CP Debugging Tools

Task TWP1.1: Clarification of functionalities. Selection of the tools.

Deliverable: Draft report D.WP1.1.M1.1

Release: 7.3

July 10, 1997

A. Aggoun
F. Benhamou
F. Bueno
M. Carro
P. Deransart
W. Drabent
G. Ferrand
F. Goualard
M. Hermenegildo
C. Lai
J. Lloyd
J. Małuszyński
G. Puebla
A. Tessier

Contents

CP Debugging Tools

A. Aggoun* F. Benhamou† F. Bueno‡ M. Carro‡ P. Deransart§
W. Drabent¶ G. Ferrand† F. Goualard† M. Hermenegildo‡ C. Lai‖
J. Lloyd** J. Małuszyński†† G. Puebla‡ A. Tessier†

1 Introduction

The purpose of this Part 2 ("CP Debugging Tools") of deliverable D.WP1.1.M1.1 is to present the constraint debugging tools to be realized in the DiSCiPl project.

Debugging is needed when the behavior of the program at hand does not fulfill the user's expectations. The results of the questionnaires show (see D.WP1.1.M1.1-Part 1) that this happens when the answers produced are different from expected, or when some expected answers are missing. It shows also that a poor performance of the program is often a major problem; the time needed for computing answers may be too long for user's needs. Thus, programs may require

- correctness debugging, and
- performance debugging.

The tools developed in the DiSCiPl project are also intended to facilitate use of constraint programming for non-specialists (end-users), and therefore must offer powerful integrated tools. Such tools will include the most recent solutions derived from validation and abstract interpretation techniques.

D.WP1.1.M1.1-Part 1 also emphasized lack of high level debugging tools for existing constraint systems (most of them use improved "box-model" traces or display of dynamic constraint values). Enhanced debugging tools will take advantage of the clear separation between declarative and operational semantics of CLP, thus allowing high-level debugging facilities which do not refer to a particular constraint solver.

The tools will also offer the possibility to structure a CLP program by different means: by *predicates* as the clausal form of CLP allows to do, and by isolating on demand subsets of constraints. The later will be referred as *S-box* structuring. In fact, predicate definitions are a way

*Cosytec, Parc Club Orsay Universite, 4, rue Jean Rostand, 91893 Orsay Cedex, France. aggoun@cosytec.fr
†LIFO, University of Orléans, Rue Léonard de Vinci, B.P. 6759, 45067 Orléans Cedex 2, France. {Frederic.Benhamou,Frederic.Goualard,Gerard.Ferrand,Alexandre.Tessier}@lifo.univ-orleans.fr
‡Facultad de Informática, Universidad Politécnica de Madrid, 28660-Boadilla del Monte, Madrid, Spain. {bueno,mcarro,herme,german}@fi.upm.es
§INRIA-Rocquencourt, Projet LOCO, BP 105, F-78153 Le Chesnay Cedex, France. Pierre.Deransart@inria.fr
¶Institute of Computer Science, Polish Academy of Sciences. Poland. wdr@mimuw.edu.pl
‖PrologIA, Case 919, Parc Scient. et Techno. de Luminy, 163 Avenue de Luminy, F-13288 MARSEILLE Cedex 9. claude@prologianet.univ-mrs.fr
**University of Bristol, DCS, Tyndalls Avenue, Senate House, BS8 1TH, Bristol, UK. jwl@compsci.bristol.ac.uk
††Linköping University, Department of Computer and Information Science, Östergötland, S 581 83 Linköping, Sweden. jmz@ida.liu.se

to give the search-space a structure, as S-boxes are a way to put some structure on a constraint store and constraint propagation.

The integrated practical use of these paradigms (use of different semantics and program structuring) will be referred to as the *DiSCiPl programming methodology.* A *DiSCiPl debugging methodology* is associated with this programming methodology. Tools supporting this methodology will

- help program structuring,
- clearly distinguish between declarative/operational semantics,
- take advantage of the declarative semantics of CLP,
- support and simplify tracing,
- include advanced visualization tools,
- support run-time user interactions.

The DiSCiPl debugging methodology is therefore based on two fundamental aspects of the project:

- High-level declarative CLP language.
- Program and constraint structuring capabilities.

1.1 CP debugging

The constraint problems considered in the DiSCiPl project are expressed in constraint logic programming languages (CLP). This means that their description has two components: clauses (i.e., the logic programming component) or predicate definitions, and constraints (the constraint component). A CLP application is a combination of both components in a CLP program.

There are programs in which clauses dominate, others where constraints dominate, and intermediate situations. A typical situation of the first case is logic programming, or applications written in Prolog. All CLP platforms developed in the DiSCiPl project are based on ISO Prolog as underlying logical language.

A typical situation of the second case is a pure CSP problem specified by a set of constraints only. The usual intermediate situation corresponds to what is called CLP. A particular aspect of constraint problems is that no distinction necessarily is made a priori between program and data: data are coded into the program (usually as facts) or constraints.

Typical CLP programs include:

- Data.
- Constraints, knowledge (or knowledge bases) and methods (e.g., combination of predefined methods = use of global constraints).
- Strategy (e.g., labeling), description of heuristics.
- Data and solution display, user interfaces, …

Different kinds of errors arise when developing a CLP program. They are detected through *symptoms* which depend on the part of the program where they originate (here we refer to a program structure essentially based on predicates).

- Data.

  Error: mainly data inconsistency.

  Symptom: bad display of data, deviation w.r.t. predicate semantics.

- Constraints, knowledge, methods:

  Error: incorrect clause in program, bad control.

  Symptom: deviation w.r.t. predicate semantics, non-termination.

  Global constraints need specific methods. (Nothing defined yet).

- Strategy, heuristics description.

  Error: bad control (intervals) or bad strategy or heuristic (intervals, FD).

  Symptom: non-termination (intervals) or bad performance (intervals, FD)

- Data and solution display and User interfaces

  Error: incorrect clause in program, control.

  Symptom: deviation w.r.t. predicate semantics, non-termination.

Notice that a symptom is observed after some execution. It is therefore related to *one computation* (intended here as what happens after a CLP processor is activated with a goal). Another possibility is to study the CLP program itself, trying to find out whether or not there may exist error symptoms. Showing that there is no possibility of symptom is exactly a proof of correctness (no possibility of deviation). It means that the program is thus error free and also that the program behaves correctly (i.e. as expected) for *all computations.*

A general approach to debugging will thus combine efforts showing absence of symptom, and if there is some evidence that there exists some (either by establishing formally their existence, or as resulting from an incorrect execution) to use such information to localize in the program the origin or the reasons of the unexpected behavior. In the former case tools are related to all computations, as they are intended to debug "all computation". In the later case tools are related to "one computation", as they use a specific computation as starting point.

It must be observed that if incorrect behavior comes from the model and/or the programmed strategies, the error cannot be identified just at the level of some predicate or constraints. This is particularly true for performance debugging. The tools must allow thus to analyse efficiently very complex computations. Tools must also allow to re-do or modify computations at some point with the purpose of exploring different strategies.

A general approach to debugging will also try to take advantage of the declarative semantics of CLP as much as possible. It is in fact possible, using declarative semantics only, to show predicate correctness or to localize errors related to predicate definitions, or erroneous constraints (correctness debugging). This allows limiting the use of tools related to the operational semantics to trapping errors related to the operational semantics or search space (in particular, performance debugging).

