] An assertion A is false in a program P for a set of queries Q iff E(A,P,Q) \neq \emptyset.

Definition 3.5 [Checked Assertion] An assertion A is checked in a program P for a set of queries Q iff E(A,P,Q) = \emptyset.

The definitions of false and checked assertions are complementary, thus, it is clear that given a program P and a set of queries Q, any assertion A is either false or checked. The goal of assertion checking is to determine whether each assertion A is false or checked in P for Q. There are two
kinds of approaches to doing this. One is based on actually trying all possible execution paths (derivations) for all possible queries. When it is not possible to try all derivations an alternative is to explore a hopefully representative set of them. This approach is explored in Section 4. The second approach is to use global analysis techniques and is based on computing safe approximations of the program behavior statically. This approach is studied in Section 5.

Definition 3.6 [Partial Correctness] A program P is partially correct w.r.t. a set of assertions A and a set of queries Q iff \( \forall A \in A \text{ a query in } G \text{ is checked in } P \text{ for } Q \).

If all the assertions are checked, then the program is partially correct. Thus, our framework is of use both for validation and for detection of errors.

Finally, in addition to checked and false assertions, we will also consider true assertions. True assertions differ from checked in that true assertions hold of the program for any set of queries Q.

Definition 3.7 [True Assertion] An assertion A is true in program P iff \( \forall Q: E(A, P, Q) = \emptyset \).

Clearly, any assertion which is true in P is also checked for any Q, but not vice-versa. Since true assertions hold for any possible query they can be regarded as query-independent properties of the program. Thus, true assertions can be used to express analysis information as already done, for example, in [5]. This information can then be reused when analyzing the program for different queries.

4 Run-Time Checking of Assertions

The main idea behind run-time checking of assertions is, given a program P, a set of queries Q, and a set of assertions A, to directly apply Definitions 3.4 and 3.5 in order to determine whether the assertions in A are checked or false. It is not to be expected that Definition 3.5 can be used to determine that an assertion is checked as this would require checking the derivations from all valid queries which is in general an infinite set and thus checking would not terminate. In this situation, and as mentioned before, an alternative is to perform run-time checking for a hopefully representative set of queries. Though this does not allow fully validating the program in general, it allows detecting many incorrectness problems.

An important observation is that in constraint logic programming, and under suitable assumptions, it is possible to use the underlying logic inference system for checking whether the given assertions (logic formulae) hold or not. In order to be able to perform run-time checking in this way, we require that \( \text{Precond}(\theta) \) of an assertion \( \text{calls}(p, \text{Precond}) \), and \( \text{Pre}(\theta) \) and \( \text{Post}(\theta) \) of an assertion \( \text{success}(p, \text{Pre}, \text{Post}) \) can be computed in the CLP system. To this end, we restrict the admissible pre and post conditions of assertions to those which can be expressed as CLP programs. We argue that this is not too strong a restriction given the high expressive power of CLP languages. Note that the approach also implies that the program P must contain the definitions for the pre and post conditions used in assertions (Figure 2). We believe that this choice of a language for writing conditions is in fact of practical interest because it facilitates the job of programmers, which do not need to learn a specification language in addition to the CLP language.

For simplicity, in the formalization (but not in the implementation) pre and post conditions are assumed to be literals (rather than for example goals or disjunctions of goals). Note, however, that this is not a restriction since given a logic expression built using literals, conjunctions, and disjunctions, it is always possible to write a predicate whose (declarative) semantics is equivalent to the such logic expression. Also, it is crucial to ensure that run-time checking does not introduce non-termination into terminating programs. As a result, not all possible predicates which can be written in a CLP language can be used as properties in assertions:

Definition 4.1 [Test] A literal L is a test iff \( \forall \theta : \text{derivations}(P; (L, \theta)) \) is finite.

Only tests are admissible as pre and post conditions in assertions.

Definition 4.2 [Trivially Succeeds] A literal L trivially succeeds for \( \theta \) in P, denoted \( \theta \Rightarrow L \) if \( \exists \) a successful derivation for \( \{L \mid \theta \} \) with answer \( \theta \) s.t. \( \text{assert}(L) \theta = \theta \).

Theorem 4.3 [Checking of Tests] Let \( t \) be a test defined in a program P. \( t(\theta) \) holds iff \( \theta \Rightarrow t \).

Theorem 4.3 guarantees that checking of pre and post conditions, which are required to be tests, is complete since the set of derivations (search space) is finite.

4.1 A Program Transformation for Run-Time Checking

We now present a program transformation technique which given a program P, obtains another program \( P' \) which checks the assertions while running on a standard CLP system.

