Combined Static and Dynamic Assertion-based Debugging of Constraint Logic Programs[⋆]

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 for expressing properties of programs. We define several assertion schemas for 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 (i.e., statically) or run-time (i.e., dynamically). 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 briefly on the currently implemented instances of the generic framework.

1 Introduction

As (constraint) logic programming (CLP) systems [23] 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, [12] 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, 3, 13, 14, 24, 27]. Classical examples of assertions are the type declarations used in languages such as Gödel [22] 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

[⋆] This work has been supported in part by projects ESPRIT LTR #22532 DiSCiPl and CICYT TIC99-1151 EDIPIA. The authors would also like to thank Jan Małuszyński, Wlodek Drabent and Pierre Deransart for many interesting discussions on assertions.

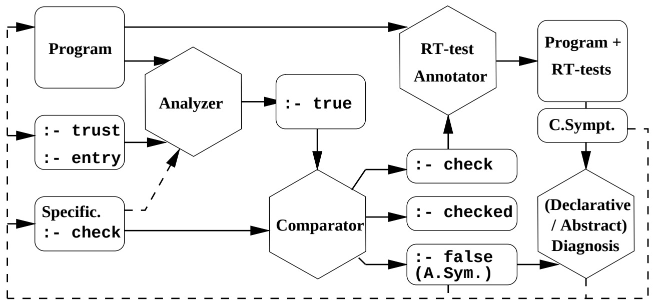

Fig. 1. A Combined Framework for Program Development and Debugging

nature, including properties which are *undecidable*, and, on the other hand, 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 decidable.

Consequently, the proposed framework needs to deal throughout with *approximations* [6,10,20,19]. 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 specification. However, while the system can be complete with respect to decidable properties (e.g., certain type systems), it cannot be complete in general, in the sense that when undecidable properties are used in assertions, there may be errors 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 are (optionally) 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., [5,7,16,17,25] 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. Thus, we would like to take advantage of standard program analysis tools, rather than developing new abstract procedures, such as concrete [3,13,14] or abstract [8,9] diagnosers and debuggers, or using traditional proof-based methods [1,2,11,15,30].

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. This facilitates

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

communication among the different tools, enables easy reuse of information, and makes such communication understandable for the user.

Assertions are also used to write a (partial) specification of the (possibly partially developed) program. 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 [27], with a prefix denoting their status (*check*, *trust*, ...). The program analyzer generates an approximation of the actual semantics of the program, expressed using assertions with the flag *true* (in the case of CLP programs standard analysis techniques –e.g., [17, 16]– are used for this purpose). The comparator, using the abstract operations of the analyzer, 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). 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 assertions, some assertion schemas, and the notions of correctness and errors of a program with respect to those assertions. We then present techniques for static and dynamic checking of the assertions. This paper is complementary to other more informal ones in which we present the framework from a more application-oriented perspective. Details on the assertion language at the user-level can be found in [27]. Details on the debugging framework and on the use of assertions within it from a user's perspective can be found in [26] (which also includes a discussion on the practicality of the approach) and in [21] (which also includes a preliminary performance evaluation).

In this paper, after the necessary preliminaries of Section 2, we define formally our notion of assertion and of assertion-based debugging and validation in Section 3, and some particular kinds of assertions in Section 4. We then formalize dynamic and static debugging in sections 5 and 6, respectively. Finally, Section 7 briefly reports on the implemented instances of the proposed framework.

2 Preliminaries and Notation

A *constraint* is essentially a conjunction of expressions built from predefined predicates (such as term equations or inequalities over the reals) whose arguments are constructed using predefined functions (such as real addition). We let

[2] The user may provide additional information by means of "*entry*" assertions (which describe the external calls to a module) and "*trust*" assertions (which provide information that the analyzer can use even if it cannot prove it) [5, 27].

$\bar{\exists}_L\theta$ be the constraint θ restricted to the variables of the syntactic object L. We denote constraint entailment by $\models$, so that $c_1 \models c_2$ denotes that c_1 entails c_2.

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 constraint. A *goal* is a finite sequence of literals. A *rule* is of the form $H\,\text{:-}\,B$ where H, the *head*, is an atom 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, and it facilitates the presentation. However, in the examples (and in the implementation of our framework) we use non-normalized programs.

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 a goal G and a constraint store (or *store* for short) θ. A state $\langle L :: G \mid \theta\rangle$ where L is a literal can be *reduced* as follows:

1. If L is a 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 $(L\,\text{:-}\,B) \in defn_P(L)$.

where :: denotes concatenation of sequences and we assume for simplicity that the underlying constraint solver is complete. We use $S \leadsto_P S'$ to indicate that in program P a reduction can be applied to state S to obtain state S'. Also, $S \leadsto_P^* S'$ indicates that there is a sequence of reduction steps from state S to state S'. 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 $S_n = \langle G \mid \theta\rangle$ 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 system 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*$(P, \mathcal{Q}) = \bigcup_{Q\in\mathcal{Q}}$ *derivations*(P, Q).