The tools will be thus presented according to a classification which combines two parameters: declarative or operational semantics on one side, one computation (observed symptom) or all computations (absence or existence of symptom) on the other.

1.2 A Classification of Debugging Tools

The tools to be developed should be useful to a wide range of users. This includes *naïve users*, which are mainly concerned with a basic understanding of the operation of the constraint program and of correctness issues. However, such users are often also interested in a basic understanding of some performance issues. The tools should also be interesting to *advanced users*, which will be developing large applications, and which will be more concerned with performance issues and the operational details of the search. Finally, *system developers* represent yet another group of users, who are basically interested in studying the performance of internal algorithms. However, although the tools developed will probably also be of use to system developers, they are not perceived as the primary target of the project's tools.

Due to the variety of users and aspects apparent from the previous discussion, the range of possible tools for debugging of constraint programs is quite large. Many of the aspects to be dealt with are quite experimental, and the literature on the topic is relatively scarce. This implies some difficulty in defining a priori the characteristics of the debugging and visualization tools to be developed in a very precise way. Some of these aspects are user's requirements for debugging, controlling, understanding and improving constraint programs developed for complex and combinatorial use applications [SC95b].

As explained in the previous section, tools will be classified according to the following criteria:

- Semantics: *declarative* (D-based) or *operational* (LD-tree).
- Symptoms: *one computation* (observed symptom) or *all computations* (proved absence or existence of symptom, also called "abstract symptom").

	all comp.	1 comp.
Declarative sem.	A1	B1
Operational sem.	A2	B2

The tools are presented according to the combinations of these two paradigms.

A1: Declarative semantics, all computations:

Proof of declarative properties, *generation* and *static diagnosis*. Static diagnosis consists of localization in the program of erroneous clause by observation of failed or inconclusive proofs (of declarative properties).

Here we plan tools for both inferring and checking program properties, mainly for the purpose of correctness debugging. The properties will be expressed as assertions, the proposal focuses on the use of types.

A2: Operational semantics, all computations:

Proof of operational properties, *generation* and *static diagnosis*. Static diagnosis consists of localization in the program of erroneous clause by observation of failed or inconclusive proofs (of operational properties).

The difference with above tools A1 above comes only from the kind of considered properties (instead of being declarative and independent from the solvers, they are operational and related to the resolution or solver strategies).

It is worth observing that the difference between static diagnosis (of operational properties) and run-time checking, or "dynamic diagnosis", lies in the fact that deviations are analyzed for all computations or with one computation respectively.

B1: Declarative semantics, one computation:

Declarative diagnosis consists of localization in the program of erroneous clauses (by observation of declarative property violation).

Two kinds of errors are considered: incorrectness and insufficiency errors. Declarative diagnosis is a particular kind of searching an LD-tree; a proof tree used in incorrectness diagnosis can be seen as an abstraction of a branch of an LD-tree. In traditional approaches the LD tree traversed is hidden from the user, who has to concentrate on declarative aspects, answering queries chosen by the diagnoser. But a graphical interface may visualize the nodes of the tree connected with the queries of the diagnoser. A query may have the form of a constrained atom and constraint debugging may also be used to analyse it.

Assertions may be used to help answering queries.

B2: Operational semantics, one computation:

Visualization of search space: selective observation of dynamic behavior. These tools support investigation of a single test run, i.e., of the search space for some initial goal. A graphical tool for presentation of the search space to the user, mainly for purposes of performance debugging. The tool(s) should provide various means of focusing and abstraction, in order to cope with the, usually great, size of the search space. This will include as a special case functionalities present in box model Prolog debuggers.

Here we also include tools supporting specific strategies for selecting information from the search space.

Notice also that the language of assertions may be useful as a focusing tool for graphical presentation of the search space, for example for localization of certain nodes.

Run-time checking: localization in the program of the violation of a property at run-time. It may also be viewed as an auxiliary device for tools A2. Proper program error localisation will be called also *dynamic diagnosis*.

1.3 The DiSCiPl Debugging Methodology and Tools

We now present an overall design of the DiSCiPl debugging methodology and introduce a first description of the type of tools that will be needed to implement this methodology. We believe that an integrated environment incorporating the tools that we propose can be of great help in speeding up the code development process. The debugging environment that our proposal enforces is as depicted in Figure 1. The tools are grouped as follows:

- Static Validator: a module which at compile-time performs inference and checking of both dynamic and operational properties for all computations (A1 and A2). As a result of this process, specifications can have been validated, found erroneous (in which case symptoms are reported), or none of the previous. Symptoms of this nature relate to an "abstraction" of several possible computations and are referred to *abstract symptoms*.
- Static (or Abstract) Diagnoser: a module which performs diagnosis of abstract symptoms.
- Declarative and Dynamic Diagnosers: modules which perform localisation of program errors from symptoms found in one computation. Tools based on declarative semantics (B1) will be preferred to those based on operational semantics (B2) and used whenever possible.
- Performance Debugger: as performance of a program can only be understood operationally, this module will allow observing the operational run-time behavior of the program for one computation (B2).

The underlying debugging methodology is as follows. The first objective is, understandably, program correctness. Correctness is dealt with at a first stage at compile-time. Static diagnosis is

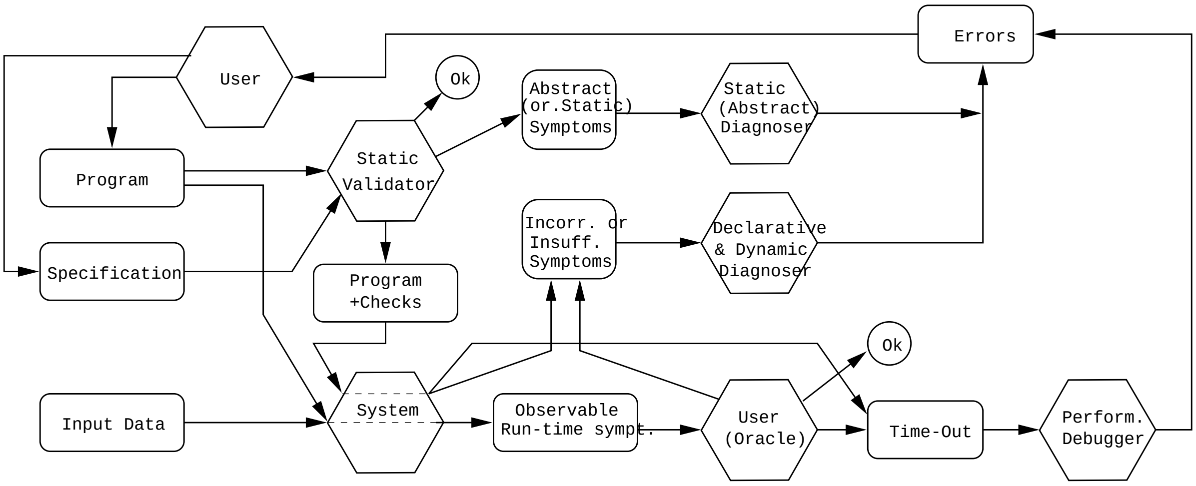

Figure 1: DisCiPl Debugging Methodolgy and Tools

performed to locate (and correct) as many errors as possible. The intention is to detect bugs as early as possible, i.e., during compilation or even editing. This can only be achieved by (semi-) automatic analysis of the (not necessarily completely developed) program in the presence of some (approximate) specifications. An example of such techniques is type checking, which has been proved to be useful for this purpose. Our approach puts a framework for working with properties that will be more general than classical type systems. Static diagnosis is performed by generating properties which are true for the program at hand and/or checking (proving, or trying to prove) the properties given as specifications for the program. When (abstract) symptoms are found, abstract diagnosis can be used to identify errors in the program that produce the symptom.