The program transformation from P into P' given a set of assertions A is as follows. Let \( \text{new}(P, p) \) denote a function which returns an atom of a new predicate symbol different from all predicates defined in P with same arity and arguments as p. Let \( \text{renaming}(A, p, p') \) denote a function which returns a set of assertions identical to A except for the assertions renamed to p which are now referred to \( p' \), and let \( \text{renaming}(P, p, p') \) denote a function which returns a set of rules identical to P except for the rules of predicate p which are now referred to \( p' \). We obtain \( P' = \text{rtchecks}(A, P) \), where:

\[
\text{rtchecks}(A, P) = \begin{cases} 
\text{rtchecks}(A', P') & \text{if } A = \{A \} \cup A'' \\
\text{rtchecks}(A, P) & \text{if } A = \emptyset 
\end{cases}
\]

where

\[
A' = \text{renaming}(A', p, p') \\
P' = \text{renaming}(P, p, p') \cup \{CL\} \\
P' = \text{new}(P, p) \\
CL = \begin{cases} 
\text{p; check}(C, A), p' & \text{if } A = \text{calls}(p, C) \\
\text{p; ts}(C) & \text{if } A = \text{success}(p, C, S)
\end{cases}
\]

As usual, the construct \( \text{cond -> then : else} \) is the Prolog if-then-else. The program above contains two undefined predicates \( \text{check}(C, A) \) and \( \text{ts}(C) \). \( \text{check}(C, A) \) must check whether C holds or not and raise an error if it does not. \( \text{ts}(C) \) must return true iff for the current constraint store \( \theta, \theta \Rightarrow p \). As an example, for the particular case of
5 Complete-Time Checking

In this section, we present some techniques which allow in
certain cases determining at complete-time the results of run-
time assertions. With this aim, we assume the existence of a

5.1 Abstract Interpretation

An interpretation [11] is a technique for static program
analysis, in which executable program is described on
an abstract domain. It is used for the purpose of produc-
ing information about the program while it is running.

5.2 Exploiting Information from Abstract

Interpretation

Before presenting the actual sufficient conditions that we
present some definitions and results which will then be
instrumental.

Definition 5.3 (Abstract Trial Success Subset). An ab-

tract substitute $A \subseteq L$ is an abstract trial success subset
of $L$ in $P$ if $A$ is an abstract trial success subset of $L$ in
$\Sigma_{L}(P)$.

Definition 5.4 (Abstract Trial Success Subspace). An ab-

tract subspace $A$ is an abstract trial success subspace
of $L$ in $P$ if $A$ is an abstract trial success subset of $L$ in
$\Sigma_{L}(P)$.

Definition 5.5 (Abstract Trial Subspace). An abstract

subspace $A$ is an abstract trial subspace of $L$ in $P$ if $A$ is an
abstract trial success subset of $L$ in $\Sigma_{L}(P)$.

Definition 5.6 (Abstract Trial Success Superset). An ab-

tract substitute $A$ is an abstract trial success subset of $L$ in
$\Sigma_{L}(P)$ if $A$ is an abstract trial success subset of $L$ in
$P$.

Definition 5.7 (Abstract Trial Success Subset). An ab-

tract substitute $A$ is an abstract trial success subset of $L$ in
$\Sigma_{L}(P)$ if $A$ is an abstract trial success subset of $L$ in
$P$.

In order to apply Lemmas 5.3 and 5.7, effectively ac-
curate and well-founded, an abstract program with the set of
abstract queries $\mathcal{Q}$ can simply be done by analyzing the
program with the set of abstract queries $\mathcal{Q}$.
$Q_\alpha = \{(L, T)\}$. Since our analysis is goal-dependent, the initial abstract substitution $T$ is used in order to guarantee that the information which will be obtained is valid for any call to $L$. The result of analysis will contain a tuple of the form $(L, T, \lambda^*)$ and thus we can take $\lambda_{TS(L, P)} = \lambda^*$, as correctness of the analysis guarantees that $\lambda^*$ is a superset approximation of $TS(L, P)$.

Unfortunately, obtaining a (non-trivial) correct $\lambda_{TS(L, P)}$ in an automatic way is not so easy, assuming that analysis provides superset approximations. In [27], correct $\lambda_{TS(L, P)}$ for built-in predicates were computed by hand and provided to the system as a table of "built-in abstract behaviors". This is possible because the semantics of built-ins is known in advance and does not depend on $P$ (also, computing by hand is well justified in this case because, in general, code for built-ins is not available since for efficiency they are often written in a lower-level language — e.g., C— and analyzing their definition is thus not straightforward).