The observational behavior of a program is given by its "answers" to queries. A finite derivation from a query (L, θ) for program P is *finished* if the last state in the derivation cannot be reduced. A finished derivation from a query (L, θ) is *successful* if the last state is of the form $\langle nil \mid \theta'\rangle$, where *nil* denotes the empty sequence. The constraint $\bar{\exists}_L\theta'$ is an *answer* to S. We denote by $answers(P, Q)$ the set of answers to query Q. A finished derivation is *failed* if the last state is not of the form $\langle nil \mid \theta\rangle$. Note that *derivations*$(P, \mathcal{Q})$ contains not only finished derivations but also all intermediate derivations from a query. A query Q *finitely fails* in P if *derivations*(P, Q) is finite and contains no successful derivation.

3 Assertions and Program Correctness

We now provide a formal definition of assertions and of correctness of a program w.r.t. a set of assertions. Our definition of assertion is very open. In the next section we will provide several more specific schemas for assertions which correspond to the traditional *pre-* and *post*conditions.

Definition 1 (Condition on Derivations). *Let $\mathcal{D}$ be the set of all derivations. A* condition on derivations *is any boolean function $f : \mathcal{D} \to \{true, false\}$ which is total.*

Definition 2 (Assertion). *An* assertion *A for a program P is a pair (app_A, sat_A) of conditions on derivations.*

Conditions on derivations are boolean functions which are decidable for any derivation. As an intuition, given an assertion A, the role of app_A is to indicate whether A is *applicable* to a derivation D. If it is, then sat_A should take the value *true* on D for the assertion to hold.

Definition 3 (Evaluation of an Assertion on a Derivation). *Given an assertion $A = (app_A, sat_A)$ for program P, the* evaluation of A on a derivation *D, denoted $solve(A, D, P)$, is defined as:*

$$solve(A, D, P) = app_A(D) \to sat_A(D).$$

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, i.e., queries which are "expected". The set of valid queries to the program is represented by $\mathcal{Q}$. We now provide several simple definitions which will be instrumental.

Definition 4 (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 5 (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 6 (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 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.

Definition 7 (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 assertions in that true assertions hold in the program for any set of queries $\mathcal{Q}$.

Definition 8 (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 Assertion Schemas

An *assertion schema* is an expression which, given a syntactic object AS, produces an assertion $A = (app_A, sat_A)$ 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 A. In what follows we use r and $r(O)$ to represent a variable renaming and the result of applying it to some syntactic object O, respectively.

Condition Literals: In the assertion schemas pre- and postconditions will be used. For simplicity, in the formalization (but not in the implementation) pre- and postconditions in assertions are assumed to be literals (rather than for example conjunctions and/or disjunctions of literals).[3] We call such literals *condition literals* and assume that they have a particular meaning associated.

Definition 9 (Meaning of a Literal). *The* meaning of a literal *L, denoted $|L|$ is a set of constraints. If L is a constraint, we define $|L| = \{L\}$. If L is an atom we assume that a definition of the form $|L'| = \{\theta_1, \dots, \theta_n\}$ is given s.t. $L = r(L')$. Then $|L| = r(\{\theta_1, \dots, \theta_n\})$.*

Intuitively, the meaning of a literal contains the "weakest" constraints which make the literal take the value *true*. A constraint θ is *weaker* than another constraint θ' iff $\theta' \models \theta$. We denote by M a set of meanings of literals.

Example 1. Consider defining $|list(A)|=\{A = [], A = [B|C] \wedge list(C)\}$ and $|sorted(A)|=\{A = [], A = [B], A = [B, C|D] \wedge B \leq C \wedge E = [C|D] \wedge sorted(E)\}$.

Definition 10 (Holds Trivially). *A literal L* holds trivially *for θ in M, denoted $\theta \models_M L$ iff $\exists \theta' \in |L|$ s.t. $\theta \models \theta'$ and $\exists c : (c \wedge \theta' \not\models false) \wedge (\theta' \wedge c \models \theta)$.*

[3] However, it is straightforward to lift up this restriction, and in the implementation of the framework conjunctions and disjunctions are indeed allowed.

Example 2. Assume that $\theta = (A = f)$ and $M = \{|list(A)|, |sorted(B)|\}$. Since $\forall \theta' \in |list(A)| : \theta \not\models \theta'$, as we would expect, $\theta \not\models_M list(A)$. Assume now that $\theta = (A = [\_|Xs])$. Though A is compatible with a list, it is not actually a (nil terminated) list. Again in this case $\forall \theta' \in |list(A)| : \theta \not\models \theta'$ and thus again $\theta \not\models_M list(A)$. The intuition behind this is that we cannot guarantee that A is actually a list given θ since a possible instance of A in θ is $A = [\_|f]$, which is clearly not a list. Finally, assume that $\theta = (A = [B] \wedge B = 1)$. In such case $\exists \theta' = (A = [B|C] \wedge C = [])$ s.t. $\theta \models \theta'$ and $\exists c = (B = 1)$ s.t. $(c \wedge \theta' \not\models false) \wedge (\theta' \wedge c \models \theta)$. Thus, in this last case $\theta \models_M list(A)$.