Unfortunately, the fact that no more abstract symptoms are detected does not imply that no bugs are left in the program. In fact static diagnosis is incomplete in practice. At this point, program testing in the classical sense has to be performed. However, the methodology also aids the programmer in this respect: programs will be automatically or manually annotated with run-time checks for those requirements which analysis has not been able to prove statically. This will allow detecting further violations of the specifications during program execution on sample input data. The result is detection of symptoms of incorrectness or insufficiency. Diagnosis of these symptoms can then be used to identify the errors in the program that produce the symptom. Since run-time tests introduce relevant overhead in program execution, the annotation with run-time checks is performed only upon user's request and possibly temporarily, i.e., only until the behavior of the program can be assumed to be correct.

Once correctness is sufficiently established, it is time for performance debugging. If the requirements on the program include issues related to performance, then symptoms can relate to efficiency as well as correctness. The errors causing the former can be identified from a dynamic performance diagnosis session, which will be aimed at identifying the "error" in the program that is causing the performance problem.

To support the above mentioned activities we may need the following components:

- Program analyzers (or generators): compute properties which hold for the program at hand. Abstract interpretation techniques will often be used with this purpose.
- Provers and static comparators: compare the user's requirements with the information known about the program at hand. This information can have been generated by analysis, or may have to be extracted by the prover directly from the program. Provers can produce three different results: (1) the requirement is verified, (2) it does not hold, or (3) it

cannot be decided.

- Run-time check annotator: it is in charge of introducing run-time tests into the program which would make sure that the requirements hold, so that possible violations can be detected during execution of the program.
- Diagnosers for abstract symptoms: they try to localize the program construct responsible for an abstract symptom of error. They will locate possible error sources.
- Diagnosers for correctness and insufficiency symptoms: they are used in order to underpin all subsequent bugs which appear during program testing and execution.
- Performance diagnosers/visualizers: reconstruct the program behavior during execution of a particular computation, in order to allow the user to identify sources of correctness and/or efficiency "insatisfactions".

During the project, a series of debugging tools which combine several basic tools (search tree visualization, provers, generators, static, declarative or dynamic diagnosers) will be developed. The better understood will be developed by the industrial partners, and the more experimental will be prototyped by the university partners. Four CLP platforms will be upgraded with such debugging environments:

- CLP(FD), CLP(FD + Interval),
- CIAO,
- CHIP V5 (beta version),
- Prolog IV (beta version).

1.4 Organization of the report

This report describes the tools to be developed by the project as precisely as possible: functionalities of tools are explained, then the platforms on which several tools will be combined are briefly described. It is not possible at the present state of the project to give a precise description of the final advanced tools, because their efficient integration is the goal of the project itself and needs further investigation. However, it is possible, from this report, to have a clear picture of the main functionalities of the final tools.

The report is organized as follows.

- Semantical background (Section 2): a short presentation of the different CLP semantics the tools refer to (declarative, operational, abstract constraint solving). It helps to understand the level of action of each tool.
- Discussion and presentation of four families of tools (Sections 5,4,3).
- Selection of the tools to be developed in the current project on different academic and industrial CLP platforms (Section 6). This includes intermediate modules used to build integrated tools and short platform descriptions.

2 Semantic background

This section outlines a few semantic notions needed for presentation of the proposed tools.

As stated above, debugging is required if the program behaves differently than expected. Commonly the user would notice that the system delivers unexpected answers, that some expected answers are missing, or that the waiting time for some answers is unacceptably long.

The answers delivered by the system result from computations. Thus clarification of the discrepancies would generally require analysis of the particular computation where the unexpected behavior has been noticed. For this we need an abstract model of the computation, suitable as a basis for presentation to the user and for analysis. We adopt for this purpose the notion of LD-tree, or search tree, originating from logic programming. Most of the tools proposed have a connection to LD trees. In particular, various visualization tools refer directly to LD-trees.

For a large class of CLP programs, answers can be characterized independently of computations, by a declarative semantics. Such semantics provides thus a basis for comparing the user's expectations with the answers of the actual program. In particular, the user may be interested in whether or not all answers of a program for a given class of queries have a certain property, for example if they belong to a certain type. The program analysis tools proposed in this report address such questions.

Another difficulty comes from the size of the constraint store associated to every computation step. The abstract constraint solver approach (Task T.WP2.3) is a basic semantical approach for analysis of constraint store structure and constraint propagation.

The rest of this section discusses:

- A simple example program, used in the sequel for illustration of the outlined notions.
- The D-based declarative semantics (where D is the constraint domain) to be used in program analysis tools.
- The notion of LD-tree, or search tree, referred to by the majority of the proposed tools.
- Some basics on abstract constraint solving and constraint store structuring.

2.1 Example

A very small example will be used to illustrate semantics.

The following programs specify integer scalar product, i.e.: they define the ternary relation p, $p(p1, p2, p3)$ where $p2$ and $p3$ are vectors with integer coordinates, and

$$p1 = p2 \times p3 \ (\textit{scalar product})$$

First version : CLP(I) (*Integer arithmetics*).

```
Pr1  c11: p( N+X*P, [X|S], [P|R]) :- p(N, S, R).
      c12: p(0,[],[]).
```

Second version: CLP(PA) (*Presburger arithmetics*)

```
Pr2  c21: p(N+L, [X+1|S], [L|R]) :-  p(N, [X|S], [L|R]).
     c22: p(N, [0|S], [P|R])     :-  p(N, S, R).
     c23: p(0, [], []).
```

The answers of the program for the query

```
p(10, L, [5,2]).
```

may be linked to some practical problems. For example they may show how to pay 10 money units with coins of 5 and 2, or how to cover a segment of length 10 by segments of length 5 and 2.

In some CLP languages used in the project the programs could be represented as follows.

In Prolog IV (intervals):

```
p(N.+.X.*.P,[X|S],[P|R]) :-  ge(X,0),p(N,S,R).
p(0,[],[]).

% Interval  Solver
% Query: p(10,L,[5,2]), intsplit(L).
% Answers: L = [0,5];
%          L = [2,0].
```

In CHIP V5 (Rationals and Finite Domains):

```
p(N+X*P, [X|S], [P|R]) :-  p(N, S, R).
p(0, [], []).

p1(P1, P2, P3) :-  p(Scalar, P2, P3), Scalar ^= P1.

% Rational Solver
% Query: P2 = [5, 2], P3 = [X1, X2], p1(10, P2, P3) ?
% Answer: P3 = [2 - (2/5)*X2, X2]

p2(P1, P2, P3) :-  P3 :: 0..10, p(Scalar, P2, P3), Scalar #= P1,
                   labeling(P3).

labeling([]).
labeling([X|Y]) :-  indomain(X), labeling(Y).


% Finite Domain Solver
% Query: P2 = [5, 2], P3 = [X1, X2], p2(10, P2, P3),
% Answer: P3 = [0, 5]
%         P3 = [2, 0]
```

2.2 Domain based declarative semantics

The D-based semantics (where D is a constraint domain) characterizes all possible answers of the program without referring to the way of its execution. It is fully described in [DM93, JM94, BFL$^+$95, Fag96]. This section gives only a brief outline illustrated with examples. Some of the program analysis tools to be developed in the project refer to this semantics.

