Declarative Diagnosis of Constraint Programs: an assertion-based approach

Johan Boye, Włodek Drabent, Jan Małuszyński*

April 21, 1997

Abstract

This paper discusses adaptation of the declarative diagnosis techniques for the use in constraint logic programming. The objective is to show how the well-known concepts are to be modified in this setting. In particular, the paper outlines basic algorithms for diagnosing incorrectness errors and insufficiency errors for constraint programs over arbitrary domains.

The main focus is on defining kinds of assertions needed to facilitate the task of answering of debugger queries. The examples illustrate the use of the proposed assertions in the declarative diagnosis algorithms for finite domain constraint programs.

1 Introduction

How to debug constraint logic programs is an important practical problem. Due to the complexity of the operational semantics of such programs, the information obtained by tracing the execution is difficult to understand. The alternative idea of declarative diagnosis, proposed for logic programs (see e.g. [19, 14, 17]), is applicable also in the case of constraint programs [16, 20]. However, the declarative diagnosis algorithms assume that the user is able to answer queries whether or not certain intermediate results of the computation correspond to her expectations. In practice, such questions may be very complex and virtually impossible to answer. We believe that this approach is unrealistic even for moderately sized programs.

This paper proposes an alternative approach where the user has a possibility of answering queries indirectly, by formulating simple assertions about the expected results. We will not commit ourselves to any specific assertion language, but an obvious possibility is to state assertions as additional constraints or small constraint programs. In this case, the capabilities of the constraint solver may be exploited to answer the queries. The use of assertions in declarative diagnosis of Prolog programs has been already advocated in our previous work [11, 12] but the assertions used there are different from those considered in this work.

In the framework of declarative diagnosis, two kinds of error symptoms are distinguished:

- An incorrectness symptom, where the program computes a result which is not expected by the user;

*Linköping University, Sweden; janma@ida.liu.se. The second author is also affiliated at Institute of Computer Science, Polish Academy of Sciences, Warsaw; drabent@ipipan.waw.pl.
This work was partly supported by ESPRIT project DiSCiPL (22532). The second author was also supported by Polish Academy of Sciences.

- An insufficiency symptom, where a program fails to compute some results expected by the user, after traversing whole search space (which must thus be finite).

Figure 1 shows the general situation.

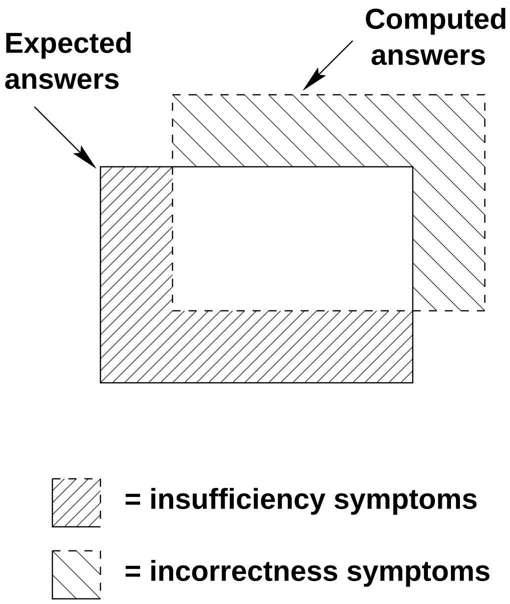


Figure 1: Error symptoms

The user initiates a diagnosis session when she observes an error symptom. In case of incorrectness, the proof tree resulting from the computation is analysed, and the aim of the diagnosis is to find an erroneous *clause* in the program. This kind of diagnosis is called *incorrectness diagnosis*. In case of insufficiency, the search space for a given goal is to be analysed. The aim in this case is to identify a unique *predicate* whose definition must be extended to avoid the insufficiency symptom that triggered the diagnosis. This *insufficiency diagnosis* also produces a constraint, that gives more precise information about the source of the problem.

incorrectness diagnosis

insufficiency diagnosis

When diagnosing incorrectness, the user has to tell the diagnoser if given computed results are correct or not. When diagnosing insufficiency, the user has to say whether or not some answers are missing in the set of all answers computed for a given goal. These two kinds of questions are of different nature and if we want to answer them by assertions, different kinds of assertions may be needed. The paper shows that in incorrectness diagnosis assertions specifying universal properties of the computed answers may be useful. A property which should be satisfied by every computed answer determines a superset of the set of correct answers and is called a *superset assertions*. Any answer which does not have the required property is thus incorrect. On the other hand, when diagnosing insufficiency we may state that there should be a computed answer satisfying certain property. Such an *existential assertion* may be very useful for providing an evidence that some answers are missing.

The paper is organised as follows. Section 2 provides some basic notions concerning constraint logic programs. Section 3.1 outlines our approach to incorrectness diagnosis with assertions, and section 3.2 discusses insufficiency diagnosis with assertions. Sections 4 and 5 outline some future research and discuss related work.

A running example of a simple constraint program (n queens) is used as an illustration of the approach.

2 Constraint Logic Programs

This section summarises some formal notions needed for a precise statement of the method. The reader familiar with foundations of constraint logic programming may directly proceed to Section 3.

As stated above, the declarative diagnosis is triggered when the user encounters a discrepancy between her expectations and the actual outcome of a computation. But outcomes of computations can be characterised by a declarative semantics, without referring to any specific model of computation. In our approach, this reference semantics used for comparison with user expectations will be the least $\mathcal{D}$-model semantics (see e.g. [15]). On the other hand, for localisation of errors we will use the notions of proof tree and search space, which can be linked to computations.

2.1 Syntax

We will consider definite programs interpreted over some a priori given domains, called the constraint domains.

A *constraint domain* is a pair $\langle \mathcal{L}, \mathcal{D} \rangle$ where $\mathcal{L}$ is a first-order language with equality, and $\mathcal{D}$ is a domain (a set). All function symbols of $\mathcal{L}$ are given a fixed interpretation on $\mathcal{D}$.