Calls Assertions: This assertion schema is used to describe preconditions for predicates. Given a program P and an expression $AS = calls(p, Precond)$, where p is a normalized atom and $Precond$ a condition literal which refers to the variables of p, we obtain an assertion A for program P whose app_{AS} and sat_{AS} are defined as:

$$app_{calls(p,Precond)}(D) = \begin{cases} true & \text{if } curr\_state(D) = \langle q :: G \mid \theta \rangle \ \wedge \ q = r(p) \\ false & \text{otherwise} \end{cases}$$
$$sat_{calls(p,Precond)}(D) = curr\_store(D) \models_M r(Precond).$$

Clearly, there is no way an assertion $calls(p, Precond)$ can be violated unless the next literal q to be reduced is of the same predicate as p.

Example 3. The procedure `partition(A,B,C,D)` expects a list in `A` to "partition" it into two other lists based on the "pivot" `B`. Thus, the following assertion states that it should be called with `A` a list. It appears in the schema oriented syntax that we use herein, as well as in the program oriented syntax of [27].

```
:- calls partition(A,B,C,D) : list(A).
% { calls( partition(A,B,C,D) , list(A) ) }
```

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 a program P and an expression $AS = success(p, Pre, Post)$, where p is a normalized atom, and both Pre and $Post$ condition literals which refer to the variables of p, we obtain an assertion A for P whose app_{AS} and sat_{AS} are defined as follows:

$$app_{success(p,Pre,Post)}(D) = \begin{cases} true & \text{if } curr\_state(D) = \langle G \mid \theta \rangle \ \wedge \ \exists q \exists \theta' \exists r : \\ \langle q :: G \mid \theta' \rangle \in D \ \wedge \ q = r(p) \ \wedge \ \theta' \models_M r(Pre) & \\ false & \text{otherwise} \end{cases}$$
$$sat_{success(p,Pre,Post)}(D) = \forall\, S = \langle q :: G \mid \theta' \rangle \in D \text{ s.t. } \exists r \ q = r(p) : \\ \theta' \models_M r(Pre) \rightarrow curr\_store(D) \models_M r(Post).$$

Note that, for a given assertion A and derivation D, several states of the form $\langle q :: G \mid \theta' \rangle$ s.t. $q = r(p)$ may exist in D in which the precondition, i.e., *r(Pre)* holds. As a result, the postcondition *r(Post)* will have to be checked several times with different renamings r which relate the variables of p, and thus (some of) those in $Post$, with different states in D.

Example 4. The following assertion states that if a procedure `qsort(A,B)` succeeds when called with `A` a list then `B` should be sorted on success.

```
:- success qsort(A,B) : list(A) => sorted(B).
% { success( qsort(A,B) , list(A) , sorted(B) ) }
```

5 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 5 and 6 in order to determine whether the assertions in $\mathcal{A}$ are checked or false, i.e., obtaining (a subset of) the derivations by running the program and determining whether they belong to the error set of the assertions. It is not to be expected that Definition 6 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 it seems natural to define the meaning of condition literals as CLP programs rather than as (recursive) sets. We thus restrict the admissible conditions of assertions to those literals L_p for which a definition of the corresponding predicate p exists s.t. $answers(P, (L_p, true)) = |L_p|$. We argue that this is not too strong a restriction given the high expressive power of CLP languages.[4] Note that the approach also implies that the program P must contain the definitions of all the predicates p for literals L_p used in conditions of assertions. Thus, from now on we assume the program P contains the definition of M as CLP predicates. 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 they are already familiar with.

Example 5. Consider defining `list(A)` and `sorted(A)` of Example 1 as:

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

Once we have decided to define condition literals in CLP,[5] the next question is how to determine the value of $\theta \models_M L$ using the underlying CLP system. At first sight, one possibility would be to compute $answers(P, (L, \theta))$. Clearly, if such set is empty, $\theta \not\models_M L$. However, $\theta \models_M L$ is not guaranteed to hold if $answers(P, (L, \theta))$ is not empty. This is why we introduce the definition below.

Definition 11 (Succeeds Trivially). *A literal* L succeeds trivially *for* θ *in* P*, denoted* $\theta \Rightarrow_P L$*, iff* $\exists \theta' \in answers(P, (L, \theta))$ *s.t.* $\theta \models \theta'$.

[4] Note that the scheme of [27, 26] allows approximate definitions of such predicates and sufficient conditions for proving and disproving them.

[5] Note that, given a logic expression built using literals, conjunctions, and disjunctions, it is always possible to write such expression as a predicate definition.

Intuitively, a literal L succeeds trivially if L succeeds for θ in P without adding new "relevant" constraints to θ. Note that, if L is a constraint, this means that L was already entailed by θ. For program predicate atoms, it means that θ was constrained enough to make L succeed without adding relevant constraints to θ. This means that we are considering condition literals as *instantiation* checks [27, 21]. They are true iff the variables they check for are at least as constrained as their predicate definition requires. Note that the notion of L succeeding trivially for θ in P corresponds to $\theta \models_M L$.