We consider constraint programs over a given constraint domain D. The D-based semantics associates with a given program a (usually infinite) set of D-*atoms* of the form $p(v_1, ..., v_n)$ where p is an n-ary predicate and $v_1, ..., v_n$ are elements of D, or data structures built out of such elements, e.g., lists. This set is called the *least D-model* of the program and can be defined in terms of *proof trees*, as sketched below.

The variables of the program range over D (and the data structures over D). Take a valuation of the variables of a clause that satisfies the constraints of this clause. Under this valuation a clause may be used as a rule consisting of D-atoms. For the program `Pr2`, $p(10, [2, 0], [5, 2])$:– $p(5, [1, 0], [5, 2])$ is an example of such a rule. The clauses without body atoms give rise to facts. The rules can be represented by trees and can be combined in a natural way as illustrated by the following example. A tree whose frontier nodes correspond to facts (and thus cannot be further extended) is called a *proof tree*. The D-based semantics consists of the D-atoms labeling the nodes of all proof trees.

The following example shows a proof tree of Program `Pr2`.

```
. c21,  p(10, [2,0], [5,2])
|
. c21,  p(5, [1,0], [5,2])
|
. c22,  p(0, [0,0], [5,2])
|
. c22,  p(0, [0], [2])
|
. c23,  p(0, [], [])
```

Each node of the tree is labeled by the clause name used to extend it and the corresponding D-valued atom.

Domain based declarative semantics describes the relations defined by the program. The question is whether these relations correspond to user expectations, that is, whether they have some expected properties.

For example, the user may expect that the ternary relation p defined by the program `Pr1` has the following property:

For all $(p1, p2, p3)$ in p, *if* $p2$ *and* $p3$ *are lists of integers, then* $p1$ *is an integer*, or

For all $(p1, p2, p3)$ in p, *if* $p1$ *is an integer and* $p3$ *is a list of integers, then* $p2$ *is a list of integers.*
A formal expression of these properties would resemble a type declaration.

Another expected property could be: For all $(p1, p2, p3)$ in p, *if* $p2$ *and* $p3$ *are integer vectors then* $p1$ *is their scalar product* $p2 \times p3$.

Given a statement S of a property the user may be interested, whether or not:

- Each element of the domain based semantics of P has this property, in which case P is said to be *partially correct* w.r.t. S. This concerns for example type checking of a program w.r.t. to a given type declaration.
- Every element with property S is included in the semantics of P, in which case P is said to be *complete* w.r.t. S. For example the user may expect that the semantics of program P contains all elements of the form $\mathtt{p}(\vec{\mathtt{p2}} \times \vec{\mathtt{p3}}, \vec{\mathtt{p2}}, \vec{\mathtt{p3}})$. Intuitively, completeness means that all answers expected by the user can be delivered by the program.

Such a proof tree may be obtained from a *skeleton*. A skeleton is a tree built out of uninstantiated clauses. With every node of such tree it is possible to associate the constraints that must be satisfied. The tree from previous example can be obtained from the following skeleton with constraints (instances of clauses are renamed).

```
                                                      . goal
                  {N0=10, Q0=[5,2]}                   |
                                                      . c21,  p(N0, P0, Q0)
{N0=N1+L1,P0=[X1+1|S1],Q0=[L1|R1],P1=[X1|S1],Q1=[L1|R1]} |
                                                      . c21,  p(N1, P1, Q1)
{N1=N2+L2,P1=[X2+1|S2],Q1=[L2|R2],P2=[X2|S2],Q2=[L2|R2]} |
                                                      . c22,  p(N2, P2, Q2)
                  {N2=N3, P2=[0|P3], Q2=[L3|Q3]}      |
                                                      . c22,  p(N3, P3, Q3)
                  {N3=N4, P3=[0|P4], Q3=[L4|Q4]}      |
                                                      . c23,  p(N4, P4, Q4)
                  {N4=0, P4=[], Q4=[]}                |
                                                      .
```

The constraints are obtained from "normalised" clauses of `Pr2`:

```
c21: p(N0, P0, Q0) :- {N0=N1+L1, P0=[X1+1|S1], Q0=[L1|R1],
                              P1=[X1|S1], Q1=[L1|R1]}, p(N1, P1, Q1).
c22: p(N0, P0, Q0) :- {N0=N1, P0=[0|P1], Q0=[L1|Q1]}, p(N1, P1, Q1).
c33: p(N0, P0, Q0) :- {N0=0, P0=[], Q0=[]}.
```

2.3 Operational semantics: LD-trees

The notion of LD-tree (SLD-tree with Prolog Computation rule), or search space, for a CLP program and goal is an adaptation of a similar notion known for logic programs (see for example [DM93]). It is an abstraction of a computation performed for a given goal. Rigorous presentations of various notions of search space can be found for example in [BFL+95, Fag96] and (in terms of generalised and-or trees) in [dlBHB+96, dlBH93].

Nodes of LD-trees represent states of the computation, and correspond to procedure calls. Each node is labelled by a sequence of atoms and by a constraint. The labels of the edges show the clauses of the program used for resolution at a given step of computation. The impact of the constraint solver involved is represented by the form of the constraints labelling the nodes. The fail nodes are those where the constraint solver detects unsatisfiability of the node constraints. The success nodes are those labelled with the empty sequence of atoms. The corresponding constraint determines then the computed answer for the initial goal. Thus, LD-trees have *failed*, *success* and *infinite branches.*

In computations of CLP(FD) a distinction is made between Prolog computation phase generating constraints, and the labelling phase where solutions of the generated constraints are enumerated. However, on the abstraction level of LD-tree no distinction need to be made between choice points of the Prolog computation and choice points of labelling. The labelling strategy can itself be described as a CLP program, as shown in the CHIP version of `Pr1` for finite domains.

The concept of LD-tree is illustrated by the following example concerning the program `Pr2`.

```
                            p(10,[A,B],[5,2])
                                    |
                    ----------------------------------------------------
  {10=N1+5,A=X1+1}  | c21                         {A=0}  | c22           | c23
           p(N1, [X1,B], [5,2]){N1=5}             p(10, [B], [2])       fail
                    |                                    |
                 --------------------      ---------------------------
{5=N2+5,X1=X2+1} | c21       | c22  | c23  | c21  {B=0} | c22          | c23
   p(N2,[X2,B],[5,2]){N2=0} fail   fail     .      p(10,[],[])      fail
                 |                          .            |
    -------------------------               .      -----------
    | c21        | c22      | c23           .      |    |    |
  fail      p(0,[B],[2]    fail             .     fail fail fail
(0 = N3+5)       |                          .
     ---------------------                  .
     | c21 {B=0} | c22   | c23              .
    fail     p(0,[],[]) fail                .
                 |                    one success
        -----------------           (5 times c21, then c23)
        | c21    | c22  | c23
       fail     fail   success
```

Search for all answers for a given goal corresponds to complete traversal of the LD-tree in the depth-first left-to-right order. The LD-semantics is essential for debugging as it abstracts useful information about the computations, which can thus be presented to the user and compared with his expectations:

- The LD-semantics gives a basis for studying invariants of program execution. Invariants may be associated with predicates. For instance a call invariant for p is a property of arguments of p satisfied whenever p is selected for resolution in a class of LD-trees. For example, for the Program `Pr1` and the last query from section 2.1, *"the first argument of* `p` *is an unbound variable"* is a call invariant. Other kinds of invariants may be of interest, for instance describing the arguments of a predicate both at its calls and successes.
- The LD-tree determines all answers for a given goal. Thus, it can be a basis for querying the user concerning correctness and completeness of the answers. This observation is exploited by the declarative debugging techniques which localise the error by a querying process controlled by the form of the LD-tree and by user answers.
- The LD-tree represents complete search for answers for a given goal. Its visualisation may provide necessary information for performance debugging.
- The nodes of LD-tree are abstractions of computational states and provide information about relations between variables, about the state of constraints etc. Further abstraction may be needed to present this information to the user. The proposed visualisation tools may help to understand some properties of the store and its evolution.
- The LD tree shows the constraints produced by the constraint solver, thus can be a basis for studying local constraint propagation.

