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}@fi.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 diagnosers 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., run-time) 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., [5, 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 before, 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.[1] All these assertions (and those which will be mentioned later) are written in the same syntax, with a prefix denoting their status (*check*, *trust*, ...). 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 those

[1] The user may optionally provide additional information to the analyzer by means of "*entry*" assertions (which describe the external calls to a module) and "*trust*" assertions (which provide abstract information on a predicate that the analyzer can use even if it cannot prove it) [5, 26].

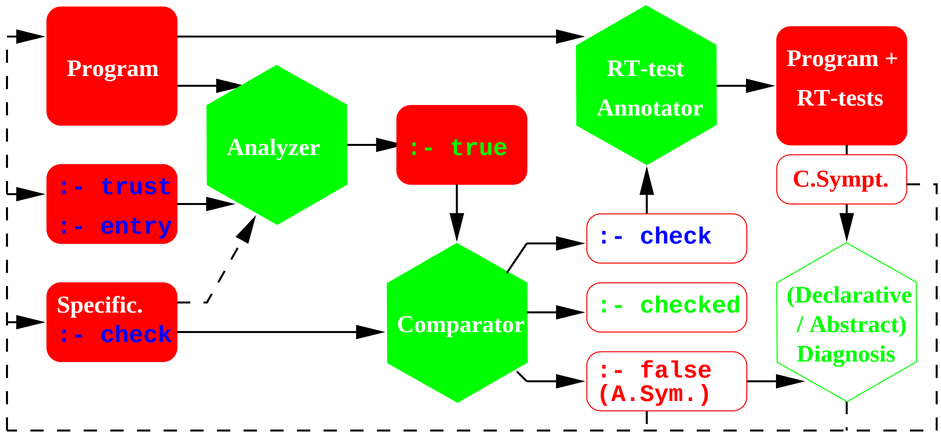

Figure 1: A Combined Framework for Program Development and Debugging

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 $\overline{\exists}_W \theta$ be constraint θ restricted to the variables W.

An *atom* has the form $p(t_1, ..., t_n)$ 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 `:-` 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, $defn_P(A)$, 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 $p(X_1, ..., X_n)$ where $X_1, ..., X_n$ 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 $\langle G \,|\, \theta \rangle$ consists of the current goal G and the current constraint store (or *store* for short) θ. A state $\langle L :: G \,|\, \theta \rangle$ 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 $\langle G \,|\, \theta \wedge L \rangle$.
2. If L is an atom, it is reduced to $\langle B :: G \,|\, \theta \rangle$ for some rule $(L$ `:-` $B) \in defn_P(L)$.

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 $S_0 \leadsto_P S_1 \leadsto_P ... \leadsto_P S_n$ where S_0 is S and there is a reduction from each S_i to S_{i+1}. Given a non-empty derivation D, we denote by *curr_state(D)* and *curr_store*(D) the last state in the derivation, and the store in such last state, respectively. E.g., if D is the derivation $S_0 \leadsto_P S_n$ with $S_n = \langle G \,|\, \theta \rangle$ then $curr\_state(D) = S_n$ and $curr\_store(D) = \theta$. A *query* is a pair (L, θ) where L is a literal and θ a store for which the CLP systems starts a computation from state $\langle L \,|\, \theta \rangle$. The set of all derivations from Q for P is denoted $derivations(P, Q)$. We will denote sets of queries by $\mathcal{Q}$. We extend *derivations* to operate on sets of queries as follows: $derivations(P, \mathcal{Q}) = \bigcup_{Q \in \mathcal{Q}} derivations(P, Q)$.

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 $\langle nil \,|\, \theta \rangle$, where *nil* denotes the empty sequence. The constraint $\overline{\exists}_{vars(S)} \theta$ is an *answer to* S. A finished derivation is *failed* if the last state is not of the form $\langle nil \,|\, \theta \rangle$.

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 (app_A, val_A) s.t. both app_A and val_A are first-order logic formulae and $app_A(D)$ and $val_A(curr\_store(D))$ are decidable for any derivation D for P.