The predicate symbols of $\mathcal{L}$ are divided into two disjoint sets:

- *constraint predicates*, which are given a fixed interpretation in $\mathcal{D}$. These include the symbol $=$, interpreted as identity. *constraint predicates*
- *defined predicates*, which may occur in program clause heads and for which the user has an intended interpretation (on $\mathcal{D}$) *defined predicates*

A *primitive constraint* is an atomic formula whose predicate symbol is a constraint predicate, e.g. $\mathtt{X} \geq \mathtt{Y} + \mathtt{1}$. For the purpose of this paper we define a *constraint* to be either a primitive constraint, or a formula $c_1 \wedge c_2$, where c_1 and c_2 are constraints, or a formula $\exists_{\vec{x}} c$, where c is a constraint and $\vec{x}$ is a tuple of variables. Generally, constraints are defined as a subset of the first-order formulae over the alphabet of function symbols and constraint predicates.

constraint

An *atom* is a formula of the form $p(x_1, \ldots, x_n)$ where p is an n-ary defined predicate and $x_1, \ldots, x_n$ are distinct variables. A *clause* is a formula of the form $a \leftarrow c, b_1, \ldots, b_n$ where $n \geq 0$, $a, b_1, \ldots, b_n$ are atoms and c is a constraint. It may be assumed that clauses are normalized in the sense that all arguments in the atoms of a clause are variables and each variable in a clause occurs in at most one atom. The clauses used in the examples are not normalized and can be seen as a syntactic sugar for their normalized versions.

atom

clause

A *constraint program* is a set of clauses.

program

A *goal* is a conjunction of constraints and atoms. In the sequel we will assume that the initial goal consists of a single atom g of the form $p(\vec{x})$, and of a single constraint c, which may be empty. Such a *constrained atom* will be denoted $g[c]$. Every initial goal can be transformed into the required form by adding an extra clause to the program.

goal

constrained atom

A common practice in programming is to deal with data structures such as lists, trees, etc. In the examples discussed in this paper some function symbols of $\mathcal{D}$ are distinguished as *constructors*. Any nullary constructor k belongs to $\mathcal{D}$ and is interpreted as k. A n-ary constructor c is an interpreted as a function that maps

constructors

any n-tuple $(d_1, ..., d_n)$ of the domain elements into a tree (which is an element of the domain) whose root is labeled c and whose children are $d_1, ..., d_n$. Thus, $\mathcal{D}$ consists of some basic values and of labeled trees whose leaves are basic values. In the examples we will use the Prolog list notation when dealing with the list constructors.

2.2 Logical Semantics

valuation

$\mathcal{D}$-interpretation

A *valuation* is a mapping from the variables of $\mathcal{L}$ to $\mathcal{D}$. Valuations are extended to terms, atoms and constraints in the standard way. A valuation v *satisfies* a constraint c if $v(c)$ is true in $\mathcal{D}$. The notion of a *$\mathcal{D}$-interpretation* is a generalization of the notion of an Herbrand interpretation in logic programming. Constructors are interpreted as defined above. The remaning function symbols have their fixed interpretation in $\mathcal{D}$. We can view a $\mathcal{D}$-interpretation I as a set of objects of the form

$$p(d_1, \ldots, d_n)$$

$\mathcal{D}$-atom

$\mathcal{D}$-model

where p is an n-ary defined predicate, and $d_1, \ldots, d_n \in \mathcal{D}$. Such objects will be called *$\mathcal{D}$-atoms*. A $\mathcal{D}$-atom $p(\vec{d})$ is true in I iff $p(\vec{d}) \in I$. The usual concept of a model can now be defined as follows. A $\mathcal{D}$-interpretation I is a *$\mathcal{D}$-model* of a clause

$$h \leftarrow c, b_1, \ldots, b_n$$

if the following condition holds:

> Whenever a valuation v satisfies c, and $v(b_1), \ldots, v(b_n)$ are all true in I, then also $v(h)$ is true in I.

A $\mathcal{D}$-interpretation is a $\mathcal{D}$-model of a program if it is a $\mathcal{D}$-model of every clause of the program.

least $\mathcal{D}$-model

The $\mathcal{D}$-models are sets ordered by set inclusion. A well-known result is that every constraint program has a *least* $\mathcal{D}$-model, which can also be characterised as a least fixpoint of a monotonic operator (see e.g. Jaffar and Maher [15]). We will assume that the constraint domain $\mathcal{D}$ is fixed for a given program P.

represents

goal clause

Let g be an atom of the form $p(x_1, ..., x_n)$, where $x_1, ..., x_n$ are distinct variables. Then for every valuation v on $\mathcal{D}$ $v(g)$ is a $\mathcal{D}$-atom. A goal $g[c]$ *represents* the set of all $\mathcal{D}$-atoms $v(g)$ such that c is true under the valuation v. This set will be denoted $[\![g[c]]\!]$. By the *goal clause* corresponding to a goal $g[c]$ we mean $\leftarrow c, g$.

2.3 An Example Program

The n-queens problem is the problem of placing n chess queens on an $n \times n$ chessboard, so that no two queens attack each other. A solution to the problem can be represented as an n-place list, where the value j at place i means that the queen in file i is placed on row j. Fig. 2 shows a solution to the 8-queens problem, together with its list representation. The program in Fig. 2, which is written in the CLP language ECLiPSe [13], solves the problem. The domain $\mathcal{D}$ is the set of integers between 1 and some fixed integer `N` and of the lists over such integers. The call `make_list(N,List)` creates a list of `N` distinct variables, and the call `List :: 1..N` states that all variables in `List` range from 1 to `N`. The call `labeling(List)` enumerates all the possible values for `List` in a straightforward way. The predicate `constrain_queens` and its sub-predicates constrain the variables in `List`. The constraint predicates are `##` and `#=`. They denote "not equal to" and "equal to", respectively. The non constructor function symbols are the integers and the arithmetic operators, interpreted in the obvious way.

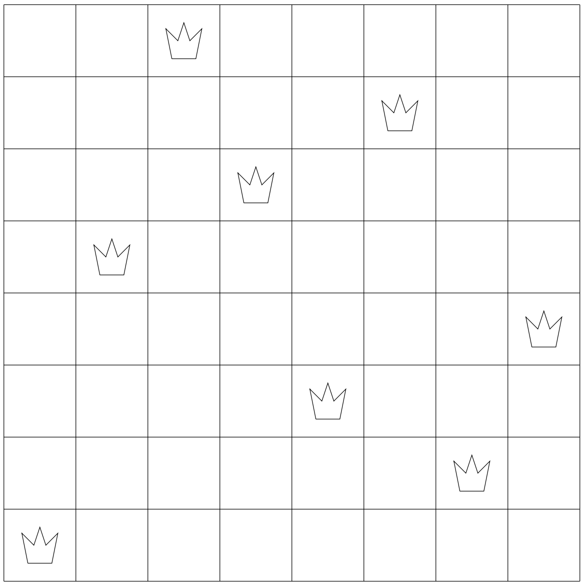
2.4 Proof trees

This section discusses a notion of a proof tree, which makes it possible to relate the least model semantics of a program to computations. The approach follows that of [8]. The notion of a proof tree will be used in declarative diagnosis of incorrectness.

