A Framework for Assertion-based Debugging in Constraint Logic Programming

Germán Puebla, *Francisco Bueno*, and *Manuel Hermenegildo*

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

Abstract. We propose a general framework for assertion-based debugging of constraint logic programs. Assertions are linguistic constructions which allow expressing properties of programs. We define assertion schemas which allow writing (partial) specifications for constraint logic programs using quite general properties, including user-defined programs. The framework is aimed at detecting deviations of the program behavior (symptoms) with respect to the given assertions, either at compile-time or run-time. We provide techniques for using information from global analysis both to detect at compile-time assertions which do not hold in at least one of the possible executions (i.e., static symptoms) and assertions which hold for all possible executions (i.e., statically proved assertions). We also provide program transformations which introduce tests in the program for checking at run-time those assertions whose status cannot be determined at compile-time. Both the static and the dynamic checking are provably safe in the sense that all errors flagged are definite violations of the specifications. Finally, we report on an implemented instance of the assertion language and framework.

1 Introduction

As (constraint) logic programming (CLP) systems [19] 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, 12, 13, 22]. Classical examples of assertions are the type declarations used in languages such as Gödel [18] or Mercury [25] (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, 10, 17]. 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 determined to be true or false statically.

Our approach is strongly motivated by the availability of powerful and mature static analyzers for (constraint) logic programs (see, e.g., [5, 7, 15, 16, 21] and their references), generally based on abstract interpretation [10]. These systems can statically infer a wide range of properties (from types to determinacy or termination) accurately and efficiently, for realistic programs.

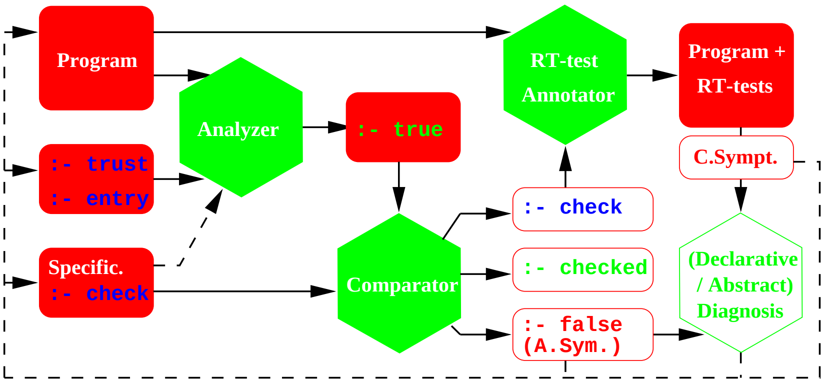

Fig. 1. A Combined Framework for Program Development and Debugging

Thus, we would like to take advantage of standard program analysis tools, rather than developing new abstract procedures, such as concrete [4, 12, 13] or abstract [8, 9] diagnosers and debuggers, or using traditional proof-based methods [2, 3, 11, 14, 26].

Figure 1 presents the general architecture of the type of debugging environment that we propose.[1] Hexagons represent the different tools involved and arrows indicate the communication paths among such tools. It is a design decision of the framework implementation that most of such communication be performed in terms of assertions, and that, rather than having different languages for each tool, the same assertion language be used for all of them (due to space limitations, we cannot present the assertion language itself – see [22] for details). This facilitates communication among the different tools, enables easy reuse of information (i.e., once a property has been stated there is no need to repeat it for the different tools), and makes such communication understandable for the user. Note that not all tools need to be capable of dealing with all properties expressible in the assertion language. Rather, each tool will only make use of the part of the information given as assertions which the tool "understands."

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.[2] 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., [16, 15]– 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.

[1] The implementation (to be described later) includes also other techniques, such as traditional procedural debugging and visualization, which are however beyond the scope of the work presented in this paper.

[2] 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, 23].

Given this overall design, in the rest of the paper we first define formally a series of assertions and the notions of correctness and errors of a program with respect to those assertions. We then present and prove correct techniques for static and dynamic checking of the assertions. Finally, we report on the implementation of the framework, and present some preliminary performance results.