Lemma 1 (Checking of Condition Literals). *Let L be a condition literal in an assertion for program P. If $answers(P, (L, true)) = |L|$ then for any θ, $\theta \models_M L$ iff $\theta \Rightarrow_P L$.*

Proof. $\theta \models_M L \Rightarrow \theta' \in |L| = answers(P, (L, true)) \wedge \theta \models \theta' \Rightarrow \theta' \in answers(P, (L, \theta')) \wedge \theta \models \theta' \Rightarrow \theta' \in answers(P, (L, \theta)) \wedge \theta \models \theta'$. Conversely, $\theta \Rightarrow_P L \Rightarrow \theta' \in answers(P, (L, \theta)) \wedge \theta \models \theta' \Rightarrow \theta' \models \theta \wedge \theta \models \theta' \Rightarrow \theta \in answers(P, (L, \theta))$ $\Rightarrow \exists \theta'' \in answers(P, (L, true)) = |L| \; \exists c : (c \wedge \theta'' \not\models false) \wedge (\theta'' \wedge c \models \theta)$.

The lemma above allows us to use in a sound way the results of $\theta \Rightarrow_P L$ as the value of $\theta \models_M L$. Unfortunately, from a practical point of view, computing $\theta \Rightarrow_P L$ is problematic, since it may require computing $answers(P, (L, \theta))$, which may not terminate. Thus, unless we introduce some restrictions, run-time checking may introduce non-termination into terminating programs.

Existing CLP systems do compute $derivations(P, Q)$ using some fixed strategy, which induces an ordering on the set of derivations. A strategy is determined by a *search rule* which indicates the order in which the program rules that can be used to reduce an atom should be tried, coupled with a strategy to decide which of the unfinished derivations should be further reduced. The typical search strategy is LIFO, which implies a depth-first search. We denote by SR a search rule together with a search strategy, and by $derivations_{SR}(P, Q)$ the sequence of derivations from Q in P under strategy SR.

Definition 12 ($derivations^1_{SR}$). *Let P be a program and Q a query. We denote by $derivations^1_{SR}(P, Q)$ the prefix of $derivations_{SR}(P, Q)$ s.t.*

1. *If $answers(P, Q) = \emptyset$ then $derivations^1_{SR}(P, Q) = derivations_{SR}(P, Q)$.*
2. *Otherwise, let $derivations_{SR}(P, Q)$ be the sequence $D_1 :: \ldots :: Dn :: DS$ s.t. $\forall i \in \{1, \ldots, n-1\} : D_i$ is not successful and D_n is successful. Then, $derivations^1_{SR}(P, Q) = D_1 :: \ldots :: Dn$.*

Definition 13 (Test). *A literal L is a* test *iff $\forall \theta$:*

1. *$derivations^1_{SR}(P, (L, \theta))$ is finite, and if θ_1 is its answer then*
2. *$\forall \theta' \in answers(P, (L, \theta)) : (\theta_1 \wedge \theta' \models false \vee \theta' \models \theta_1)$.*

Example 6. Literals `sorted(B)` and `list(B)` for the predicates defined in Example 5 are tests, since for every possible initial state the execution of, e.g., `list(B)`, this literal either (1) finitely fails if `B` is constrained to be incompatible with a list, (2) succeeds once without adding "relevant" constraints if `B` is constrained to a list, or (3) has a leftmost successful derivation which constrains `B` to a list in such a way that this constraint is incompatible with the answers of the rest of successful derivations. E.g., for `list(X)` there is an infinite number of answers `X=[]`, `X=[_]`, `X=[_|_]`, ..., but they are pairwise incompatible (they have no common instance).

Theorem 1. *If L is a test then $\forall \theta : \theta \Rightarrow_P L$ iff $\exists\ D \in derivations^1_{SR}(P, (L, \theta))$ s.t. D is successful with answer θ_1 and $\theta \models \theta_1$.*

Proof. Since $\theta_1 \in answers(P, (L, \theta))$, if $\theta \models \theta_1$ then $\theta \Rightarrow_P L$. The converse also holds. We prove it by contradiction. If $\theta \Rightarrow_P L$ then $\exists\ \theta' \in answers(P, (L, \theta))$: $\theta \models \theta'$. Assume θ' is not θ_1 and $\theta \not\models \theta_1$. Then L cannot be a test, which is a contradiction. If L was a test then either (1) $\theta' \wedge \theta_1 \models false$ or (2) $\theta' \models \theta_1$. Since $\theta \models \theta'$ then (1) $\theta \wedge \theta_1 \models \theta' \wedge \theta_1 \models false$, which is impossible, since θ_1 is an answer for initial store θ and therefore compatible with it, or (2) $\theta \models \theta_1$, which is a contradiction.