As an intuition, the role of app_A is to indicate the execution states in which A is *applicable*. For the assertion to hold, val_A should take the value *true* for the corresponding constraint store in every 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., *pre* and *post* conditions. For each of these schemas we provide the meaning of the logic formulae associated to *app* and *val* corresponding to the assertion. An example program annotated with assertions of this kind is shown in Figure 2, where two assertions `A1` and `A2` are provided in the schema oriented syntax that we use herein, as well as in the program oriented syntax of [25]. In the figure, `A1` expresses that if `qsort` is called with its first argument being a list then upon success (if it succeeds) its second argument is a sorted list, and `A2` expresses that `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([],[]).

:- calls partition(A,B,C,D) : list(A).              % A2
% A2: { calls( partition(A,B,C,D) , list(A) ) }
partition([],B,[],[]).
partition([E|R],C,[E|Left1],Right):- E < C, !,
        partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):- E >= C,
        partition(R,C,Left,Right1).

sorted([]).                                     list([]).
sorted([_]).                                    list([_|L]):-
sorted([X,Y|L]):- X =< Y, sorted([Y|L]).                list(L).
```

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 $A = (app_A, val_A)$ for program P, the evaluation of A for a derivation D is $solve(A, D, P) =$

$$\forall r : app_A(r(D)) \rightarrow val_A(curr\_store(r(D))).$$

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 $A = (app_A, val_A)$ 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) $A = (app_A, val_A)$.

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

$$app_{calls(p,Precond)}(D) = \begin{cases} true & \text{if } current\_state(D) = \langle p :: G \,\rule{1pt}{6pt}\, \theta\rangle \\ false & \text{otherwise} \end{cases}$$

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

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 app_{AS} and val_{AS} are defined as follows:

$$app_{success(p,Pre,Post)}(D) = \begin{cases} true & \text{if } current\_state(D) = \langle G \,\rule{1pt}{6pt}\, \theta\rangle \text{ and} \\ & \exists\langle p :: G \,\rule{1pt}{6pt}\, \theta'\rangle \in D \text{ and } Pre(\bar{\exists}_{vars(p)}\theta') \\ false & \text{otherwise} \end{cases}$$

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

Note that, for a given assertion A and derivation D, several states of the form $\langle p :: G \,\rule{1pt}{6pt}\, \theta'\rangle$ 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 $\mathcal{Q}$. 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 $\mathcal{Q}$ is $E(A, P, \mathcal{Q}) = \{D \in derivations(P, \mathcal{Q}) | \neg solve(A, D, P)\}$.

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

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

The definitions of *false* and *checked* assertions are complementary, thus, it is clear that given a program P and a set of queries $\mathcal{Q}$, 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 $\mathcal{Q}$. 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 $\mathcal{A}$ and a set of queries $\mathcal{Q}$ iff $\forall A \in \mathcal{A}$ A is checked in P for $\mathcal{Q}$.

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 $\mathcal{Q}$.

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 $\mathcal{Q}$, 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 $\mathcal{Q}$, and a set of assertions $\mathcal{A}$, to directly apply Definitions 3.4 and 3.5 in order to determine whether the assertions in $\mathcal{A}$ 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 $Precond(\theta)$ of an assertion $calls(p, Precond)$, and $Pre(\theta)$ and $Post(\theta)$ of an assertion $success(p, Pre, Post)$ 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 $\forall\theta$: $derivations(P, (L, \theta))$ is finite.

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

Definition 4.2 [Trivially Succeeds] A literal L *trivially succeeds* for θ in P, denoted $\theta \Rightarrow_P L$, if $\exists$ a successful derivation for $\langle L \,|\, \theta\rangle$ with answer θ' s.t. $\bar{\exists}_{vars(L)}\theta' = \theta$.

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