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 $\bar{\exists}_W \theta$ 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. 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 θ. 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 $(L$:-$B) \in defn_P(L)$.

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 $S_0 \leadsto_P S_1 \leadsto_P \ldots \leadsto_P S_n$ 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 $S_n = \langle G \mid \theta \rangle$ 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 $\langle L \mid \theta \rangle$. 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: $derivations(P, \mathcal{Q}) = \bigcup_{Q} derivations(P, Q)$.

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 $\langle nil \mid \theta \rangle$. The constraint $\bar{\exists}_{vars(S)} \theta$ is an *answer to* S. A finished derivation is *failed* if the last state is not of the form $\langle nil \mid \theta \rangle$.

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 (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.

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 [22].

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

list([]).
list([_|L]):-
        list(L).
```

Fig. 2. An Example Program Annotated with Assertions

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 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 produce an assertion $A = (app_A, val_A)$ given a syntactic object AS, by syntactic manipulation only. 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) $A = (app_A, val_A)$. We use $schema(AS) = (app_{AS}, val_{AS})$ to denote that (app_{AS}, val_{AS}) is the result of the translation of AS.

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

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

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

Clearly, there is no way a calls 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 for any call to the predicate, i.e., the precondition is *true*, or only for calls satisfying certain preconditions.

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

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

Note that, for a given assertion A and derivation D, several states of the form $\langle p :: G \mid \theta' \rangle$ 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. In this section we provide several simple definitions which will be instrumental in the rest of the paper.

Definition 3.3 [Error Set] Given an assertion A, the *error set* of A in 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 [Checked Assertion] An assertion A for program P and set of queries $\mathcal{Q}$, is *checked* iff $E(A, P, \mathcal{Q}) = \emptyset$.

Definition 3.5 [True Assertion] An assertion A for program P, A is *true* iff $\forall Q : E(A, P, Q) = \emptyset$.

Definition 3.6 [False Assertion] An assertion A for program P and set of queries $\mathcal{Q}$, is *false* iff $E(A, P, \mathcal{Q}) \neq \emptyset$

It is clear that given a program P and a set of queries $\mathcal{Q}$, any assertion A is either false or checked. Also, any assertion which is true is also checked.

Definition 3.7 [Partial Correctness] A program P is *partially correct* w.r.t. a set of assertions $\mathcal{A}$ and a class $\mathcal{Q}$ of queries iff $\bigcup_{\mathcal{A}} E(A, P, \mathcal{Q}) = \emptyset$.

Our goal is to prove that a program is partially correct w.r.t. a set of assertions when it indeed is, and to detect the assertions which are false otherwise. 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 Sections 4 and 5. 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 6.

4 Run-Time Checking of Assertions

The main idea behind run-time checking of assertions is, given a program P and a set of assertions $\mathcal{A}$, to directly apply Definitions 3.4 and 3.6 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 true as this would require checking the derivations from all possible queries (to any predicate) which is in general an infinite set and thus checking may not terminate.

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.

We now provide an operational semantics which checks whether assertions hold or not while computing the derivations from a query. A *check literal* is a syntactic object $check(L, A)$ where L is either an atom or a constraint and A (an identifier for) the assertion which generated the check literal. In this semantics, a *literal* is now an atom, a constraint, or a check literal. A CLP program with assertions is a pair $(P, \mathcal{A})$, where P is a program, as defined in Section 2 and $\mathcal{A}$ is a set of assertions.[3]