Theorem 1 guarantees that checking of pre- and postconditions, which are required to be tests, is decidable, since it suffices to check only for the first successful derivation in a (sub)set of derivations (search space) which is finite. I.e., either there is a first answer computed in a finite number of reductions, or there is no answer and the checking finitely fails. In our framework we only admit *tests* as conditions in assertions which are going to be checked at run-time. This guarantees that run-time checking will not introduce non-termination.

5.1 An Operational Semantics for CLP Programs with Assertions

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.[6]

In the case of programs with assertions finished derivations can be, in addition to successful and failed, also "erroneous". We introduce a class of distinguished states of the form $\langle \epsilon \mid A \rangle$ which cannot be further reduced. A finished derivation D is *erroneous* if *curr_state(D)*= $\langle \epsilon \mid A \rangle$, where A is (an identifier for) an assertion. Erroneous derivations indicate that the assertion A has been violated.

Let $\mathcal{A}[L]$ denote a renaming of the set of assertions $\mathcal{A}$ where assertions for the predicate of atom L have been renamed to match the variables of L. A state $\langle L :: G \mid \theta \rangle$, where L is a literal can be *reduced* as follows:

1. If L is a constraint and $\theta \wedge L$ is satisfiable, it is reduced to $\langle G \mid \theta \wedge L \rangle$.
2. If L is an atom,

[6] Program point assertions can be introduced by just allowing check literals to appear in the body of rules [27]. However, for simplicity we do not discuss program point assertions in this paper.

   - if $\exists\ A = calls(L, Cond) \in \mathcal{A}[L]$ s.t. $\theta \not\Rightarrow_P Cond$, then it is reduced to $\langle \epsilon \,|\, A \rangle$.
   - otherwise, let $PostC = \{check(S, A) | \exists A = success(L, C, S) \in \mathcal{A}[L] \wedge \theta \Rightarrow_P C\}$, if $\exists (L \text{:-} B) \in defn_P(L)$ then the state is reduced to $\langle B :: PostC :: G \,|\, \theta \rangle$.
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 we define *PostC* above as a set though it should actually be a sequence of *check* literals. We do so since the order in which the *check* literals are checked is irrelevant. We will use $\rightsquigarrow_{(P,\mathcal{A})}$ to refer to reductions performed using the above operational semantics of programs with assertions. 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 2 (Run-time Checking). *Given a program P, a set of assertions $\mathcal{A}$, and a set of queries $\mathcal{Q}$,*

$$A \textit{ is false iff } \exists\ D \in derivations_{\mathcal{A}}(P, \mathcal{Q}) : curr\_state(D) = \langle \epsilon, A \rangle.$$

Proof. A is false $\Leftrightarrow E(A, P, \mathcal{Q}) \neq \emptyset \Leftrightarrow \exists D \in derivations(P, \mathcal{Q})\ \neg solve(A, D, P)$. Let $D = S \rightsquigarrow_P^* S_n$ and let $sat_A(D)$ be false. Let $S_n = \langle G \,|\, \theta \rangle$. It can be proved that $\neg solve(A, D, P) \Leftrightarrow \exists D' \in derivations_{\mathcal{A}}(P, \mathcal{Q})$ s.t. $D' = S \rightsquigarrow_{(P,\mathcal{A})}^* S'_{n-1} \rightsquigarrow_{(P,\mathcal{A})} \langle check(Prop, A) :: G \,|\, \theta \rangle \rightsquigarrow_{(P,\mathcal{A})} \langle \epsilon, A \rangle$.

Theorem 2 guarantees that we can use the proposed operational semantics for programs with assertions in order to detect violation of assertions. Moreover, Theorem 3 below guarantees that the behavior of a partially correct program is the same under the operational semantics of Section 2 and under the semantics with assertions. If P is partially correct, it is straightforward to define a one-to-one relation between derivations $S \rightsquigarrow_P^* S_n$ and derivations $S \rightsquigarrow_{(P,\mathcal{A})}^* S'_n$, so that the two kinds of derivations only differ in the reductions of the distinguished literals of the form $check(L, A)$. We denote the corresponding isomorphism between derivations of the two kinds by $\approx$.

Theorem 3. *Let P be a program, $\mathcal{A}$ a set of assertions, and $\mathcal{Q}$ a set of queries. If P is partially correct w.r.t. $\mathcal{A}$ then $derivations(P, \mathcal{Q}) \approx derivations_{\mathcal{A}}(P, \mathcal{Q})$.*

Therefore, the semantics with assertions can also be used to obtain answers to the original query. Even though this semantics can be used to perform run-time checking, an important disadvantage is that existing CLP systems 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.[7] A second approach, which is the one used in our implementation, is based on program transformation.

[7] An alternative approach is to reduce such overhead by partially evaluating the meta-interpreter w.r.t. the program with assertions prior to performing run-time checking.

5.2 A Program Transformation for Run-Time Checking

We now present a program transformation technique which given a program P, obtains another program P' which checks the assertions while running on a standard CLP system. The meta-interpretation level mentioned above 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 $P' = rtchecks(\mathcal{A}, P)$, such that:

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

where

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

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).`".