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 an 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 $\mathcal{A}$ is as follows. Let $new(P,p)$ 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 $renaming(\mathcal{A}, p, p')$ denote a function which returns a set of assertions identical to $\mathcal{A}$ except for the assertions referred to p which are now referred to p', and let $renaming(P, p, p')$ 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 $P' = rtchecks(\mathcal{A}, P)$, where:

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

where

$$\begin{aligned}
&\mathcal{A}' = renaming(\mathcal{A}'', p, p') \\
&P' = renaming(P, p, p') \cup \{CL\} \\
&p' = new(P, p) \\
&CL = \\
&\begin{cases} p\texttt{:-}check(C, A),\ p'. & \text{if } A = calls(p, C) \\ p\texttt{:-}(ts(C)\texttt{->}p', check(S, A); p'). & \text{if } A = success(p, C, S) \end{cases}
\end{aligned}$$

As usual, the construct ($cond$`->` $then$; $else$) is the Prolog if-then-else. The program above contains two undefined predicates: $check(C, A)$ and $ts(C)$. $check(C, A)$ must check whether C holds or not and raise an error if it does not. $ts(C)$ must return true iff for the current constraint store θ, $\theta \Rightarrow_P C$. As an example, for the particular case of

Prolog, $check(C, A)$ can be defined as "`check(C,A) :- ( ts(C) -> true ; error(A) ).`" where `error(A)` is a predicate which informs about the false assertion A. `ts(C)` can be defined as "`ts(C) :- copy_term(C,C1), call(C1), variant(C,C1).`".

Theorem 4.4 [Program Transformation] Let P be a program and $\mathcal{A}$ a set of assertions. Let $P' = rtchecks(\mathcal{A}, P)$. If during the execution of P' for a query Q a literal $error(A)$ is executed then A is false and $E(A, P, Q) \neq \emptyset$.

Theorem 4.4 guarantees correctness of the transformed program, i.e., if the transformed program detects that an assertion is false, it is actually false.

5 Compile-Time Checking

In this section we present some techniques which allow in certain cases determining at compile-time the results of run-time assertion checking. With this aim, we assume the existence of a global analyzer, typically based on abstract interpretation [11] which is capable of computing at compile-time certain characteristics of the run-time behavior of the program. In particular, we consider the case in which the analysis provides safe approximations of the calling and success patterns for predicates.

5.1 Abstract Interpretation

Abstract interpretation [11] is a technique for static program analysis in which execution of the program is simulated on an *abstract domain* (D_α) which is simpler than the actual, *concrete domain* (D). For this study, abstract interpretation is restricted to complete lattices over sets (i.e., power domains, in general) both for the concrete $\langle D, \subseteq\rangle$ and abstract $\langle D_\alpha, \sqsubseteq\rangle$ domains. As usual, the concrete and abstract domains are related via a pair of monotonic mappings *abstraction* $\alpha : D \mapsto D_\alpha$, and *concretization* $\gamma : D_\alpha \mapsto D$, such that

$$\forall x \in D : \ \gamma(\alpha(x)) \supseteq x \quad \text{and} \quad \forall y \in D_\alpha : \ \alpha(\gamma(y)) = y.$$

In general $\sqsubseteq$ is induced by $\subseteq$ and α (in such a way that $\forall \lambda, \lambda' \in D_\alpha : \ \lambda \sqsubseteq \lambda' \Leftrightarrow \gamma(\lambda) \subseteq \gamma(\lambda')$), and is not equal to set inclusion. The operations of *least upper bound* and *greatest lower bound* in the abstract domain are denoted $\sqcup$ and $\sqcap$ respectively. Also, as usual in abstract interpretation, $\bot$ denotes the abstract substitution such that $\gamma(\bot) = \emptyset$, whereas $\top$ denotes the most general abstract substitution, i.e., $\gamma(\top) = D$.

Goal dependent abstract interpretation takes as input a program P, an abstract domain D_α, and a description $\mathcal{Q}_\alpha$ of the possible initial queries to the program given as a set of abstract queries. An *abstract query* is a pair (p, λ), where p is a predicate symbol (denoting one of the exported predicates) and λ a restriction of the initial stores for p expressed as an abstract substitution λ in the abstract domain D_α. A set of abstract queries $\mathcal{Q}_\alpha$ represents a set of queries, denoted $\gamma(\mathcal{Q}_\alpha)$, which is defined as $\gamma(\mathcal{Q}_\alpha) = \{(p, \theta) | (p, \lambda) \in \mathcal{Q}_\alpha \wedge \theta \in \gamma(\lambda)\}$. Such an abstract interpretation computes a set of triples $Analysis(P, \mathcal{Q}_\alpha, D_\alpha) = \{\langle p_1, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle p_n, \lambda_n^c, \lambda_n^s\rangle\}$. For each predicate p in a program P we assume that the abstract interpretation based analysis computes a tuple $\langle p, \lambda^c, \lambda^s\rangle$. If p is detected to be dead code then $\lambda^c = \bot$. We now provide a couple of definitions which will be used below for stating correctness of abstract interpretation.