A *skeleton* is a labeled ordered tree with nodes labeled by (renamed) clauses of the program, by a goal clause or by "?" and such that: *skeleton*

- the root is labeled by a (program or goal) clause,
- if a node is labeled by a clause of the form $h \leftarrow a, b_1, \ldots, b_n$ (or by a goal clause $\leftarrow a, b_1$ with $n = 1$), where a is the constraint of the clause, then

---



[1, 5, 8, 6, 3, 7, 2, 4]

```
nqueens(N,List) ←
    make_list(N,List),
    List :: 1..N,
    constrain_queens(List),
    labeling(List).

constrain_queens([ ]).
constrain_queens([X|Y]) ←
    safe(X,Y,1),
    constrain_queens(Y).
```

```
safe(_,[ ],_).
safe(X,[Y|T],K) ←
    noattack(X,Y,K),
    K1 #= K+1,
    safe(X,T,K1).

noattack(X,Y,K) ←
    X ## Y,
    X+K ## Y,
    X-K ## Y.
```

Figure 2: Above: A solution to the 8-queens problem with its list representation
Below: A program for the n-queens problem

- it has n children,
- if the i-th child is labeled by a clause then the head predicate of this clause and the predicate of b_i are identical;

- no two clauses labeling two distinct nodes of the skeleton have a common variable;
- if a node is labeled by "?" then it is a leaf node; if a node is labeled by a goal clause then it is the root.

incomplete node

complete skeleton

Notice that a skeleton defines two ordering relations on the nodes: the parent-child ordering and the left-to-right ordering. A (leaf) node labeled by "?" will be called an *incomplete node* of the skeleton. A skeleton is said to be *complete* if it has no incomplete nodes.

associated atom

Let i be a number such that a node of a skeleton is the i-th child of its parent in the left-to-right ordering. Then the i-th body atom in the clause instance labeling the parent is said to be the *associated atom* of the node. Notice that for the root node the associated atom does not exist.

For a given skeleton S let $n_1, \ldots, n_k$ be the sequence of all incomplete nodes of S. The sequence ordering is defined by the left-to-right ordering on the nodes of S. By a *frontier* of a S we mean the sequence $b_1, \ldots, b_k$ of the associated atoms of $n_1, \ldots n_k$.

clause skeleton

A skeleton S is said to be a *clause skeleton* of a clause c iff its root is labeled by c and the children of the root are incomplete nodes.

skeleton's set of constraints

For a given skeleton S the set $C(S)$ of constraints, to be called *the set of constraints of S*, consists of:

- the constraints of all clauses labeling the nodes of S
- all equations $\vec{x} = \vec{y}$ where $\vec{x}$ are the arguments of the i-th body atom of the clause labeling a node n of S and $\vec{y}$ are the arguments of the head atom of the clause labeling the i-th child of n.

proper skeleton

proof tree

A skeleton is called *proper* if its set of constraints is satisfiable.

A *proof tree* for a program P is a complete skeleton whose set of constraints is satisfiable and whose nodes are labeled by variants of the clauses of P or by some goal clause. A graphical representation of an example proof tree is given in Fig. 4. In this representation the nodes are labeled not by clauses but by the associated atoms. Moreover, the arguments are replaced by their values whenever the constraints define them uniquely. (This representation is closer to a usual notion of a proof tree in logic programming).

proof tree for a constrained atom

We will say that a proof tree T (with the set of constraints C and the root labeled by $\leftarrow c', h$) is a *proof tree for a constrained atom $h[c]$* iff c is equivalent to $\exists_h C$, where $\exists_h C$ denotes existential quantification over all variables of C that do not occur in h.

For practical purposes it would be important to simplify constraints associated with a given skeleton.

2.5 Towards an operational semantics

The objective of an ideal operational semantics would be to construct all proof trees of a given goal. A simple idea would be to construct skeletons in a top-down manner, and to check satisfiability of the accumulated constraints. One should not delay the satisfiability check until construction of a complete skeleton, since for every recursive program the number of skeletons is infinite. A natural choice would be to check satisfiability after adding every new instance of a clause to the existing skeletons.

However, in general there may be no complete algorithms for checking satisfiability of constraints in a given constraint domain. Following [20] this can be abstracted as an assumption that there is some sufficient condition for unsatisfiability to be called the *reject criterion.* This is an abstraction for a spectrum of behaviours. An operational semantics of a program can now be defined in terms of skeletons, using reject criterion. We will also need the concept of a *selection rule* (or computation rule), which selects an incomplete node of a given skeleton in order to extend it with a copy of a program clause. In this paper we assume that the selection rule is total, i.e. that the selection rule will always select an incomplete node if such a node exists.

reject criterion

selection rule

For given goal $p(x_1, \ldots, x_n)[c]$, reject criterion and selection rule, a *computation* is a sequence of proper skeletons $S_0, \ldots, S_k$, such that S_0 is a clause skeleton of a clause of the form $\leftarrow c, p(x_1, .., x_n)$, and for each $i = 1, \ldots, k$ S_i is obtained by extending the incomplete node of S_{i-1} selected by the selection rule. By the *selected subgoal* of the computation step we mean the constrained atom $g[c]$ where g is the associated atom of the selected node of S_{i-1} and c is the constraint of this skeleton.

computation

selected subgoal

Notice that there may be several different extensions of the selected node. This happens if several program clauses are applicable and give proper skeletons. Thus, for a given selection rule and given initial goal $g[c]$ the *search space* may be represented as a tree with the root labeled by the clause skeleton for the goal clause $\leftarrow c, g$. Every node corresponds then to a skeleton, and each of its children corresponds to a different extension. The transitive and irreflexive closure of the child relation will be called the offspring relation.

search space

Every node of the search space is labeled by a sequence of atoms and has an associated constraint. Both elements are determined by the skeleton S corresponding to the considered node. They are: the frontier of S and the constraint of S. A leaf labeled by an empty sequence whose associated constraint is satisfiable[1] will be called a success node. In practice we have to extend this concept by replacing satisfiability condition by the condition that the constraint is not rejected by the reject criterion.

In the special case of the Herbrand constraint domain the constraints are represented by substitutions and the notion of search space reduces to the notion of SLD-tree.

If the selection rule is that of Prolog then a node n' of the search space is said to be a *proper offspring* of a node n labeled by a sequence of atoms $a_1, .., a_n$, iff

proper offspring

- n' is an offspring of n in the search space,
- the label of n' is $a_2, ..., a_n$,
- there is no node m with the above properties and such that m is an offspring of n and n' is an offspring of n.

So the fragment of computation between node n and n' corresponds to a success of a_1.