Note that the above transformation will introduce nested levels of conditionals when there are several assertions for the same predicate. This is prevented in the implementation using an equivalent transformation, which avoids nesting conditionals. However, the transformation presented is easier to prove correct. The following theorem guarantees that the transformed program detects that an assertion is false iff it is actually false.

Theorem 4 (Program Transformation). *Let P be a program, $\mathcal{A}$ a set of assertions, and let $P' = rtchecks(\mathcal{A}, P)$. Given a set of queries $\mathcal{Q}$, $\forall A \in \mathcal{A}$: $E(A, P, \mathcal{Q}) \neq \emptyset$ iff $\exists D \in derivations(P', \mathcal{Q})$ s.t. $\exists S \in D$ with S of the form $\langle error(A) :: G \mid \theta\rangle$.*

Proof. (*Sketch*) There is a direct correspondence between $derivations(P', \mathcal{Q})$ and $derivations_{\mathcal{A}}(P, \mathcal{Q})$. The result then follows directly from Theorem 2.

6 Compile-Time Checking of Assertions

In this section we present some techniques for detecting errors at compile-time rather than at run-time, and also proving assertions to hold, i.e., (partially) validating specifications. 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 or because the available analyzers are not accurate enough. Those which cannot be checked at compile-time can, in general, (optionally) be checked at run-time.

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). An abstract value is a finite representation of a, possibly infinite, set of actual values in the concrete domain (D). The set of all possible abstract semantic values represents an abstract domain D_α which is usually a complete lattice or cpo which is ascending chain finite. However, for this study, abstract interpretation is restricted to complete lattices over sets both for the concrete $\langle 2^D, \subseteq \rangle$ and abstract $\langle D_\alpha, \sqsubseteq \rangle$ domains.

Abstract values and sets of concrete values are related via a pair of monotonic mappings $\langle \alpha, \gamma \rangle$: *abstraction* $\alpha : 2^D \rightarrow D_\alpha$, and *concretization* $\gamma : D_\alpha \rightarrow 2^D$, such that $\forall x \in 2^D : \ \gamma(\alpha(x)) \supseteq x$ and $\forall y \in D_\alpha : \ \alpha(\gamma(y)) = y$. In general $\sqsubseteq$ is induced by $\subseteq$ and α. Similarly, the operations of *least upper bound* ($\sqcup$) and *greatest lower bound* ($\sqcap$) mimic those of 2^D in a precise sense:

$$\begin{array}{r}
\forall \lambda, \lambda' \in D_\alpha : \ \lambda \sqsubseteq \lambda' \Leftrightarrow \gamma(\lambda) \subseteq \gamma(\lambda') \\
\forall \lambda_1, \lambda_2, \lambda' \in D_\alpha : \ \lambda_1 \sqcup \lambda_2 = \lambda' \Leftrightarrow \gamma(\lambda_1) \cup \gamma(\lambda_2) = \gamma(\lambda') \\
\forall \lambda_1, \lambda_2, \lambda' \in D_\alpha : \ \lambda_1 \sqcap \lambda_2 = \lambda' \Leftrightarrow \gamma(\lambda_1) \cap \gamma(\lambda_2) = \gamma(\lambda')
\end{array}$$

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 (L, λ), where L is an atom (for one of the exported predicates) and $\lambda \in D_\alpha$ an abstract constraint which describes the initial stores for L. A set of abstract queries $\mathcal{Q}_\alpha$ represents a set of queries, denoted $\gamma(\mathcal{Q}_\alpha)$, which is defined as $\gamma(\mathcal{Q}_\alpha) = \{(L, \theta) \mid (L, \lambda) \in \mathcal{Q}_\alpha \wedge \theta \in \gamma(\lambda)\}$. Such an abstract interpretation computes a set of triples $Analysis(P, \mathcal{Q}_\alpha, D_\alpha) = \{\langle L_p, \lambda^c, \lambda^s \rangle \mid p \text{ is a predicate of } P\}$. For each predicate p in a program P we denote L_p a representative of the class of all normalized atoms for p, and we assume that the abstract interpretation based analysis computes exactly one tuple $\langle L_p, \lambda^c, \lambda^s \rangle$ for each predicate p.[8] If p is detected to be dead code then $\lambda^c = \lambda^s = \bot$. As usual in abstract interpretation, $\bot$ denotes the abstract constraint such that $\gamma(\bot) = \emptyset$, whereas $\top$

[8] This assumption corresponds to a mono-variant analysis. Extension to a multi-variant analysis is straightforward, but we prefer to keep the presentation simple.

denotes the most general abstract constraint, i.e., $\gamma(\top) = D$. We now provide a couple of definitions which will be used below for stating correctness of abstract interpretation-based compile-time checking.

Definition 14 (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* $C(p, P, \mathcal{Q}) = \{\ \bar{\exists}_{L_p}\theta |\ \exists D \in derivations(P, \mathcal{Q})\ :\ curr\_state(D) = \langle L_p :: G \,\mathbf{|}\, \theta\rangle\ \}$.