Definition 5.1 [Calling Context] Consider a program P, a predicate p and a set of queries $\mathcal{Q}$. The *calling context* of p for P and $\mathcal{Q}$ is $C(p, P, \mathcal{Q}) = \{\ \bar{\exists}_{vars(p)}\theta |\ \exists D \in derivations(P, \mathcal{Q})$ with $current\_store(D) = \langle p :: G \,|\, \theta\rangle\ \}$.

Definition 5.2 [Success Context] Consider a program P, a predicate p, a constraint store θ, and a set of queries $\mathcal{Q}$. The *success context* of p and θ for P and $\mathcal{Q}$ is $S(p, \theta, P, \mathcal{Q}) = \{\ \bar{\exists}_{vars(p)}\theta' |\ \exists D \in derivations(P, \mathcal{Q})$ with $D = \cdots \rightsquigarrow_P \langle p :: G \,|\, \theta\rangle \rightsquigarrow_P \cdots \rightsquigarrow_P \langle G \,|\, \theta'\rangle\}$.

We can restrict the constraints in the calling and success contexts to the variables in p since this does not affect the behavior of calls and success assertions. Correctness of abstract interpretation guarantees that $\gamma(\lambda^c) \supseteq C(p, P, \gamma(\mathcal{Q}_\alpha))$ and $\gamma(\lambda^s) \supseteq \bigcup_{\theta \in \gamma(\lambda^c)} S(p, \theta, P, \gamma(\mathcal{Q}_\alpha))$. In order to ensure correctness of compile-time checking for a set of queries $\mathcal{Q}$, the analyzer must be provided with a suitable $\mathcal{Q}_\alpha$ such that $\gamma(\mathcal{Q}_\alpha)) \supseteq \mathcal{Q}$. In our implementation of the framework, $\mathcal{Q}_\alpha$ is expressed by means of *entry* declarations [25].

5.2 Exploiting Information from Abstract Interpretation

Before presenting the actual sufficient conditions that we propose for performing compile-time checking of assertions, we present some definitions and results which will then be instrumental.

Definition 5.3 [Trivial Success Set] Given a literal L and a program P we define the *trivial success set* of L in P as

$$TS(L, P) = \{\bar{\exists}_{vars(L)}\theta\ | \theta \Rightarrow_P L\}$$

This definition is an adaptation of that presented in [27], where analysis information is used to optimize automatically parallelized programs.

Definition 5.4 [Abstract Trivial Success Subset] An abstract substitution $\lambda^-_{TS(L,P)}$ is an *abstract trivial success subset* of L in P iff $\gamma(\lambda^-_{TS(L,P)}) \subseteq TS(L, P)$.

Lemma 5.5 Let λ be an abstract substitution and let $\lambda^-_{TS(L,P)}$ be an abstract trivial success subset of L in P.

1. if $\lambda \sqsubseteq \lambda^-_{TS(L,P)}$ then $\forall\ \theta \in \gamma(\lambda) : \theta \Rightarrow_P L$
2. if $\lambda \sqcap \lambda^-_{TS(L,P)} \neq \bot$ then $\exists\ \theta \in \gamma(\lambda) : \theta \Rightarrow_P L$

Definition 5.6 [Abstract Trivial Success Superset] An abstract substitution $\lambda^+_{TS(L,P)}$ is an *abstract trivial success superset* of L in P iff $\gamma(\lambda^+_{TS(L,P)}) \supseteq TS(L, P)$.

Lemma 5.7 Let λ be an abstract substitution and let $\lambda^+_{TS(L,P)}$ be an abstract trivial success superset of L in P.

1. if $\lambda^+_{TS(L,P)} \sqsubseteq \lambda$ then $\forall\ \theta$: if $\theta \Rightarrow_P L$ then $\theta \in \gamma(\lambda)$.
2. if $\lambda \sqcap \lambda^+_{TS(L,P)} = \bot$ then $\forall\ \theta \in \gamma(\lambda) : \theta \not\Rightarrow_P L$

In order to apply Lemmas 5.5 and 5.7 effectively, accurate $\lambda^+_{TS(L,P)}$ and $\lambda^-_{TS(L,P)}$ are required. Finding a correct, and hopefully accurate $\lambda^+_{TS(L,P)}$ can simply be done by analyzing the program with the set of abstract queries