A finished derivation for a query Q in a CLP program P may be either successful (with answer θ) or failed. In the case of programs with assertions, we consider a third case for finished derivations which we refer to as *erroneous*. We introduce a class of distinguished states of the form (ϵ, A) which cannot be further reduced. A finished derivation is erroneous if the last state in the derivation is of the form (ϵ, A), where A is (an identifier for) an assertion. Erroneous derivations indicate that the assertion A has been violated.

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,
   - if $\exists\ A = calls(p, Cond) \in \mathcal{A}$ s.t. $\theta \not\Rightarrow_P Cond$, then it is reduced to $\langle \epsilon \,\|\, A\rangle$.
   - otherwise if $\exists (L\text{:-}B) \in defn_P(L)$ it is reduced to $\langle B :: PostCond :: G \,\|\, \theta\rangle$ where $PostCond = \{check(S, A) |\ \exists\ A = success(L, C, S) \in \mathcal{A} \wedge \theta \Rightarrow_P C\}$.
3. If L is a check literal $check(prop, A)$,
   - if $\theta \Rightarrow_P prop$ then it is reduced to $\langle G \,\|\, \theta\rangle$
   - otherwise it is reduced to $\langle \epsilon \,\|\, A\rangle$.

Note that the relative order of the check literals in $PostCond$ is not fixed in the semantics. However, this order is irrelevant as they may be checked in any order. We will write derivations using the operational semantics for programs with assertions as $\langle G \,\|\, \theta\rangle \rightsquigarrow_{(P,\mathcal{A})} \cdots \rightsquigarrow_{(P,\mathcal{A})} \langle G' \,\|\, \theta'\rangle$ in order to distinguish them from derivations using the operational semantics of Section 2. Also, the set of derivations from a set of queries $\mathcal{Q}$ in a program P using the semantics with assertions is denoted $derivations_{\mathcal{A}}(P, \mathcal{Q})$.

Theorem 4.4 [Run-time Checking] Given a program P, a set of assertions $\mathcal{A}$, and a set of queries $\mathcal{Q}$,

$$A \text{ is false iff } \exists\ D \in derivations_{\mathcal{A}}(P, \mathcal{Q}) \text{ with } current\_state(D) = \langle \epsilon, A\rangle$$

Theorem 4.4 guarantees that we can use the proposed operational semantics for programs with assertions in order to detect violation of assertions.

Corollary 4.5 Given a program P, a set of assertions $\mathcal{A}$, and a set of queries $\mathcal{Q}$, A is checked iff $\forall\ D \in derivations_{\mathcal{A}}(P, \mathcal{Q}) : D$ is not erroneous.

[3] Program point assertions can be introduced by just allowing check literals to appear in the body of rules [22]. However, for simplicity we do not discuss program point assertions in this paper.

Corollary 4.5 is a direct consequence of Theorem 4.4. However, proving $\forall\ D \in$ $derivations_{\mathcal{A}}(P, \mathcal{Q})$: D is not erroneous is often not feasible in practice as in general $\mathcal{Q}$ is infinite. Furthermore, for a single query Q $derivations_{\mathcal{A}}(P, Q)$ may also be infinite. The approach usually taken in practice is to take a finite $\mathcal{Q}$'$\subset$ $\mathcal{Q}$ (the test set) which is considered to be representative of $\mathcal{Q}$. Then, (a subset of) $derivations(P, \mathcal{Q}')$ is computed. If an erroneous derivation is found, diagnosis is started. Otherwise, P is (unsafely) assumed to be correct w.r.t $\mathcal{A}$ though it has not actually been proved, or more testing is performed. Furthermore, this semantics can also be used to obtain answers to the original query, as stated by Theorem 4.6 below.

Theorem 4.6 Let P be a program, $\mathcal{A}$ a set of assertions, and $\mathcal{Q}$ set of queries. If P is partially correct w.r.t. $\mathcal{A}$then $derivations(P, Q) = derivations_{\mathcal{A}}(P, Q)$.

5 Run-Time Checking with Existing CLP Systems

Even though the semantics for programs with assertions presented in the previous section can be used to perform run-time checking, an important disadvantage is that existing CLP system do not implement such semantics. Modification of a CLP system with that aim is not a trivial task due to the complexity of typical implementations. Thus, it seems desirable to be able to perform run-time checking on top of existing systems without having to modify them. Writing a meta-interpreter which implements this semantics on top of a CLP system is not a difficult task. However, the drawback of this approach is its inefficiency due to the overhead introduced by the meta-interpretation level.[4] A second approach, which is the one used in our implementation, is based on program transformation. Given a program P, another program P' is obtained which checks the assertions while running on a standard CLP system. The meta-interpretation level is eliminated since the process of assertion checking is compiled into P'.

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

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