In the case of user defined predicates, precomputing $\lambda_{TS(L, P)}$ is not possible since their semantics is not known in advance. However, the user can provide trust assertions which provide this information. Also, since in this case the code of the predicate is present, analysis of the definition of $L$ can also be applied and will be effective if analysis is precise for $L$, i.e., $\gamma(L) = \bigcup_{x \in \gamma(L)} S(p, \theta, P, Q)$ rather than $\gamma(L) \supseteq \bigcup_{x \in \gamma(L)} S(p, \theta, P, Q)$. In this situation we can use $\lambda^*$ as the (best possible) $\lambda_{TS(L, P)}$. Requiring that the analysis be precise for any arbitrary literal $L$ is not realistic. However, if the success set of $L$ corresponds exactly to some abstract substitution $\lambda_L$, i.e. $TS(L, P) = \gamma(\lambda_L)$, then analysis can often be precise enough to compute $(L, \lambda^*, \lambda^*)$ with $\lambda^* = \lambda_L$. This implies that not all the tests the user could write are checkable at compile-time, but only those of them which coincide with some abstract substitution. This means that if we only want to perform compile-time checking, then it is best to use tests which are perfectly captured by the abstract substitution.

An interesting situation in which this occurs is the use of regular programs as type definitions (as in Figure 2). There is a direct mapping from type definitions (i.e., the abstract values in the domain) to regular programs and vice-versa which allows accurately relating any abstract value to any program defining a type (i.e., to any regular program).

5.3 Checked Assertions

In this section we provide sufficient conditions for proving at compile-time that an assertion is never violated. Detecting checked assertions at compile-time is quite useful. First, if all assertions are found to be checked, then the program has been validated. Second, even if only some assertions are found to be checked, performing run-time checking for those assertions can be avoided, thus improving efficiency of the program with run-time checks. Finally, knowing that some assertions have been checked also allows the user to focus debugging on the remaining assertions.

Theorem 5.8 [Checked Calls Assertion] Let $P$ be a program, $calls(p, Precond)$ an assertion, $Q$ a set of queries, and let $Q_{\alpha}$ be s.t. $Q_0 \supseteq Q$. Assume that $(p, \lambda^*, \lambda^*) \in Analysis(P, Q_{\alpha}, D_{\alpha})$. If $\lambda^* \subseteq \lambda_{TS(Precond, P)}$ then $A$ is checked in $P$ for $Q$.

Theorem 5.8 states that there are two situations in which a calls assertion is checked. Case 1 indicates that the predicate $P$ is never reached during execution, and thus the precondition does not need to be tested. Case 2 indicates that the precondition holds for all stores in the calling context.

Theorem 5.9 [Checked Success Assertion] Let $P$ be a program, $success(p, Pre, Post)$ an assertion, $Q$ a set of queries, and let $Q_{\alpha}$ be s.t. $\gamma(Q_{\alpha}) \supseteq Q$. Assume that $(p, \lambda^*, \lambda^*) \in Analysis(P, Q_{\alpha}, D_{\alpha})$. If

1. $\lambda^* \cap \lambda_{TS(Pre, P)}^+ = \bot$, or
2. $\lambda^* \subseteq \lambda_{TS(Post, P)}^-$

then $A$ is checked in $P$ for $Q$.

Theorem 5.9 states that there are two situations in which a success assertion is checked. Case 1 indicates that the precondition is never satisfied, and thus the postcondition does not need to be tested. Case 2 indicates that the postcondition holds for all stores in the success contexts, which is a superset of the applicability set of the assertion.

5.4 False Assertions

The aim of this section is to find sufficient conditions which ensure statically that there is an erroneous derivation $D \in derivations(P, Q)$, i.e., without having to actually compute derivations($P, Q$). Unfortunately, this is a bit trickier than it may seem at first sight if analysis over-approximates computation states, is as the usual case.

Theorem 5.10 [False Calls Assertion] Assuming the premises of Theorem 5.8, if $\lambda_{TS(Precond, P)}^+ \not\subseteq \lambda^*$ and $(p, \lambda^*, \lambda^*) \in Analysis(P, Q_{\alpha}, D_{\alpha})$, then $A$ is false in $P$ for $Q$.

In order to prove that a calls assertion is false it is not enough to prove that $(p, \lambda_{TS(Precond, P)}^+) \not\subseteq \lambda^*$ as the contexts which violate the assertion may not appear in the real execution but rather may have been introduced due to the loss of accuracy of analysis w.r.t. the actual computation. Furthermore, even if $\lambda^*$ and $\lambda_{TS(Precond, P)}^+$ are incompatible, it may be the case that there are no calls for predicate $P$ in derivations($P, Q$) (and analysis is not capable of detecting so). This is why the condition $C(p, P, Q) \neq \emptyset$ is also required.

Theorem 5.11 [False Success Assertion] Assuming the premises of Theorem 5.9, if

1. $\lambda^* \cap \lambda_{TS(Pre, P)}^+ = \bot$ and
2. $\lambda^* \cap \lambda_{TS(Post, P)}^- = \bot$ and $\exists \theta \in \gamma(\lambda^*) \cap \lambda_{TS(Pre, P)}^+$ such that $S(p, \theta, P, Q) \neq \emptyset$.

then $A$ is false in $P$ for $Q$.