$\mathcal{Q}_\alpha = \{(L, \top)\}$. Since our analysis is goal-dependent, the initial abstract substitution $\top$ 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 $\langle L, \top, \lambda^s \rangle$ and thus we can take $\lambda^+_{TS(L,P)} = \lambda^s$, as correctness of the analysis guarantees that λ^s 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 "builtin 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., $\gamma(\lambda^s) = \bigcup_{\theta \in \gamma(\lambda^c)} S(p, \theta, P, \mathcal{Q})$ rather than $\gamma(\lambda^s) \supseteq \bigcup_{\theta \in \gamma(\lambda^c)} S(p, \theta, P, \mathcal{Q})$. In this situation we can use λ^s 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 λ_L, i.e. $TS(L, P) = \gamma(\lambda_L)$, then analysis can often be precise enough to compute $\langle L, \lambda^c, \lambda^s \rangle$ with $\lambda^s = \lambda_L$. 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 domain. 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, $\mathcal{Q}$ a set of queries, and let $\mathcal{Q}_\alpha$ be s.t. $\gamma(\mathcal{Q}_\alpha) \supseteq \mathcal{Q}$. Assume that $\langle p, \lambda^c, \lambda^s \rangle \in Analysis(P, \mathcal{Q}_\alpha, D_\alpha)$. If $\lambda^c \sqsubseteq \lambda^-_{TS(Precond,P)}$ then A is checked in P for $\mathcal{Q}$.

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, $\mathcal{Q}$ a set of queries, and let $\mathcal{Q}_\alpha$ be s.t. $\gamma(\mathcal{Q}_\alpha) \supseteq \mathcal{Q}$. Assume that $\langle p, \lambda^c, \lambda^s \rangle \in Analysis(P, \mathcal{Q}_\alpha, D_\alpha)$. If

1. $\lambda^c \sqcap \lambda^+_{TS(Pre,P)} = \perp$, or
2. $\lambda^s \sqsubseteq \lambda^-_{TS(Post,P)}$

then A is checked in P for $\mathcal{Q}$.

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 $D \in derivations(P, \mathcal{Q})$, i.e., without having to actually compute $derivations(P, \mathcal{Q})$. Unfortunately, this is a bit trickier than it may seem at first sight if analysis over-approximates computation states, as is the usual case.

Theorem 5.10 [False Calls Assertion] Assuming the premises of Theorem 5.8, if $C(p, P, \mathcal{Q}) \neq \emptyset$ and $\lambda^c \sqcap \lambda^+_{TS(Precond,P)} = \perp$ then A is false in P for $\mathcal{Q}$.

In order to prove that a calls assertion is false it is not enough to prove that $\lambda^+_{TS(Precond,P)} \sqsubset \lambda^c$ 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 λ^c and $\lambda^+_{TS(Precond,P)}$ are incompatible, it may be the case that there are no calls for predicate P in $derivations(P, \mathcal{Q})$ (and analysis is not capable of detecting so). This is why the condition $C(p, P, \mathcal{Q}) \neq \emptyset$ is also required.

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

1. $\lambda^c \sqcap \lambda^-_{TS(Pre,P)} \neq \perp$, and
2. $\lambda^s \sqcap \lambda^+_{TS(Post,P)} = \perp$ and $\exists\ \theta \in \gamma(\lambda^c \sqcap \lambda^-_{TS(Pre,P)})$: $S(p, \theta, P, \mathcal{Q}) \neq \emptyset$.

then A is false in P for $\mathcal{Q}$.

Now again, λ^s is an over-approximation, and in particular it can approximate the empty set. This is why the extra condition $\exists\ \theta \in \gamma(\lambda^c \sqcap \lambda^-_{TS(Pre,P)})$: $S(p, \theta, P, \mathcal{Q}) \neq \emptyset$ is required.

If an assertion A is *false* then the program is not correct w.r.t. A. Detecting the mininal 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 $calls(p, Precond)$ can never be found to be true, as the calling context of 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 $Precond$ is a tautology.

Theorem 5.12 [True Success Assertion] Assuming the premises of Theorem 5.9, if

1. $\lambda^+_{TS(Pre,P)} \sqsubseteq \lambda^c$, and
2. $\lambda^s \sqsubseteq \lambda^-_{TS(Post,P)}$

then A is true in P.

Condition 1 guarantees that λ^s describes any store which is a descendent of a calling state of p which satisfied the precondition. Condition 2 ensures that any store described by λ^s 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.

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

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., [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: `ciaopp` [18], the CIAO system preprocessor, developed by UPM, and `chipre` [7], an assertion-based type inferencing and checking tool also developed at UPM in collaboration with Pawel Pietrzak from the U. of Linköping. The analysis used is an adaptation to CLP(fd) of the regular approximation approach of [16]. `chipre` has been interfaced by Cosytec with the CHIP system (adding a graphical user interface) and is currently under industrial evaluation.

CIAO[2] 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.