2.4 Abstract constraint solving and S-boxes

One of the aim of the tools is to help programmers in their debugging activities by visualizing and manipulating the constraint store. Basically, the corresponding constraint store should be associated to each node of the LD-tree. Understanding the state of the store at some computation step will be referred as *constraint debugging*.

Unfortunately, the specific processing of constraint debugging presents a number of difficulties.

To understand the different aspects of constraint debugging, it seems necessary to inspect with care the main different classes of solving techniques and the corresponding information which is to be found in the store. The two main different aspects of constraint solving concern:

1. reduction of variables' domains,

2. symbolic manipulations of formulas.

In the case of finite domain and interval constraints (the main focus of the project), we are mainly concerned with reduction of variables' domains but as will be developed, there are several reasons to take symbolic manipulations of formulas into account. The work in progress in task T.WP2.3 let us think that both types of constraint solving methods can be unified, at least at a theoretical level. However, if specific investigations of parts of the store concerned with complete/symbolic solving algorithms (e.g., Simplex/Gaussian elimination, complete Boolean solvers, etc.) are required for debugging purposes, crucial differences between the two aspects must be taken into account. We will come back on this matter in the following sections.

More precisely, as shown in [Ben96] and as will be fully developped in task T.WP2.3, a possible alternative representation of constraint stores is to define a function which associates to every set of formulas a set of operators, called constraint narrowing operators, defined over Cartesian products of subsets of the constraint domain. For example, to the constraint $z = x + y$ is associated an operator which takes the Cartesian product of three intervals representing the domains for the variables x, y and z (let us call such Cartesian products *boxes*) and computes the smallest box (with respect to a certain floating point representation of numbers) containing the intersection of the initial box and of the relation $add = \{z \in \mathbb{R} \mid x + y = z\}$. Then, the overall computation over the store is expressed in terms of fixed points of the constraint narrowing operators. For example, in most interval-based system [BO97, OV93, OB93, Gou95, Col96, BT95], a floating point interval (that is a real-valued interval whose bounds are floating point numbers) is associated to every variable and constraints are decomposed in primitive constraints (fresh variables are introduced when necessary). These constraints are related one to each other by variable sharing and form a network. At each computation step (node of the search tree) constraints are added to the network (tell operations) which is stabilized, leading to variables' domain contractions and possibly inconsistency detection. Note that *no symbolic transformation of formulas representing constraints is involved in this process.*

In this last representation, the most relevant information at hand for debugging purposes is the newly computed set of variables' domains. Examination of this state (which can be conveniently formalized as an "hyper-box") will lead to the discovery of *error symptoms* in a very broad sense. To state it roughly, in the course of the debugging process, the state of the variables' domains is considered correct by the programmer before the processing of the computation step and incorrect after the domain reduction process (unexpected inconsistency, no reduction, too much reduction, etc.). Such unexpected behavior may have a number of possible causes:

- the program rule which was used at the corresponding node is incorrect in the sense that the constraint set added to the store is different from what was expected by the programmer (addition of wrong equalities from the unification process, misspelled variables, mathematical modeling of the sub-problem is erroneous),

- the above problem was encountered in previous nodes but remained undetected (no immediate "visible" consequences, only visible consequences are on untracked variables, etc.)

- everything is correct, but the incompleteness of the solver does not allow for sufficient domain contraction, thus leading to unexpected behavior (for example in most interval-based systems, adding the constraints $x - x = y, x \in [0,1], y \in [0,1]$ to an empty store does not lead to any domain modification for variable y.

Possible courses of action to track the cause(s) of the problem involves:

- examination of the constraint store,
- dynamic examination of the fixed point computation and its intermediate consequences on the variables' domains,
- examination of previous domain modifications (history)

The first difficulties come from the size, complexity and code-independence of the current store.

3 Tools related to all computations (A1, A2)

As tools A1 and A2 are very similar and differ only on the kind of properties which are considered (declarative or operational) they are presented together. Their implementation however may differ as they refer to different semantics.

The tools discussed in Sections 5 and 4 facilitate analysis of a single computation while searching for the reasons of unexpected behaviour. This approach is in general not practical to prove correctness or efficiency of a program as the set of computations is usually infinite or too large to study in reasonable time. If fact, the tools presented in Sections 5 and 4 are often used once a computation which does not satisfy the (either correctness or performance) requirements is found. Finding such a computation (a symptom) may be quite difficult since this may require exhaustive testing of the program.

The tools proposed in this section address a more general question: whether or not *all* computations of a given program satisfy a given requirement. Note that for this kind of task it is not needed to find a symptom in advance.

The proposed tools related to all computations will mainly be concerned with correctness debugging. It is possible that static analysis techniques could also be applied to performance debugging. This would require some kind of global cost analysis for all computations of the program. This may be a subject of future research, but in the present state of the art it is rather difficult to propose a tool for performance debugging based on global analysis techniques.

3.1 A general view

As sketched above, ideally we would like to prove that a given program satisfies a requirement concerning every answer, or every computation (in the considered class). In that case the requirement is *verified*. Requirements may refer to both declarative and operational semantics, as well as performance criteria.

Independently of the technique used, *verification* can be seen as a process which receives a program and a set of requirements as input and returns, for each requirement, one of the following three values:

- **proved:** the requirement holds, i.e., it is verified. This is the most favourable situation.
- **disproved:** the requirement is proved not to hold. This is a “symptom” that something is wrong in the program and diagnosis should start in order to correct it.

- **unknown:** the verifier is unable to prove nor disprove the requirement. This may be because the requirement is not fulfilled or because the verifier is not complete.

Figure 1 contains two boxes which refer to tools related to all computations, the static validator and the diagnoser for abstract symptoms. In turn, the static validator can be split into a series of tools which deal with all computations:

- **Analysers**: tools which infer properties from a program. Take as input a program. Give as output a set of properties.
- **Provers**: tools which attempt to prove that a certain property holds. Take as input a program, an intended property (requirement) and possibly a set of properties which hold in the program. Give as output the result "proved", "disproved", or "unknown".
- **Annotator for Run-time Tests**: this tool takes a program and a set of requirement and returns a new program which contains run-time tests which will inform if (some of) the requirements do not hold at run-time.
- **Translators**: it may be the case that the intended properties of the program are not in the same domain of properties which the prover understands. In such cases it is needed to translate properties expressed in different domains. A typical example of different domains are the concrete and abstract domains used in abstract interpretation.