As explained above, every computation of a program corresponds to construction of a branch of the search space. By an *exhaustive search* we mean the process which eventually constructs the whole search space. In practice this is possible only if the space is finite.

search

3 Declarative Diagnosis with Assertions

In the previous work on declarative diagnosis it has been assumed that there exists a unique set I, usually called "the intended model", describing exactly the expec-

[1] Thus the corresponding skeleton is a proof tree.

tations of the user. When referring to the least model semantics, this would mean that by correcting all errors in a given program, we would obtain a new program with the least model I. Recall that in the case of constraint programs the elements of the set I are $\mathcal{D}$-atoms. Generally, a single $\mathcal{D}$-atom may not be expressible in the language, for example when the elements of $\mathcal{D}$ are real numbers. In that case the language can only specify sets of atoms.

assertion

An assertion is a statement that partially describes I. Our example assertions will describe supersets of I when diagnosing incorrectness, and sets having non-empty intersection with I, when diagnosing insufficiency. In this article, we will not concern ourselves with any rigorous definition of the assertion language (the assertions will be given in natural language, and their semantics will be informally discussed). However, in all the examples the assertions could be defined as simple constraints or small constraint programs.

3.1 Incorrectness Diagnosis

incorrectness symptom

Incorrectness is a situation when a computed constraint does not conform to the expectations of the user. More precisely, for a given goal[2] $g[c]$ a constraint a is computed such that the set of $\mathcal{D}$-atoms represented by $g[a]$, denoted $[\![\, g[a] \,]\!]$ is not a subset of the intended model I. The constrained atom $g[a]$ is called an *incorrectness symptom.* We will say that $g[a]$ is *true* in the intended model I iff $[\![\, g[a] \,]\!] \subseteq I$.

incorrectness diagnosis

Incorrectness diagnosis starts from an incorrectness symptom, and attempts to localize the clause in the program that is responsible for this unexpected result. More precisely, the aim is to find a clause $h \leftarrow c, \vec{b}$ such that, for some valuation v satisfying c, $v(\vec{b})$ is true in the intended model I, and $v(h)$ is false in I. It is easy to prove that such a clause must exist whenever an incorrectness symptom exists (see e.g. Lloyd [17]). Intuitively, such a clause is the source of incorrectness, since it transforms expected intermediate results into an incorrectness symptom. From the logical point of view, the clause is false in the intended model I.

intermediate result

Since an incorrectness symptom $g[a]$ has been computed, there is a proof tree T for it. (So the root of T is labeled by a goal clause $\leftarrow c, g$ and has a single child with the associated atom g). Let t be the conjunction of the constraints of T. By the *intermediate result* connected to a node n of T we mean the constrained atom $p_n[t_n]$, where p_n is the associated atom of n and t_n is the constraint $\exists_n t$ where $\exists_n$ denotes existential quantification over all variables of t which do not appear in p_n. Clearly, the intermediate result connected to the child of the root is an incorrectness symptom (as a is $\exists_m t$, where m is the root's child).

The diagnosis process is essentially a (partial) traversal of the proof tree T. In the basic algorithm, the diagnoser poses a question to the user for each visited node: "Is this intermediate result what you expect?", or equivalently, "Is this constrained atom true in the intended model?". The algorithm halts when encountering a node n such that the intermediate result connected to n is an incorrectness symptom and the intermediate results of all children of n are true in I. It reports then that the clause labeling n is incorrect, and shows the intermediate results as justification.

The problem is that the questions mentioned in the previous paragraph can be very difficult to answer. Our intention is to facilitate this task by allowing the user the possibility of answering the questions by giving assertions. For instance, when facing the problem of answering the question "Is $g[c]$ true in the intended model?", the user may answer: "I expect that $g[c]$ has the property P". The diagnoser then checks whether the property P is satisfied or not. In case of a negative answer (i.e. if $g[c]$ is in the shaded area of Fig. 3), the diagnoser concludes that $g[c]$ is an

[2]Remember that we consider atomic initial goals, to simplify the presentation. This does not lead to a loss of generality.

incorrectness symptom, and progress is made in the diagnosis process. Otherwise, $g[c]$ is or is not an incorrectness symptom but property P is too weak to conclude this.

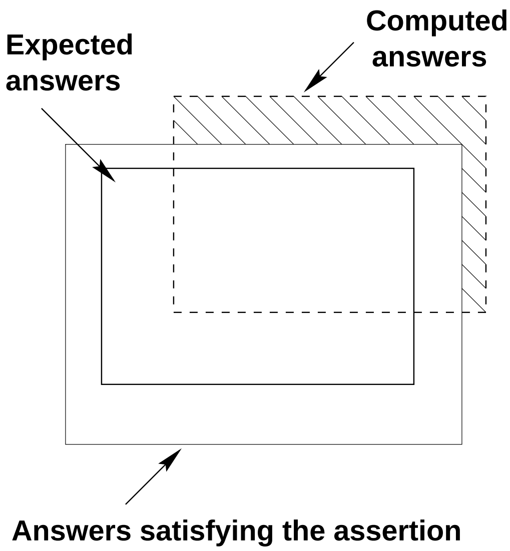


Figure 3: A superset assertion

This kind of assertions will be called *superset assertions*, since they, roughly speaking, define a superset of the set represented by the expected answers. The set of $\mathcal{D}$-atoms represented by any correct answer should be included in the set represented by a superset assertion. This is a constraint entailment check if the assertion is expressed as a constraint. The set of all assertions given for a predicate, will be called the *specification* for the predicate.

superset assertion

specification

We will now summarize how the diagnosis algorithm works. At the beginning of the diagnosis process, the specification for each predicate is empty. As stated above, the proof tree for the incorrectness symptom is traversed. At each visited node, the system first attempts to show that the associated constrained atom $p(x_1, \ldots, x_n)[c]$ is an incorrectness symptom by using the current specification for p. If this fails, a question "Is $p(x_1, \ldots, x_n)[c]$ true in the intended model?" is posed by the system. The user can answer with:

- a specification that augments the existing one. If the augmented specification is not sufficient for showing that $p(x_1, \ldots, x_n)[c]$ is an incorrectness symptom the question is repeated. If it shows that $p(x_1, \ldots, x_n)[c]$ is an incorrectness symptom then the algorithm proceeds as in the case of "No" answer below.
- a "Yes" answer: the user confirms that $p(x_1, \ldots, x_n)[c]$ is true in the intended model. In this case the subtree rooted at this node will be excluded from further search.
- a "No" answer: the user confirms that $p(x_1, \ldots, x_n)[c]$ is an incorrectness symptom. In this case the diagnosis re-starts for the subtree rooted at one of the children nodes.
- a "Don't know" answer. A simple solution is to exclude the subtree rooted at this node from further search, similarly as in the "yes" case. An alternative approach would be to search through this subtree with the hope that it includes error symptoms which can be identified.