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 5.1 [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 5.1 guarantees correctness of the transformed program, i.e., if the transformed program detects that an assertion is false, it is actually false.

[4] Partial evaluation may be used to reduce such overhead for those parts of the program in which no assertion is to be checked.

6 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 [10] 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. Note that it is not to be expected that all assertions are checkable at compile-time, either because the properties in the assertions are not decidable at compile-time or because the available analyzers are not accurate enough. Those which cannot be checked at compile-time should, in general, be checked at run-time.

6.1 Abstract Interpretation

Abstract interpretation [10] 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, $\perp$ denotes the abstract substitution such that $\gamma(\perp) = \emptyset$.

Definition 6.1 [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 $C(p, P, \mathcal{Q}) = \{ \ \bar{\exists}_{vars(p)}\theta | \ \exists D \in \mathit{derivations}(P, \mathcal{Q})$ with $\mathit{current\_store}(D) = \langle p :: G \,|\, \theta \rangle \ \}$.

Definition 6.2 [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 $S(p, \theta, P, \mathcal{Q}) = \{ \ \bar{\exists}_{vars(p)}\theta' |$ $\exists D \in \mathit{derivations}(P, \mathcal{Q})$ with $D = \cdots \leadsto_P \langle p :: G \,|\, \theta \rangle \leadsto_P \cdots \leadsto_P \langle G \,|\, \theta' \rangle$.

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.

Goal dependent abstract interpretation takes as input a program P, a set $\mathcal{Q}_\alpha$ of pairs (p_j, λ_j), where p_j is a predicate symbol (denoting one of the exported predicates) and λ_j a restriction of the initial stores for p expressed as an abstract substitution λ in the abstract domain D_α, and which represents the set of concrete queries $\mathcal{Q} = \gamma(Q_\alpha)$. Such an abstract interpretation computes a set of triples $\mathit{Analysis}(P, \mathcal{Q}_\alpha, D_\alpha) = \{\langle p_1, \lambda_1^c, \lambda_1^s \rangle, \ldots, \langle p_n, \lambda_n^c, \lambda_n^s \rangle\}$.

For each predicate p in a program P not detected to be dead code, we assume that the abstract interpretation based analysis computes a tuple $\langle p, \lambda^c, \lambda^s \rangle$. Correctness of abstract interpretation guarantees that $\gamma(\lambda^c) \supseteq C(p, P, \mathcal{Q})$ and $\gamma(\lambda^s) \supseteq \bigcup_{\theta \in \gamma(\lambda^c)} S(p, \theta, P, \mathcal{Q})$.

6.2 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 6.3 [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 [24], where analysis information is used to optimize automatically parallelized programs.

Definition 6.4 [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 6.5 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 $\lambda \sqcap \lambda^-_{TS(L,P)} \neq \perp$ then $\exists\, \theta \in \gamma(\lambda) : \theta \Rightarrow_P L$

Definition 6.6 [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 6.7 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 $\lambda \sqcap \lambda^+_{TS(L,P)} = \perp$ then $\forall\, \theta \in \gamma(\lambda) : \theta \not\Rightarrow_P L$

In order to apply Lemmas 6.5 and 6.7 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 literal L and taking $\lambda^+_{TS(L,P)} = \lambda^s$ if the analysis information for L is $\langle L, \lambda^c, \lambda^s \rangle$. 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 [24], 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). In our implementation of the framework the user can choose whether to use type definitions or regular programs for defining tests. In the first case, the corresponding regular program is automatically generated if run-time checking is to be performed. Unfortunately, in general there is no such straightforward mapping from abstract substitutions to programs for a given arbitrary abstract domain.

6.3 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 6.8 [Checked Calls Assertion] Let $calls(p, Precond)$ be an assertion, P a program, and $\langle p, \lambda^c, \lambda^s \rangle$ the analysis information for p and a class of queries $\mathcal{Q}$. If $\lambda^c \sqsubseteq \lambda^-_{TS(Precond,P)}$ then A is checked.

Theorem 6.9 [Checked Success Assertion] Let $success(p, Pre, Post)$ be an assertion, and P a program. Let $\langle p, \lambda^c, \lambda^s \rangle$ be the analysis information for p and a class of queries $\mathcal{Q}$. If

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

then A is checked.