`ciaopp`, the CIAO precompiler, can perform a number tasks, including: (a) Inference of properties of program predicates and literals, including types (using [16]), modes and other variable instantiation properties (using the CLP version of the PLAI abstract interpreter [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 [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 [19] for LP/CLP programs based on program assertions and machine-readable comments. Generates manuals in many formats including postscript, pdf, info, HTML, etc.

[2]The CIAO system and related tools are available from `http://www.clip.dia.fi.upm.es/Software`.

Acknowledgments

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

References

[1] A. Aggoun, F. Benhamou, F. Bueno, M. Carro, P. Deransart, W. Drabent, G. Ferrand, F. Goualard, M. Hermenegildo, C. Lai, J.Lloyd, J. Maluszynski, G. Puebla, and A. Tessier. CP Debugging Tools: Clarification of Functionalities and Selection of the Tools. Technical Report D.WP1.1.M1.1-2, DISCIPL Project, June 1997.

[2] 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.

[3] K. R. Apt and D. Pedreschi. Reasoning about termination of pure PROLOG programs. *Information and Computation*, 1(106):109–157, 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, Linkoping, Sweden, May 1997. U. of Linkoping Press.

[5] 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.

[6] 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, Linkoping, Sweden, May 1997. U. of Linkoping Press.

[7] F. Bueno, P. López, G. Puebla, M. Hermenegildo, and P. Pietrzak. The CHIP Assertion Preprocessor. Technical

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 DISCIPL.

[8] 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.

[9] 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.

[10] M. Comini, G. Levi, and G. Vitiello. Abstract debugging of logic programs. In L. Fribourg and F. Turini, editors, *Proc. Logic Program Synthesis and Transformation and Metaprogramming in Logic 1994*, volume 883 of *Lecture Notes in Computer Science*, pages 440–450, Berlin, 1994. Springer-Verlag.

[11] 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.

[12] P. Deransart. Proof methods of declarative properties of definite programs. *Theoretical Computer Science*, 118:99–166, 1993.

[13] 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.

[14] 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.

[15] G. Ferrand. Error diagnosis in logic programming. *J. Logic Programming*, 4:177–198, 1987.

[16] J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In Pascal Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613. The MIT Press, 1994.

[17] 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, 1996.

[18] 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.

[19] M. Hermenegildo and The CLIP Group. An Automatic Documentation Generator for (C)LP – Reference Manual. The CIAO System Documentation Series – TR CLIP5/97.1, Facultad de Informática, UPM, August 1997.

[20] M. Hermenegildo and The CLIP Group. Programming with Global Analysis. In *Proceedings of ILPS'97*, pages 49–52, Cambridge, MA, October 1997. MIT Press. (abstract of invited talk).

[21] P. Hill and J. Lloyd. *The Goedel Programming Language*. MIT Press, Cambridge MA, 1994.

[22] J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[23] D. Le Métayer. Proving properties of programs defined over recursive data structures. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–99, 1995.

[24] 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.

[25] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. In *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming*, October 1997.

[26] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. Technical Report CLIP2/97.1, Facultad de Informática, UPM, July 1997.

[27] G. Puebla and M. Hermenegildo. Abstract Specialization and its Application to Program Parallelization. In J. Gallagher, editor, *VI International Workshop on Logic Program Synthesis and Transformation*, number 1207 in LNCS, pages 169–186. Springer-Verlag, 1997.

[28] G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *Journal of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 1999. In press.

[29] 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.

[30] E. Vetillard. *Utilisation de Declarations en Programmation Logique avec Constraintes*. PhD thesis, U. of Aix-Marseilles II, 1994.