If "don't know" subtrees are not examined the search terminates when all answers for n are "yes" or "don't know". If all answers are "yes" the clause labeling n is reported erroneous. If at least one of the answers is "don't know" the error is not precisely localized. It is reported that it is caused by the clause labeling either n or a node in one of the subtrees rooted at the children of n with "don't know" answers. Better localization of the error may be achieved by finding an error symptom in one of the "don't know" subtrees. The decision, where to search may be consulted with the user. Alternatively, the "don't know" answers may be withdrawn with a new effort to give a definitive answer to the queries.[3]

Example 1 Consider the queens program of Sect. 2.3. We introduce an error in the program by replacing the clause for `noattack` by the clause

```
noattack(X,Y,K) ←
    X ## Y,
    X+K ## Y.
```

This clause is now under-constrained, and the first result produced by the call `nqueens(8,List)` is

$$\texttt{List} = [1,4,8,7,2,5,3,6]$$

It is possible to discover that this solution is incorrect by looking at a fragment of the list above: The queens on files 3 and 4 are placed on rows 8 and 7, respectively, so one can directly see that they attack each other. Thus the incorrect solution violates a general property of the desired solutions, expressed below as an assertion.

> **Assertion:** Whenever `constrain_queens`(l) is a logical consequence of the program, no two consecutive values in the list l should differ by 1.

The above assertion is a superset assertion, i.e. every desired solution satisfies this assertion, but a list satisfying the assertion is not necessarily a desired solution. We will now illustrate how this assertion is used in the debugging process. Fig. 4 shows the proof tree for the incorrectness symptom

$$\texttt{constrain\_queens}([1,4,8,7,2,5,3,6])$$

At the top node, the user is asked whether this result is true or not in the intended model. The user responds by giving the assertion above. Since the atom at this node does not satisfy the assertion, the diagnoser concludes that the atom is an incorrectness symptom, and proceeds to examine the children nodes. Since the child node

$$\texttt{constrain\_queens}([4,8,7,2,5,3,6])$$

and its child node

$$\texttt{constrain\_queens}([8,7,2,5,3,6])$$

do not satisfy the assertion, the diagnoser automatically commits to the subtree rooted by the latter atom. The second question posed to the user thus concerns one of the children nodes, i.e.

$$\texttt{safe}(8,[7,2,5,3,6],1)$$

[3]The algorithm outlined above commits the search to a subtree as soon as a "no" answer is encountered. It is possible that some nodes excluded from the search would also give a "no" answer. Exploring these branches would also lead to (partial) localization of an erroneous clause. This may be another occurrence of the same erroneous clause, or an occurrence of another erroneous clause. Finding and correcting one error at a time may be a good strategy but if answering queries in one "no" branch is too difficult, it may be sometimes worthwhile to explore some other branches.

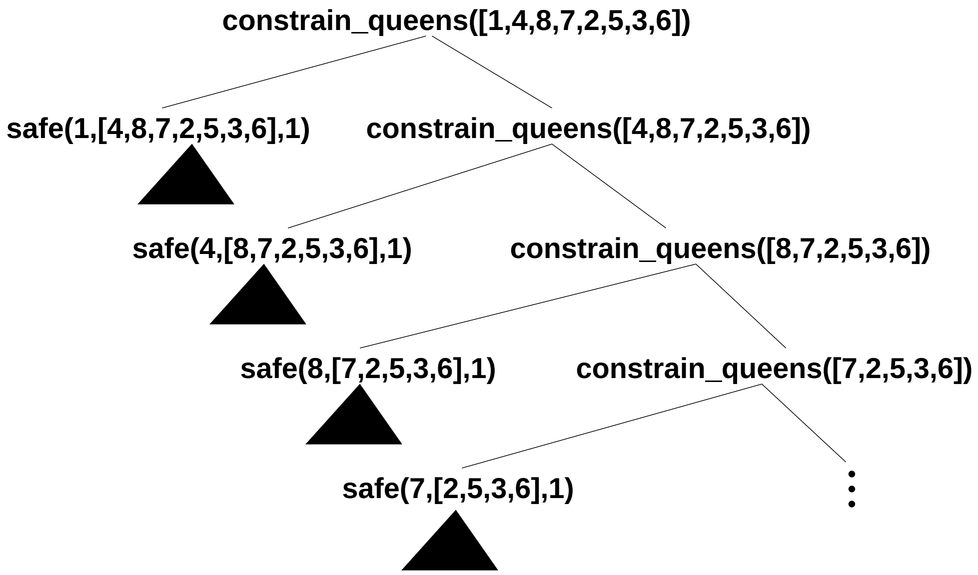


Figure 4: A proof tree. At the nodes their associated atoms are displayed (instead of the clauses labelling the nodes). The root node is not shown (it does not have its associated atom).

Intuitively, this atom is incorrect for the same reason as the previous incorrectness symptoms: The queen on row 8 attacks the queen on row 7. The user can now give a new assertion concerning the predicate `safe`, or simply answer "No". In the latter case, the diagnoser will immediately proceed to a child node, for instance

$$\texttt{noattack}(8, 7, 1)$$

If the user confirms that this atom is an incorrectness symptom, the diagnoser concludes that the clause

```
noattack(X,Y,K) ←
    X ## Y,
    X+K ## Y.
```

is erroneous, since the body consists only of constraints. (End of example)

The example shows a simple assertion that eliminates the need for user interaction in several nodes of the proof tree. Notice that the granularity of incorrectness diagnosis is limited to program clauses. Notice also that a superset assertion is not able to confirm correctness of a constrained atom; it is only able to find out its incorrectness.

3.2 Insufficiency Diagnosis

Informally, insufficiency is the situation where a program fails to compute some expected solution. For a precise definition, we need some preliminary notions. Throughout this section we require that the Prolog selection rule is used (the leftmost literal is selected).