Theorem 6.9 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.

6.4 True Assertions

As with checked assertions, if an assertion is true then it is guaranteed that it will not raise any error at run-time. Thus, there is no need to consider it when performing run-time checking. However, there is an important difference between them. Assertions which are checked will not raise errors for the considered (class of) queries, but may not hold for other queries. True assertions hold for any possible query and thus can be used as a (goal-independent) property of the program, regardless of the query. 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.

Note that, due to the definition of true assertions, 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 6.10 [True Success Assertion] Let $success(p, Pre, Post)$ be an assertion, and P a program. Let $\langle p, \lambda^c, \lambda^s \rangle$ be the analysis information for p and a class of queries $\mathcal{Q}$. If

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

then A is true.

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 originated from a calling state which satisfied the precondition satisfies the postcondition.

6.5 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 6.11 [False Calls Assertion] Let $calls(p, Precond)$ be an assertion, and P a program. Let $\langle p, \lambda^c, \lambda^s \rangle$ be the analysis information for p and a class of queries $\mathcal{Q}$. If $C(p, P, \mathcal{Q}) \neq \emptyset$ and $\lambda^c \sqcap \lambda^+_{TS(Precond,P)} = \perp$ then A is false.

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.

		Types			*Modes*			*Aliasing*		
Prog	**Ps**	**Props**	**Infer**	**Simp**	**Props**	**Infer**	**Simp**	**Props**	**Infer**	**Simp**
ann	66	514	9.64	0.55	265	1.60	1.22	419	2.22	6.57
palin	6	28	0.56	0.19	15	0.18	0.02	22	0.21	0.02
progeom	10	58	0.70	0.65	56	0.08	0.06	57	0.06	0.06
queen	6	28	0.23	0.09	26	0.05	0.03	28	0.04	0.04
warplan	31	132	8.33	0.12	71	1.83	0.07	98	2.35	0.10

Fig. 3. Analysis/Checking Performance

Theorem 6.12 [False Success Assertion] Let $success(p, Pre, Post)$ be an assertion, and P a program. Let $\langle p, \lambda^c, \lambda^s\rangle$ be the analysis information for p and a class of queries $\mathcal{Q}$. 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.

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. However, such static diagnosis is out of the scope of this paper.

7 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 can be incorporated in a straightforward way. Currently, two different experimental debugging environments have been developed using this framework: `ciaopp`, the CIAO system preprocessor, developed by UPM, and `fdtypes`, an assertion-based type inferencing and checking tool developed by Pawel Pietrzak at the U. of Linköping, in collaboration with UPM. Also, an assertion-based preprocessor for PrologIV has been developed by Claude Lai of PrologIA extending the work of [26], which is based on the same overall design, but separately coded and using simpler analysis techniques. These three environments share the same source language (ISO-Prolog + finite domain constraints) and the same assertion language [23], so that source and output programs (annotated with assertions and/or run-time tests) can be easily exchanged. `fdtypes` has been interfaced by Cosytec with the CHIP system (adding a graphical user interface) and is currently under industrial evaluation.

`ciaopp` uses as analyzers both the CLP version of the PLAI abstract interpreter [16] and adaptations of Gallagher's type analysis [15], and works on the domains of moded types, definiteness, freeness, and grounding dependencies (as well as more complex properties, such as bounds on cost or non-failure for Prolog programs). This tool is currently an integral part of the CIAO system.

The actual evaluation of the practical benefits of these tools is beyond the scope of this paper, but we believe that the significant industrial interest shown is encouraging. Also, it has certainly been observed during use by the system developers and a few early users that these tools can indeed detect some bugs much earlier in the program development process than with any previously available tools. Interestingly, this has been observed even when no specifications are available from the user: in these systems the system developers have included a rich set of assertions inside library modules (such as those defining the system built-ins and standard libraries) for the predicates defined in these modules. As a result, symptoms in user programs are often flagged during compilation simply because the analyzer/comparator pair detects that assertions for the system library predicates are violated by program predicates.