3.2 Expressing Properties

3.2.1 Assertions

Both the requirements stated by the user and the properties inferred by the analysers will be expressed by means of *assertions*. Thus, defining the language of assertions is a first objective. This language will allow two way communication between the tools and the user. As the requirements may concern various kinds of semantics the assertions must allow expressing properties which refer to different semantics. For example, we may need type declarations referring to the declarative semantics (i.e., D-model semantics) as well as mode declarations referring to the operational semantics (i.e., LD-trees).

We plan to develop a general assertion language which will be independent of the particular being used, specific domains, and specific semantics. For example, we may need assertions expressing the range or the cardinality of the finite domain of an argument of a predicate in the D-model semantics of a finite domain constraint program. We may need other assertions describing the intervals of arguments of a predicate at the success of the narrowing phase of a program based on the interval domain.

While assertions provide a basis for provers, analysers and comparators, they can also be used in tools concerning a single computation, for example for expressing information about the actual constraint store, or for selecting some particular data from the actual LD-tree. Clarification of the potential use of assertions in all proposed tools is a subject of a separate research task.

3.2.2 Abstract interpretation

Central for several tools proposed in this section will will be *abstract interpretation* [CC77]. It is a standard method of inferring program properties. For automatic inference of properties by abstract interpretation one has to define a priori a restricted class of properties, called the *abstract domain*, with a special ordering (cpo).

A typical example is the problem of *definiteness* analysis for a given constraint program and query, discussed in [dlBH93]. Roughly speaking, the question is: given a program and a generic query, which variables of the query will have their domains restricted to a single value in any answer obtained. A generic query can be seen as an atom with all arguments being distinct variables, with additional information about which of the variables have their domains restricted to a single value. For example, for the scalar product program $Pr1$ with a generic query $p1(X, Y, Z)$, where Y and Z are definite, the variable X will have its domain restricted to a single value in any answer obtained.

This kind of assertion associates a variable, e.g., X with a set of variables, e.g., $\{Y, Z\}$, to state that a property holds for X iff it holds for every variable in the set. Such pairs are building blocks for the abstract domain defined in [dlBH93] which is used by an automatic system. The system can infer correct definiteness information for any given program and query. The inference is a kind of fixpoint computation using the operations induced by the ordering of the domain.

There exist generic abstract interpretation tools for CLP which can take the abstract domain as a parameter and thus can analyse different kinds of properties. We propose to develop such an abstract interpreter as a basic module to be used for various purposes as described below.

3.3 Tools

3.3.1 Analysers

An analyser infers properties of programs. The kind of properties we are interested in may be very different for different requirements. With the use of abstract interpretation techniques the properties inferred are restricted to those defined by the underlying abstract domain. A very important task for achieving flexibility is thus development of an efficient generic abstract interpreter, where the abstract domain can be taken as parameter. One such generic analysis framework for CLP has been proposed in [dlBHB$^+$96]. Building on such a framework, several domains of analysis can be integrated.

Properties of the program inferred by static analysis may be very informative and may be used for debugging purposes. The programmer may read the analysis information and compare with what he expected. However, this may be a tedious task as analysis information may be hard to interpret and comparisons with the requirements may be difficult. Thus we aim at performing such comparison automatically.

We will give high priority to building dedicated type analysers for CLP(FD) and CLP(intervals).

A very important feature of analysers for debugging is that the analysis algorithm be incremental. This is because during the debugging phase of the life-cycle, the program usually gets small modifications and is checked again for inconsistency with the requirements. In such a case a non incremental analyser repeats much of the work performed previously. We will thus aim at obtaining analysis tools which are incremental, using the algorithms presented in [HPMS95] as an starting point.

3.3.2 Provers

Provers try to conclude that a given requirement holds or does not hold. In addition to the program, they may receive a set of properties inferred by program analysis. Clearly, if the abstract domain for analysis is appropriate, analysis information may be very valuable and make proving of the requirements relatively simple. However, if no analysis information is given, proving requirements is in general a rather difficult task.

Three approaches to proving requirements will be investigated:

- Use abstract interpretation for inferring properties about the program and them compare them with the intended ones. In this case, the difficulty lies in the analysis process, while this comparison may simply involve a few operations in the abstract domain.
- The use of abstract interpretation techniques for inference and checking of program properties. A more efficient alternative is to give the properties given as requirements as starting point for fixpoint computation during analysis. If requirements are given for all predicates and the analysis is not multivariant, only one step of fixpoint computation is needed. If a fixpoint is reached, the requirements are proved. If the new value computed in this step is incompatible with the one in the requirement, then the requirement is disproved. Otherwise it is unknown.
- The use of dedicated proof techniques. In this situation the prover does not receive any analysis information and has to prove that the requirements hold by checking the program code. Dedicated provers are in general hard to implement, and not easy to use. An important exception are type checkers, discussed in more detail below.

An important kind of properties we may be interested in requiring and proving is *types*. Type descriptions are usually not too complex for the user to give and may allow detecting many bugs at compile-time. Types may be defined as an abstract domain. Standard interpretation techniques can then be applied for type inference or checking taking as a starting point a type information provided by the user.

We propose to follow the approach to abstract-interpretation-based static validation pioneered by the CIAO system, in which user requirements are *optional* and can be *partially specified*. An advantage of this approach is that missing type information can be inferred, and in the extreme, when no information is provided by the user all types are inferred by the system. A starting point for development of a tool based on this approach may be the previous work done for logic programs, e.g., [JB92] or [SG95]. The type system proposed therein should then be extended to the constraint domain in question. The underlying analyzer of the CIAO system, PLAI, supports this kind of partial specification of information [BCHP96].

In contrast, many programming languages enforce a strict type discipline (i.e., stating the types of all predicates). This has the advantage that the presence of a complete type specification of the program can facilitate early discovery of some errors. In the logic programming field this has been confirmed by the experience with the type checker for the language Goedel [HL94]. However, most commercial constraint programming languages do not enforce any type discipline at all, and the resulting liberty during programming, coupled is instrumental in rapid-prototyping, which has been identified as one of the strongest points of logic and constraint programming languages, is preferred by many users. We feel that this justifies the use of the CIAO system's intermediate approach outlined above: allowing a very general form of assertions concerning the program, including descriptive types, but in a completely optional way. As pointed out above, stated assertions are checked, while non-stated assertions are generated automatically.

A special case is where requirements are given completely, and also for some type systems, since then a dedicated prover is not too complex. In this case type checking reduces to verification of a few conditions for each program clause and can be more efficient than fully fledged abstract interpretation. The starting point for development of a tool based on this approach may be the previous work done for logic programs [BM97] and for a subset of Prolog III [Vet94].

3.3.3 Translators

When using abstract interpretation for program analysis, the properties about the program will be expressed in terms of the abstract domain. However, the requirements will often be expressed in other domains (for example in a subset of the CLP language used). For such cases it must be studied which are the conclusions which can still be derived about the requirements when the prover has to translate them before checking whether they are proved or disproved (or none). Note also that if the user requirements are not expressed in the language of the abstract domain, a previous translation phase should be introduced which can introduce some loss of accuracy in the comparison.

3.3.4 Diagnosers for Abstract Symptoms (Static Diagnosis)

The aim of this tool is to localise the program construct responsible for the failure of a proof. Usually the proof procedures, like type checkers, perform separate tests for different program clauses. Thus the clauses where the tests fail are candidates for error localisation.