Definition 15 (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* $S(p, \theta, P, \mathcal{Q}) = \{\ \bar{\exists}_{L_p}\theta' |\ \exists D \in derivations(P, \mathcal{Q})\ \exists G : \langle L_p :: G \,\mathbf{|}\, \theta\rangle \in D$ *and* $curr\_state(D) = \langle G \,\mathbf{|}\, \theta'\rangle\ \}$.

We can restrict the constraints in the calling and success contexts to the variables in L_p since this does not affect the evaluation of calls and success assertions. Correctness of abstract interpretation guarantees that for any $\langle L_p, \lambda^c, \lambda^s\rangle$ in $Analysis(P, \mathcal{Q}_\alpha, D_\alpha)$, $\gamma(\lambda^c) \supseteq C(p, P, \gamma(\mathcal{Q}_\alpha))$ 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* assertions [27].

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 16 (Trivial Success Set). *Given a literal L and a program P we define the* trivial success set *of L in P as* $TS(L, P) = \{\bar{\exists}_L\theta\ |\theta \Rightarrow_P L\}$.

This definition is an adaptation of that presented in [28], where analysis information is used to optimize automatically parallelized programs.

Definition 17 (Abstract Trivial Success Subset). *An abstract constraint $\lambda^-_{TS(L,P)}$ is an* abstract trivial success subset *of L in P iff* $\gamma(\lambda^-_{TS(L,P)}) \subseteq TS(L, P)$.

Lemma 2. *Let λ be an abstract constraint 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 $\exists\, \theta \in \gamma(\lambda)\ :\ \theta \Rightarrow_P L$.*

Proof. Let TS denote $TS(L, P)$.

1. $\lambda \sqsubseteq \lambda^-_{TS} \Rightarrow \gamma(\lambda) \subseteq \gamma(\lambda^-_{TS}) \subseteq TS \Rightarrow \forall\, \theta \in \gamma(\lambda) : \theta \in TS$.
2. $\lambda \sqcap \lambda^-_{TS} \neq \bot \Rightarrow \gamma(\lambda) \cap \gamma(\lambda^-_{TS}) \neq \emptyset \Rightarrow \exists\, \theta \in \gamma(\lambda) : \theta \in \gamma(\lambda^-_{TS}) \subseteq TS$.

Definition 18 (Abstract Trivial Success Superset). *An abstract constraint $\lambda^+_{TS(L,P)}$ is an* abstract trivial success superset *of L in P iff* $\gamma(\lambda^+_{TS(L,P)}) \supseteq TS(L, P)$.

Lemma 3. *Let λ be an abstract constraint 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 $\forall\, \theta \in \gamma(\lambda) : \theta \not\Rightarrow_P L$.*

Proof. Let TS denote $TS(L, P)$.

1. $\lambda^+_{TS} \sqsubseteq \lambda \Rightarrow TS \subseteq \gamma(\lambda^+_{TS}) \subseteq \gamma(\lambda) \Rightarrow \forall\, \theta \in TS : \theta \in \gamma(\lambda)$.
2. $\lambda \sqcap \lambda^+_{TS} = \bot \Rightarrow \gamma(\lambda) \cap \gamma(\lambda^+_{TS}) = \emptyset \Rightarrow \gamma(\lambda) \cap TS = \emptyset$.

In order to apply Lemma 2 and Lemma 3 effectively, accurate $\lambda^+_{TS(L,P)}$ and $\lambda^-_{TS(L,P)}$ are required. It is possible to find a correct $\lambda^+_{TS(L,P)}$, which may also hopefully be accurate, by simply analyzing the program with the set of abstract queries $\mathcal{Q}_\alpha = \{(L, \top)\}$. Since our analysis is goal-dependent, the initial abstract constraint $\top$ 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 [28], 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 [27] which provide this information. Also, since in this case the code of the predicate is present, analysis of the definition of the predicate p 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 constraint 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 that the user could write can be proved to hold at compile-time, but only those of them which coincide with some abstract constraint. This means that if we only want to perform compile-time validation, 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 with the property (type) `list` defined in Example 5). 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).

Checked Assertions. We now 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 (Checked Calls Assertion). *Let P be a program, $A = calls(p, Precond)$ 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 $\langle p, \lambda^c, \lambda^s \rangle \in Analysis(P, \mathcal{Q}_\alpha, D_\alpha)$. A is checked in P for $\mathcal{Q}$ if $\lambda^c \sqsubseteq \lambda^-_{TS(Precond,P)}$.*

Proof of theorem 5 is trivial since λ^c is a safe approximation of the calling context of the predicate of p. Thus, it provides a sufficient condition for a calls assertion to be checked.

Theorem 6 (Checked Success Assertion). *Let P be a program, $A = success(p, Pre, Post)$ 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 $\langle p, \lambda^c, \lambda^s \rangle \in Analysis(P, \mathcal{Q}_\alpha, D_\alpha)$. A is checked in P for $\mathcal{Q}$ if (1) $\lambda^c \sqcap \lambda^+_{TS(Pre,P)} = \bot$ or (2) $\lambda^s \sqsubseteq \lambda^-_{TS(Post,P)}$.*

Theorem 6 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 assertion holds and 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 safe approximation of the current stores of derivations in which the assertion is applicable.