It is also not our current purpose to perform a detailed evaluation of the performance of these systems. However, preliminary results also show that the performance is quite reasonable. Figure 3 presents results for `ciaopp`, inferring *types* (using Gallagher's type analyzer [15]), *modes* (using a variant of the Sharing+Freeness domain [20]), and *variable aliasing* (using the

	With Run-time Checks					
	Types		*Modes*		*Aliasing*	
Prog	**Props**	**Slowdown**	**Props**	**Slowdown**	**Props**	**Slowdown**
ann	514	2.95	265	3.55	419	3.50
palin	28	15.0	15	6.00	22	9.00
progeom	58	104	56	65.0	57	66.0
queen	28	6.10	26	6.10	28	6.10
warplan	132	190	71	151	98	177

Fig. 4. Run-Time Checking Cost

standard Sharing+Freeness). Analysis times are relatively well understood for these domains. The assertion processing time (normalization, simplification, etc.) obviously depends on the number of assertions in the input program. Given the lack at this point of a standardized set of benchmarks including assertions, for our preliminary evaluation we have opted for a simple and repeatable method of generating assertions automatically: previous to our measurements, we have run the analyzer on the program, producing `true` assertions which express the analysis results, rewritten such assertions into `check` assertions, and used this program as input for the system. **Prog** is the program being debugged and **Ps** the number of predicates, and, thus, of assertions (analysis variants were collapsed into one per predicate) in the program. **Props** is the number of properties which appear in the program assertions. **Infer** the analysis time, and **Simp** the time taken by the comparator to simplify the input assertions. These times are relative to the time taken by the standard (SICStus-)Prolog compiler to compile the program without assertions. For example, a 2 for **Infer** means that analysis time is twice the normal Prolog compiler time for the benchmark.

Clearly, in our case all assertions should be proven to be checked statically (and, indeed `ciaopp` does so). Figure 4 provides some data on the run-time cost of the assertions eliminated. It shows the slowdowns incurred when running the programs with the assertions relative to the running times of the original programs without assertions. **Prog** and **Props** are as before. Obviously, in our stylized case, when running the programs with assertions through `ciaopp` no slowdowns occur, since all run-time checks are eliminated.

Again, the purpose of presenting these results is just to give a flavor for the behavior of the system. Clearly, the results should be contrasted with those obtained in an exhaustive evaluation, using more realistic, user provided assertions, which is left as future work.

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 assertion-based debugging, to Pawel Pietrzak for his important contribution in the adaptation of the John Gallagher's type analysis for CLP($\mathcal{FD}$), and to Abder Aggoun, Helmut Simonis, Eric Vetillard and Claude Lai for their feedback on the assertion language design.

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. 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.
8. 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.
9. 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.
10. 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.
11. P. Deransart. Proof methods of declarative properties of definite programs. *Theoretical Computer Science*, 118:99–166, 1993.
12. 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.
13. 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.
14. G. Ferrand. Error diagnosis in logic programming. *J. Logic Programming*, 4:177–198, 1987.
15. 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.
16. 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.
17. M. Hermenegildo and the CLIP Group. Programming with Global Analysis. In *Proceedings of ILPS'97*. MIT Press, October 1997. (abstract of invited talk).
18. P. Hill and J. Lloyd. *The Goedel Programming Language*. MIT Press, Cambridge MA, 1994.
19. J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
20. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.
21. 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. Originally published as Technical Report FIM 59.1/IA/90, Computer Science Dept, Universidad Politecnica de Madrid, Spain, August 1990.
22. 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.
23. 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.
24. 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.
25. 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.
26. E. Vetillard. *Utilisation de Declarations en Programmation Logique avec Constraintes*. PhD thesis, U. of Aix-Marseilles II, 1994.