The requirement which is proved not to hold may be considered a "static" symptom. A static symptom ensures existence of an incorrectness or insufficiency symptom in some computations. The static diagnosis starting from a static symptom may then use algorithms related to those of declarative diagnosis in order to identify the program construct responsible for the symptom. Development of a tool based on this idea would require:

- The study of the theoretical aspects involved (starting from [BDD$^+$97], [BCHP96] and the work on "abstract debugging" [CLMV96]).
- Developing methods for localising the program fragment responsible for the discrepancy between the desired property and the result of the analysis.

The "don't know" failure of the proof provides no information about existence (or non-existence) of run-time symptoms. Both cases are possible. However, if the prover checks local conditions for every clause, the run-time symptoms, if any, can only be caused by the clauses where the local tests failed. So the static diagnosis can also be applied in this case, with the difference that the localised fragments of the programs are not guaranteed to be erroneous. The information obtained may be used for inclusion of dynamic tests, to check in run-time whether the requirements are violated, or not.

For simple properties, such as types, static diagnosis seems to be simpler than dynamic diagnosis. Another advantage of static diagnosis is that it can be completely performed at compile-time as it does not require the knowledge of input values. Clearly, the more bugs get corrected at compile-time, the less bugs will remain when testing or running the program.

4 Tools related to declarative semantics and a single computation: declarative diagnosis (B1)

Declarative diagnosis consists of localization in the program of erroneous clauses by observation of declarative properties violation.

If a "program" consists of thousands of constraints and variables without more structure no tool may be able to help in finding easily a missing or wrong constraint. But if this set is produced by a structured program (say recursive clauses with few constraints stated in each of them), then *declarative diagnosis* by helping to explore each instance of these clauses used to put together the constraints (which lead to a wrong solution) may allow bug localization.

The starting point of a declarative diagnoser is a computation producing a result which is considered as incorrect. Since there is a result, it is not an infinite computation. An incorrect result is called a *symptom*. This notion of *symptom* depends on some *expected properties* of the program, so a symptom is a result which is *not expected*. However the notion of expected properties may be more general than a complete specification of the program semantics. From a conceptual viewpoint we have only to presuppose an *oracle* which is able to decide that a result is expected or not. In practice the presentation of a result can be very intricate and difficult to use in practice. However, in practice the oracle can be embodied by other means (for example *assertions*) and the expected properties can be defined by using an abstract (approximate, graphical, ...) view of the computed result.

4.1 Principles

The declarative diagnosis scheme is the following. The symptom is the unexpected result of a finite computation. We consider the set of computation steps of this finite computation. Then the symptom is the result of the last computation step. But we can associate a result to each computation step. Some of them can also be considered as symptoms (because they are not expected). Let us assume an order over the set of computation step (the computation steps we consider are for example the nodes of the proof tree or the nodes of the LD-tree). This order is well-founded because of the finiteness of the computation. This order provides thus a notion of *minimal symptom*.

Once a minimal symptom is localized, we associate with it a small piece of erroneous code responsible of its occurrence. This piece of code is called an *error*. A suitable definition of the notion of error is possible only when the well-founded order over the computation steps is correctly chosen (i.e., the notion of minimal symptom has some sense w.r.t. the program).

Declarative means that the user has no need to consider the computational behavior of the constraints program; he needs only knowledge of the expected properties of the program (thanks to the properties of *confluence* (independence of the computation rule) and *compositionality* in the case of CLP). Confluence is basic to define notions which are *declarative* that is to say which do not depend on a particular computational behavior.

Based on the observation that most of the behavior of constraint programs can be understood declaratively, the declarative diagnoser performs a dialog with the *oracle* (the expected properties), asking queries about computation steps, without referring to any operational semantics. This allows the user to correct errors in the program without having to deal with control aspects, which are hidden in the diagnosis algorithm.

In fact because of the relational nature of CLP languages we have to split the notion of (finite) computation into two notions. That is to say that we have to split the notion of result in two notions.

A *goal* being given, there is a first notion of result which is a *computed answer constraint*, the computation being a *success derivation*. This is a *first level of computation*. In the formal logical semantics for CLP the relation between the goal $\leftarrow g$ and the computed answer constraint c is formalized by using the implication $c \rightarrow g$. [1]

But there is a *second level* of computation that is to say another notion of finite computation which is represented by a *finite LD-tree*. Now if $c_1, \cdots, c_n$ are all the computed answer constraints of this finite LD-tree, their relation with the goal $\leftarrow g$ is formalized by the implication $g \rightarrow c_1 \vee \cdots \vee c_n$ (if $n = 0$ (*finite failure*) this implication amounts to $\neg g$). To be more formal, $g \rightarrow c_1 \vee \cdots \vee c_n$ occurs along with the *completion* of the program and to be more precise with its *only if* part (while at the first level $c \rightarrow g$ occurs along with its *if* part, that is the program itself). [2]

[1] Even from a purely operational viewpoint we can consider that $c \rightarrow g$ is computed.

[2] Even from a purely operational viewpoint we can consider that $g \rightarrow c_1 \vee \cdots \vee c_n$ is computed at this second level of computation.

These remarks motivate that we call *positive* the first operational level and *negative* the second one.

A symptom at the *positive level* will be called a *positive symptom*. To say that $c \rightarrow g$ is a *positive symptom* is an abstract way to say that c is a *wrong answer* to $\leftarrow g$. If the expected semantics is defined in a logical framework with respect to an *intended interpretation*, $c \rightarrow g$ is not true in this intended interpretation.

A symptom at the *negative level* will be called a *negative symptom*. To say that $g \rightarrow c_1 \vee \cdots \vee c_n$ is a *negative symptom* is an abstract way to say that there is *not enough answers* to $\leftarrow g$, there are *missing answers*, g is *not covered* by $c_1, \cdots, c_n$. If the expected semantics is defined in a logical framework with respect to an *intended interpretation*, then $g \rightarrow c_1 \vee \cdots \vee c_n$ is not true in this intended interpretation.

The diagnosis algorithm follows the computation, but a lot of computation steps can be skipped according to the answers to previous questions. That is, the diagnoser follows the straight path from the computed symptom toward the minimal symptom removing useless search.

The tool(s) required by a declarative diagnoser is an interface between the diagnoser and the user. For example it may be a graphical interface to show a computation step to the user in order she answers if it is expected. But the interface may also use an assertion language to express declarative properties of the program.

4.2 Diagnosis of incorrectness (wrong answer)

Type of symptom:

- positive symptom (wrong answer): $c \rightarrow a$ unexpected (a solution of the constraint c which should not be in the atom a)

Type of error:

- positive incorrectness: a clause of the program $a \leftarrow c \Box B$ + a constraint c' such that $c' \rightarrow c$ and $c' \rightarrow B$ is expected but $c' \rightarrow a$ is unexpected.

The symptoms correspond to the kind of questions to the oracle, the error corresponds to the piece of code pointed as wrong by the declarative diagnoser.

With a positive symptom is associated a skeleton (of a proof tree) whose root label is the symptom and whose set of associated constraints is satisfiable (if the solver is incomplete and the constraints is not satisfiable in D then $c \rightarrow a$ holds because c is always false). The nodes of the skeletons (more precisely, from an abstract viewpoint, the subskeleton grafted on these nodes) are the computation steps we consider. With each node of the skeleton is associated a constrained atom (of the form $c \rightarrow a$).

The declarative diagnosis environment provides tools to explore such constrained skeleton and localizes some minimal error. Different notions of minimality will be considered. For this purpose ordering of nodes or subtrees may be considered.