Suppose the execution for a goal $g[c]$ terminates after having produced answer constraints $a_1, \ldots, a_n$ (upon backtracking). Then the set $\{g[a_1], \ldots, g[a_n]\}$ will be *answer collection* called the *answer collection* for P and $g[c]$. The union

$$\bigcup_i [\![g[a_i]]\!]$$

answer set of the sets represented (conf. Section 2.2) by the answer collection will be called the *answer set* of the goal.

slice A *slice* of a set S of $\mathcal{D}$-atoms w.r.t. a constrained atom a is the intersection of S and the set represented by a.

sufficient A program P is *sufficient* for a goal g iff the search space for P and g is finite, and the slice of the intended model wrt g is a subset of the answer set.

insufficient A program P is *insufficient* for a goal g iff the search space for P and g is finite, and P is not sufficient for g. A goal g such that P is insufficient for g will be called *insufficiency symptom* an *insufficiency symptom* of P. The notion of insufficiency extends naturally for non-atomic goals.

insufficiency diagnosis *Insufficiency diagnosis* starts from an insufficiency symptom, and attempts to localise the predicate in the program whose definition needs to be augmented. More precisely, the aim is to find a constrained atom $q(x_1, \ldots, x_n)[c]$ with the following two properties:

- $q(x_1, \ldots, x_n)[c]$ is an insufficiency symptom, and
- whenever $q(x_1, \ldots, x_n) \leftarrow c', \vec{b}$ is a (possibly renamed) clause of the program, the program is sufficient for $\vec{b}[c \wedge c']$.

uncovered atom Such a constrained atom will be called an *uncovered* atom.

A basic insufficiency diagnosis algorithm communicates with the user by asking queries of the form: "Does the answer collection $\{g[c_1], \ldots, g[c_k]\}$ $(k \geq 0)$ represent all the expected solutions of the atomic query $g[c]$?" (In our terminology, the question is whether the program is sufficient for the selected subgoal). Again, such questions are usually hard to answer.

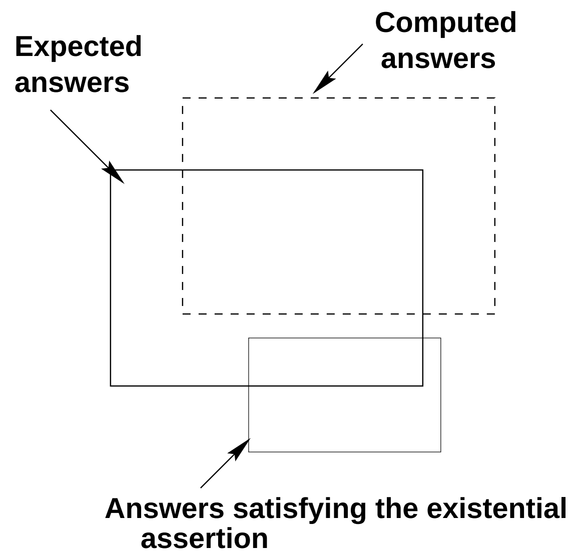


Figure 5: An existential assertion

existential assertion We propose to give the user a possibility to answer the questions by giving existential assertions. For instance, when facing the problem of answering the question

"Do $g[c_1], \ldots, g[c_k]$ represent all solutions to the goal $g[a]$?", the user may reply "I expect that there is a solution with the property P". The diagnoser then computes the intersection between the union of the sets represented by $g[c_1], \ldots, g[c_k]$, and the set of $\mathcal{D}$-atoms having the property P. If the intersection is empty (illustrated in Fig. 5), the diagnoser concludes that $g[a]$ is an insufficiency symptom, and progress is made in the diagnosis process. Otherwise, either $g[a]$ is not an insufficiency symptom or P is too weak to provide evidence for this.

In the insufficiency diagnosing algorithm, the search space[4] (with the Prolog selection rule) for the insufficiency symptom is traversed. At every step of the traversal there is a set of *active* nodes from which the visited node is selected. The initial set of active nodes includes the children of the root node of the search space[5]. At each node visited, the diagnoser asks the user whether the computed constraints for the selected subgoal at the node represent all the expected solutions. More specifically, the user faces a question of the form "Do $g[c_1], \ldots, g[c_k]$ represent all solutions to the goal $g[a]$?". She may answer it in one of the following four ways:

active node

- by giving an existential assertion. If the existential assertion is not sufficient for showing that $g[a]$ is an insufficiency symptom, the question is repeated. Otherwise, $g[a]$ is an insufficiency symptom and the algorithm proceeds as in the case of "No" answer below.
- a "Yes" answer: the user confirms that $g[c_1], \ldots, g[c_k]$ represent all the desired solutions to the goal $g[a]$. In this case the visited node in the set of active nodes is replaced by the set of all its proper offsprings (conf. page 7).
- a "No" answer: the user confirms that $g[a]$ is an insufficiency symptom. In this case the insufficiency diagnosis re-starts with the search space of this symptom.
- a "Don't know" answer: the diagnosis proceeds as in the "Yes" case. However, the user may return to this node at a later stage if she so pleases.

Upon termination of the diagnosis, the most recently found insufficiency symptom is returned as the result. If no "don't know" answers appear in the search space for the symptom, the symptom is also an uncovered atom. Otherwise an uncovered atom is not fully localised. If for a node n of the search space an answer "don't know" was given then the uncovered atom may be present in the offsprings of n. More precisely, in those offsprings of n that are not offsprings of any proper offspring (conf. Section 2.5) of n. (In other words, in the nodes between n and its proper offsprings). So, if a "don't know" answer is given about a procedure call then it is possible that the error, for which we search, occurred in the computations related to this call.

Even in the latter case the set of suspected predicates may constitute only a small subset of all program predicates.

Example 2 Consider again the queens program of Sect. 2.3. We introduce an error in the program by replacing the clause for `noattack` by the clause

```
noattack(X,Y,K) ←
    X ## Y,
    X+K ## Y,
    X-K ## Y,
    X-K ## Y-1.
```

[4] SLD-tree in the case of the Herbrand constraint domain.

[5] So if the computation begins with an initial goal $h[c]$ then, for any element of the initial set, its frontier is the right hand side of a clause applied to the initial goal.

The clause is now over-constrained, and only two solutions are computed for the call

$$\texttt{nqueens}(8, \texttt{List})$$

The user knows that there should be more solutions, so she identifies the call as an insufficiency symptom. The questions asked by the diagnosing algorithm rapidly become very difficult. For instance, one of the first subgoals to be computed is

$$\texttt{safe}(\texttt{A}, [\texttt{B}, \texttt{C}, \texttt{D}, \texttt{E}, \texttt{F}, \texttt{G}, \texttt{H}], 1)$$

which yields the constraint

```
A##B, 1+A-B##0, -1+A-B##0, 0+A-B##0, A##C, 2+A-C##0, -2+A-C##0,
-1+A-C##0, A##D, 3+A-D##0, -3+A-D##0, -2+A-D##0, A##E, 4+A-E##0,
-4+A-E##0, -3+A-E##0, A##F, 5+A-F##0, -5+A-F##0, -4+A-F##0, A##G,
6+A-G##0, -6+A-G##0, -5+A-G##0, A##H, 7+A-H##0, -7+A-H##0,
-6+A-H##0
```

Even for this small constraint, it is difficult to decide whether or not it reflects our expectations. We now give the following existential assertion:

> **Existential assertion:** There should be a solution where the queen in the first file is placed on row 7, i.e. there should be a logical consequence of the program which is an instance of $\texttt{safe}(7, [\texttt{B}, \texttt{C}, \texttt{D}, \texttt{E}, \texttt{F}, \texttt{G}, \texttt{H}], 1)$.