Now again, $\lambda^*$ is an over-approximation, and in particular it can approximate the empty set. This is why the extra condition $\exists \theta \in \gamma(\lambda^*) \cap \lambda_{TS(Pre, P)}^+$ such that $S(p, \theta, P, Q) \neq \emptyset$ is required.

If an assertion $A$ is false then the program is not correct w.r.t. $A$. Detecting the minimal part of the program responsible for the incorrectness, i.e., diagnosis of a static symptom is an interesting problem. This is subject of on-going research.
5.5 True Assertions

As with checked assertions, if an assertion is true then it is guaranteed that it will raise any error. From the point of view of assertion checking, the only difference between them is that checked assertions may raise errors if the program were used with a different set of queries.

Note that an assertion \texttt{calls(p, Precond)} can never be found to be true, as the calling context of \texttt{p} depends on the query. If we pose no restriction on the queries we can always find a calling state which violates the assertion, unless \texttt{Precond} is a tautology.

\textbf{Theorem 5.12} [True Success Assertion] Assuming the premises of Theorem 5.9, if

1. \(\lambda^+_T(S(Pre, P)) \subseteq \lambda^c\), and
2. \(\lambda^c \subseteq \lambda^-_T(Post, P)\)

then \(A\) is true in \(P\).

Condition 1 guarantees that \(\lambda^c\) describes any store which is a descendant of a calling state of \(p\) which satisfied the precondition. Condition 2 ensures that any store described by \(\lambda^c\) satisfies the postcondition. Thus, any store in the success context which originated from a calling state which satisfied the precondition satisfies the postcondition.

5.6 Equivalent Assertions

It may be the case that some assertions are not detected as checked or false at compile-time. However, it is possible some that part of the assertion can be replaced at compile time by a simpler one, i.e., one which can be checked more efficiently.

\textbf{Definition 5.13} [Equivalent Assertions] Two assertions \(A, A'\) are equivalent in program \(P\) for a set of queries \(Q\) iff \(E(A, P, Q) = E(A, P, Q)\).

If \(A\) and \(A'\) are equivalent but \(A'\) is simpler then obviously \(A'\) should be used instead for run-time checking. Generating equivalent Compile-time simplification of assertions can be done using techniques such as abstract specialization (see, e.g., \cite{28, 27}). However, space limitations prevent us from discussing further this interesting issue.

6 Implementation

We have implemented the schema of Figure 1 as a generic framework. This genericity means that different instances of the tools involved in the schema for different CLP dialects can be generated in a straightforward way. Currently, two different experimental debugging environments have been developed using the proposed framework: \texttt{ciaopp} \cite{18}, the CIAO system preprocessor, developed by UPN, and \texttt{chipre} \cite{7}, an assertion-based type inferencing and checking tool also developed at UPN in collaboration with Pawel Pietrzak from the U. of Linköping. The analysis used is an adaptation to CLP(FD) of the regular approach of \cite{16}. \texttt{chipre} has been interfaced to Cosytec with the CHIP system (adding a graphical user interface) and is currently under industrial evaluation.

\texttt{CIAO} is a next-generation, GNU-licensed Prolog system. The language subsumes standard ISO-Prolog and is specifically designed to be very extensible and to support modular program analysis, debugging, and optimization. CIAO is based on the &-Prolog/SICStus concurrent Prolog engine.

\texttt{ciaopp}, the CIAO precompiler, can perform a number of tasks, including: (a) inference of properties of program predicates and literals, including types (using \cite{16}), modes and other variable instantiation properties (using the CLP\textsuperscript{+} version of the PLAI abstract interpreter \cite{17}), non-failure, determinacy, bounds on computational cost, bounds on sizes of terms in the program, etc. (b) Static debugging including checking how programs call system libraries and also the assertions present in other modules used by the program. (c) Several kinds of source to source program transformations such as specialization, parallelization, inclusion of run-time tests, etc.

Information generated by analysis, assertions in system libraries, and any assertions optionally included in user programs are all written in the CIAO assertion language \cite{26, 25}, of which in this work we have only addressed a subset, due to space limitations. The assertion language is also used by an automatic documentation generator \cite{19} for LP/CLP programs based on program assertions and machine-readable comments. Generates manuals in many formats including postscript, pdf, info, HTML, etc.

\textbf{Acknowledgments}

This work has been supported in part by ESPRIT project DISCIP\textsuperscript{+} and CICYT project ELLA. The authors would also like to thank Jan Małuszyński, Wiodek Drabent and Pierre Demonet for many interesting discussions on assertions and to Pawel Pietrzak for adapting John Gallagher's type analysis for CLP(FD).

\textbf{References}


Report CLIP1/99.1, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, March 1999. Also as deliverable of the ESPRIT project DISCPL.