False Assertions. Our aim now is to establish 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 7 (False Calls Assertion). *Assuming the premises of Theorem 5, then A is false in P for $\mathcal{Q}$ if $C(p, P, \mathcal{Q}) \neq \emptyset$ and $\lambda^c \sqcap \lambda^+_{TS(Precond,P)} = \bot$.*

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 8 (False Success Assertion). *Assuming the premises of Theorem 6, then A is false in P for $\mathcal{Q}$ if $\lambda^c \sqcap \lambda^-_{TS(Pre,P)} \neq \bot$ and $\lambda^s \sqcap \lambda^+_{TS(Post,P)} = \bot$ and $\exists\, \theta \in \gamma(\lambda^c \sqcap \lambda^-_{TS(Pre,P)}) : S(p, \theta, P, \mathcal{Q}) \neq \emptyset$.*

Now again, λ^s is an over-approximation, and in particular it can approximate the empty set. This is the rationale behind the final extra condition.

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. This is the subject of on-going research.

True Assertions. As with checked assertions, if an assertion is true then it is guaranteed that it will not 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 9 (True Success Assertion). *Assuming the premises of Theorem 6, then A is true in P if $\lambda^+_{TS(Pre,P)} \sqsubseteq \lambda^c$ and $\lambda^s \sqsubseteq \lambda^-_{TS(Post,P)}$.*

The first condition guarantees that λ^s describes any store which is a descendent of a calling state of p which satisfied the precondition. The second condition 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.

Equivalent Assertions. It may be the case that some assertions are not detected as checked or false at compile-time. One possibility here is to default to run-time checking of the remaining assertions. However, it is possible that part of the assertion(s) can be replaced at compile-time by a simpler one, i.e., one which can be checked more efficiently.

Definition 19 (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 assertions at compile-time by simplification of assertions can be done using techniques such as abstract specialization (see, e.g., [28]). For simplicity, we do not discuss here how to obtain simpler versions of assertions. However, the implementation of the framework discussed below simplifies whenever possible those assertions which cannot be guaranteed to be false nor checked at compile-time.

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 for different CLP dialects can be generated in a straightforward way. To date, we have developed two different debugging environments as instances of the proposed framework: CiaoPP [19], the Ciao system[9] preprocessor and CHIPRE [26], an assertion-based, type inferencing and checking tool (in collaboration with Pawel Pietrzak from the U. of Linköping and Cosytec). The type analysis used is an adaptation to CLP($\mathcal{FD}$) of the regular approximation approach of [16]. CiaoPP and CHIPRE share a common source (sub-)language (ISO-Prolog + finite domain constraints) and the Ciao assertion language [27],[10] so that source and output programs (annotated with assertions and/or run-time tests) within this (sub-)language can be easily exchanged. CHIPRE has been interfaced by Cosytec with the CHIP system (adding a graphical user interface) and is currently under industrial evaluation. CiaoPP is a more general tool which can perform a number of program development 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, 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 assertion language.[11]

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.

References

1. K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6):743–765, 1994.

[9] Ciao [4] is a next-generation, GNU-licensed Prolog system (available from `http://www.clip.dia.fi.upm.es/Software`). 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.

[10] As mentioned before, for clarity of the presentation in this work we have only addressed a subset of the assertion language.

[11] The full assertion language is also used by an automatic documentation generator for LP/CLP programs, LPdoc [18], which derives information from program assertions and machine-readable comments, and which generates manuals in many formats including postscript, pdf, info, HTML, etc.

2. K. R. Apt and D. Pedreschi. Reasoning about termination of pure PROLOG programs. *Information and Computation*, 1(106):109–157, 1993.
3. J. Boye, W. Drabent, and J. Małuszyński. Declarative diagnosis of constraint programs: an assertion-based approach. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 123–141, Linköping, Sweden, May 1997. U. of Linköping Press.
4. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997.
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, Linköping, Sweden, May 1997. U. of Linköping 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. P. Deransart, M. Hermenegildo, and J. Maluszynski. Debugging of Constraint Programs: The DiSCiPl Approach. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*. Springer-Verlag, 2000. To appear.
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, *Proc. of the 11th International Conference on Logic Programming*, pages 599–613. 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. A Documentation Generator for Logic Programming Systems. In *ICLP'99 Workshop on Logic Programming Environments*, pages 80–97. N.M. State University, December 1999.
19. M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 International Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.
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. 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.
22. P. Hill and J. Lloyd. *The Goedel Programming Language.* MIT Press, Cambridge MA, 1994.
23. J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
24. 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.
25. 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.
26. G. Puebla, F. Bueno, and M. Hermenegildo. A Generic Preprocessor for Program Validation and Debugging. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming.* Springer-Verlag, 2000. To appear.
27. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming.* Springer-Verlag, 2000. To appear.
28. G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.
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.