The goal $\texttt{safe}(7, [\texttt{B}, \texttt{C}, \texttt{D}, \texttt{E}, \texttt{F}, \texttt{G}, \texttt{H}], 1)$ returns the constraint

```
B[1..5], C[1..4,8], D[1..3,6,8], E[1,2,5,6,8],
F[1,4..6,8], G[3..6,8], H[2..6,8]
```

(where the numbers in brackets denote the possible values for the different variables). The constraint is illustrated in Fig. 6, where the white squares denote the possible instantiations for the different variables, and the shaded squares denote the impossible instantiations.

The user is now asked the question whether or not this answer corresponds to her expectations. Although non-trivial, this constraint is at least easier to digest than the previous one. By looking at the picture in Fig. 6, one can conclude e.g. that the value 6 for the variable `C` is erroneously excluded. The user can thus answer the question negatively. (Another possibility is to strengthen the previous existential assertion, by asserting the existence of a solution where `C` has the value 6. However, although this assertion is true, this is far from obvious.)

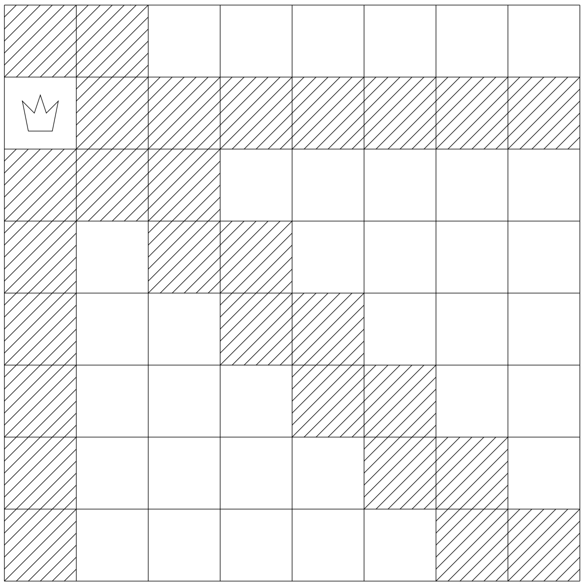
If the user confirms that the $\texttt{safe}(7, [\texttt{B}, \texttt{C}, \texttt{D}, \texttt{E}, \texttt{F}, \texttt{G}, \texttt{H}], 1)$ (together with the constraint above) is an insufficiency symptom, the diagnoser proceeds with the first subgoal in the computation of $\texttt{safe}(7, [\texttt{B}, \texttt{C}, \texttt{D}, \texttt{E}, \texttt{F}, \texttt{H}], 1)$. Table 7 shows the subsequent questions from the diagnoser, and the user's response.

Since the body of the clause for `noattack` consists only of constraints, the constrained atom

$$\texttt{noattack}(7, \texttt{C}, 2), \texttt{C}[1..4, 8]$$

is an incompletely covered atom, revealing that the definition of `noattack` needs to be extended. (End of example)

The example shows how the use of existential assertions may simplify answering the queries. In contrast to the previous example, we do not have here a case where a single assertion is able to answer more than one queries posed by the algorithm. Note that the granularity of insufficiency diagnosis is that of program predicates.



[7, B, C, D, E, F, G, H]

Figure 6: Pictorial representation of a constraint

4 Further work

We have shown that the classical declarative techniques apply in principle also in the case of constraint programs. However, the adaptation requires modification of certain semantic notions, such as proof tree and search space, which provide a formal basis for declarative diagnosis. Due to the nature of the semantic domains the queries in declarative diagnosis of constraint programs become even more complicated than in the case of logic programs, as can be seen from the examples. The use of assertions seems to be a must. We outlined basic notions needed to define declarative diagnosis algorithms for constraint programs. We sketched such algorithms. In contrast to the classical algorithms we study also the case of "don't know" answers.

We proposed two different kind of assertions and we illustrated on simple examples their potential usefulness for declarative diagnosis algorithms.

Goal	Answer constraint	User replies
`noattack(7,B,1)`	`B[1..5]`	Yes
`safe(7,[C,D,E,F,G,H],2)`	`C[1..4,8],D[1..3,6,8],E[1,2,5,6,8], F[1,4..6,8],G[3..6,8],H[2..6,8]`	No
`noattack(7,C,2)`	`C[1..4,8]`	No

Figure 7: Further questions and answers in a diagnosis session

The approach presented above is preliminary. Experiments are needed to collect some body of example assertions and to evaluate the usefulness of the paradigm. Maybe some other kinds of assertions should also be considered.

Using assertions which are constraints or simple constraint programs in the underlying language would allow to use the solver for answering the queries. The example assertions in this paper are of this kind. The tests needed to answer the queries are a disentailment test for incorrectness and an unsatisfiability test for insufficiency diagnosis.

Another way would be to exploit for the purpose of diagnosis some more powerful constraint solver in addition to that of the actual CLP system. Sometimes the solvers implemented are restricted due to efficiency reasons, for example CLP(R) does not solve non-linear constraints. In debugging, efficiency is much less important, what matters is the time spent by the users on thinking. One could expect that the additional functionality of more sophisticated solvers could contribute to the diagnosis. We may expect that eg. constraint negation and constraint entailment could be useful.

The examples show also that the problem-specific graphical representation is of great importance for answering debugger queries.

We believe that in many cases it is unrealistic to assume that the intended model is 2-valued. Often it is actually 3-valued. The user knows that some $\mathcal{D}$-atoms should be true (w.r.t. the intended model), some should be false, the rest are irrelevant. This may be related to the implicit assumption about kind of data used in execution of the program. For example, take the Herbrand domain and the standard append predicate. The programmer may implicitly assume that append is only called with the arguments being lists or variables. She may not know if *append* with certain non-list arguments should be true or false in the intended model. If we want to make all such atoms fail, we obtain a program with additional type checks, which is inefficient and different from the program actually requested. However, if such atoms appear during the computation this may be due to errors not in program but in input data, which do not satisfy the imposed assumptions. We believe that such a case should be distinguished in the diagnosis process from the case of program error symptoms.

From our previous experience [12, 18] we know that a simple implementation of a diagnosis algorithm is often rather unpleasant to use. The programmer has to answer a sequence of questions, in an order fixed by the debugger. She has no possibility to correct her own errors, to delay answers to difficult questions, to influence the workings of the debugger, etc. It is important to make such implementation user-friendly. The user should not feel restricted. It should be possible eg. to switch from insufficiency diagnosis to incorrectness diagnosis (when it turns out that one of the computed answers posed in the queries is incorrect). It should be possible to make (and withdraw) hypotheses. ("I cannot say if this sophisticated constrained atom is correct. Assume that it is, what happens then.")

5 Related work and discussion

The foundations of declarative diagnosis for constraint logic programs have been presented by Le Berre and Tessier [16, 20]. Our intention was to give an introduction to the declarative diagnosis in a setting where the oracle could be (partly) replaced by assertions. The aspect of query answering is the main focus of this paper in contrast to [16, 20] where semantic details of the diagnosis are studied.

We advocate answering the queries by assertions. The traditional yes/no answers are also allowed[6]. In addition, we allow also "don't know" answers. In this case

[6]They could also be seen as specialised types of assertions [11]. This was, however, not discussed

localisation of the error may be (but sometimes is not) less precise.

Several kinds of assertions have been proposed for various purposes. Generally assertions play a role of (partial) specifications of intended programs (i.e. describe some intended properties of a program). Then one may consider :

- Checking them at run time, as proposed in [21].
- Proving them. General techniques for proving assertions are described in e.g. [7] and [8] for declarative properties, and in e.g. [10, 9] for run-time properties. To mechanise such proofs, rather powerful general automatic provers would be necessary. However, for some restricted classes of assertions proving them boils down to relatively simple and efficient checks. They could be performed by programming systems as compile time checks. The problem of automatic type checking of directional types has been discussed e.g. in [21, 1, 2].
- Using them for localising errors in the case of error symptoms, as discussed in [11, 12] and this paper.

We may also consider algorithms that for a given program produce assertions describing its properties. The standard techniques for that purpose is abstract interpretation. The work presented in [3, 4, 5] is of direct interest here. If the obtained assertions show that the program has some undesired property, then there is a bug. One may expect that such assertions may also help in localising the bug. A work along these lines combining the declarative debugging with abstract interpretation, known as abstract debugging is reported in [6].

For automatic checking or generating it is necessary to have a formalised language of (a certain class of) assertions.

In our previous work [11, 12, 18] we used assertions for declarative diagnosis of Prolog programs. The assertions defined in that work were meta-level assertions describing sets of possibly non-ground atomic formulae, thus sets of constrained atoms under a specific restricted constraint language. So an assertion of that work corresponds to a set of assertions discussed here. In this paper we deal with a different (simpler) kind of assertions, referring directly to the semantic domain. In general case, the meta-level assertions would specify sets of constraints. Thus they would depend on the constraint language used. The usefulness of meta-level assertions for diagnosis of constraint program may be a subject of future work.

References

[1] J. Boye and J. Maluszynski. Two Aspects of Directional Types. In *Proc. of Int'l Conf. on Logic Programming '95*. MIT Press, 1995. See also: J. Boye and J. Maluszynski. Directional Types and the Annotation Method. J. Logic Programming, 1996. (To appear)

[2] J. Boye. *Directional Types in Logic Programming*, Ph.D. thesis no. 437, Linköping studies in science and technology, 1996.

[3] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Data-Flow Analysis of Prolog Programs with Extra-Logical Features. Technical Report CLIP2/95.0, Computer Science Dept., Technical U. Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, March 1995.

in this paper. A more careful study of this topic may lead to identification of new kinds of assertions useful in declarative diagnosis.

[4] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Data Analysis of Standard Prolog Programs. In *European Symposium on Programming*, Sweden, April 1996.

[5] F. Bueno, M. Garcia de la Banda, and M. Hermenegildo. The PLAI Abstract Interpretation System. Technical Report CLIP2/94.0, Computer Science Dept., Technical U.Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, February 1994.

[6] M. Comini, G. Levi and G. Vitiello. Declarative Diagnosis Revisited. In J. Lloyd, editor, *International Logic Programming Symposium.* MIT Press, 1995.

[7] P. Deransart. Proof methods of declarative properties of definite programs. Theoretical Computer Science, vol. 118, 1993.

[8] P. Deransart and J. Małuszyński. *A grammatical view on logic programming*. The MIT Press, 1993.

[9] W. Drabent. A Floyd-Hoare Method for Prolog. Post-conference workshop "Verification and Analysis of Logic Programs" at JICSLP '96 (Joint International Conference and Symposium on Logic Programming, Bonn).

[10] W. Drabent and J. Małuszyński. Inductive assertion method for logic programs. *Theoretical Computer Science*, 59:133–155, June 1988. Special issue with selected papers from TAPSOFT'87, Pisa.

[11] W. Drabent, S. Nadjm-Tehrani and J. Małuszyński. Algorithmic Debugging with Assertions. In: H. Abramson and M.H. Rogers (eds.) *Metaprogramming in Logic Programming*, 501–522. The MIT Press, 1989.

[12] W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. The Use of Assertions in Algorithmic Debugging. In *Proc. of Fifth Generation Computer Systems 88*, pages 573–581, 1988.

[13] *ECLiPSe 3.5 User Manual*. ECRC, Munich 1995.

[14] G. Ferrand. Error Diagnosis in Logic Programming. *JLP* vol. 4, 177–198, 1987.

[15] J. Jaffar and M. Maher. Constraint Logic Programming: a Survey. *JLP* vol. 19 and 20, 503–581, 1994.

[16] F. Le Berre and A. Tessier. Declarative incorrectness diagnosis in constraint logic programming. In P. Lucio, M. Martelli, and M. Navarro, editors, *Joint Conference on Declarative Programming APPIA-GULP-PRODE'96*, pages 379–391, 1996.

[17] J.W. Lloyd. Declarative Error Diagnosis. *New Generation Computing* 5, 133–154, 1987.

[18] S. Nadjm-Tehrani. Debugging Prolog Programs Declaratively. In *Proc. of Second Workshop on Meta-programming in Logic, META 90*, pages 137–155. Dept. of Computer Science K. U. Leuven, 1990.

[19] E.Y. Shapiro. *Algorithmic Program Debugging*. The MIT Press, 1982.

[20] Alexandre Tessier. Declarative debugging in constraint logic programming. In Joxan Jaffar, editor, *Asian Computing Science Conference*, volume 1179 of *Lecture Notes in Computer Science*, pages 64–73. Springer-Verlag, 1996.

[21] E. Vetillard. *Utilisation de Declarations en Programmation Logique avec Contraintes.* Ph.D. Thesis. Univ. Aix-Marseilles II, 1994