At each diagnosis step, some query is presented to the oracle. It consists of displaying a constrained atom (the constrained atom associated with the current node of the skeleton). It must be noticed that different sets of constraints can be displayed: constraints associated to the whole skeleton or constraints associated to the subtree only, or some other subset. Furthermore displaying a huge set of constraints is not acceptable. Different options will be studied and some implemented. Note that $c \rightarrow a$ is equivalent to $c' \rightarrow a$ where c' is the existential closure of c except on the free variables of a. Thus only the information on the possible variables of a needs to be displayed. Among the different options there is:

- To display an equivalent set of simplified constraints.
- To display some approximation or property.
- To display some graphical representation.

The choice may depend on the node (for example on the domain or on the type of the variables which occur in the atom).

4.3 Diagnosis of incompleteness (missing answer)

Type of symptom:

- negative symptom (missing answer): $c \wedge a \rightarrow c_1 \vee c_2 \vee \cdots \vee c_n$ (exists a solution of c which satisfies a and is solution of no c_i)

Type of error:

- negative incorrectness: a predicate definition (packet) $a \leftarrow \bigvee_i c_i \square B_i$ + a constraint c' such that $c' \rightarrow a$ is expected but $c' \rightarrow \bigvee_i c_i \square B_i$ is unexpected.

With a negative symptom is associated a finite LD-tree whose root is the symptom. The nodes of the LD-tree correspond to incomplete skeletons with a "satisfiable" associated set of constraints.

The declarative diagnosis environment provides tools to explore the LD-tree and localizes a minimal symptom. The exploration is "intelligent" again and guided by the well-founded order.

The notion of "minimality" is simple in the case of programs without coroutining. The tools will be designed for programs without coroutining. Extension to the case of coroutining will be further investigated.

At each step of diagnosis, some query is presented. With a node of the LD-tree, it corresponds a partial skeleton with associated set of constraints c', the query has the form: $c' \wedge a \rightarrow \bigvee_i c_i$ where the c_i's are the constraints associated to complete sub-skeletons derived from this node[3]. For query presentation, different options will be studied and included in the diagnoser as discussed for the positive case.

Different strategies may be implemented in order to localize the minimal symptom. Among these, obvious strategies like following the computation (trace), divide and query, heuristic driven, or on user's demand (if the oracle is the user, then she can choose the nodes w.r.t. her ability to answer the diagnoser questions).

5 Tools related to operational semantics and a single computation: search space and constraint propagation analysis (B2)

The tools presented in this section are intended to be used when a program, for a given test data, produces wrong results which cannot be detected or corrected by other tools, or shows unsatisfactory efficiency. The latter often means that no result is obtained in a reasonable amount of time.

[3]This holds without coroutining only

These tools refer to operational semantics and will present different aspects of the computation in order to help the user to understand the reasons for the undesired behavior.

A usual abstraction for a single computation of a (constraint) logic program, executed with the Prolog selection rule, is an LD-tree. Most of the tools considered in this section can be seen as means of obtaining information from an LD-tree and presenting this information to the user.

The need for novel debugging tools for constraint programming comes from the fact that the tools traditionally used in logic and imperative programming are not quite enough for constraint logic programming. In classical debuggers, the values of variables at a given execution state are visualized. In imperative languages destructive assignment is allowed and there is no extra information, besides the current value of variables and the execution program point. However, in CLP we deal with logical variables. A variable has a store containing extra information like pre/post properties, a domain (possible values) and suspended constraints. This information changes from one step in the computation to another. The user is able at any node in the LD-tree to give an interpretation to this dynamic information.

Classical debuggers somehow show the control part of the execution and the values of variables, but usually not in the most appropriate way: control information is usually too implicit, the number of variables whose value is shown is usually too large making it difficult to interpret, and they usually do not focus on errors or reasons for unexpected failures. Another aspect specific to constraint languages is the need for showing the set of active constraints in a given execution state, i.e., the store. An additional problem is that the number of active constraints is usually too large and some way of focusing in the relevant information is needed for the user to properly understand the effect of constraints. CLP systems have evolved to tackle large problems. For example in CHIP the introduction of global constraints like cumulative constraints [AB97, BC93] lacks equivalent debugging tools. An interpretation of an instance of the cumulative constraint is to show its evolving profile at runtime. There are a set of aspects to be visualized. Some of these aspects are user requirements for debugging, controlling, understanding and improving constraint programs developped for complex and combinatorial user applications [SC95a].

We now consider these three aspects separately.

5.1 Showing Control Features

Although one of the basic properties of constraint programs is that they can generally be understood declaratively, i.e., without looking at control aspects, a concrete control strategy which drives the execution flow (and, thus, the search) implicitly exists in the evaluation engine. Sometimes a lack of understanding of how this control operates makes a program difficult to debug from the performance point of view. This control has three important components: the search determined by the program, the search in labeling and the control of constraint propagation. Each of them can be influenced separately and can have a major impact on the performance of the program.

5.1.1 The Box Model

Classical Prolog debuggers are generally based on the well-known "box model" [Byr80]. This is an execution model where procedure invocation can be seen as a box, with input and output unidirectional gates named ports. These ports are associated with events happening to this call/box: entering into, exiting with success, reentering to try another choice, exiting with failure. Some other ports can be defined (for instance, a port indicating which rule number in the predicate is currently chosen, or the call arguments when the unification with the head of the tried rule succeeds, or other...).

Other ports especially devised for constraints will be studied. As an example, in the CHIC Project three ports were added to the classical box model related to the following events: the successful addition of a constraint to the store, the delay of a constraint, or the waking up of a constraint which was not instantiated enough to execute when first reached.

Following an execution with the box model can be done with several commands which show more or less details about boxes. Those allow, for example, skipping the details of a call, stepping into it, or executing until the next breakpoint. Further enhancements to a text-based debugger based on the Byrd model can include commands which display the current call and stacks. This would allow understanding in a simple way what the current state of the computation is.

Showing variables is also an interesting feature, which is present in the Byrd Box model for Prolog variables, but which should be augmented / accommodated to better suit the needs of the debugging of constraint programs.

Graphical Box-Model: A first idea would be to graphically display true boxes (which may contain other small boxes) with a zooming/unzooming mechanism when entering into/exiting from inner boxes. But the box-model metaphor, although very useful in a text environment, becomes hard to understand if more than a few levels of nested boxes are displayed. This graphical representation should thus be only used with pedagogical purposes. The next idea is to build the tree in which each node corresponds to a box, without trying to display child-boxes inside their parents. We get then a more informative and relatively condensed execution trace, as the and-or control tree described in Section 5.1.2.

5.1.2 Showing Control Graphically

It appears useful to develop one or more tools which show globally how the execution proceeded both from the point of view of the program search and of the labeling steps. It is interesting to show which variables are involved in a given node / predicate call; this particular aspect will be covered in other sections and by other tools, but it appears advantageous to interface those tools with the control view.

The third aspect related to control, i.e., computation of constraint propagation, will be dealt with in Section 5.3.3 below.

In several constraint logic programming systems (as CHIP or Prolog IV) the labeling / search strategy can be specified by passing a parameter to the appropriate predicate. At each node the user can control how good the behavior of the chosen strategy is. This, combined with visualization of the domain of variables provides the programmer with a good view of how the search space is being reduced.

Regarding the display of execution trees, we propose the following strategy:

- We will start by developing a generic search tree visualizer. The idea is to show graphically the nodes of both the programmed search and the labeling search trees.

  In the programmed search view, a node should be shown per predicate call. To display this search, information about predicate names, clause invocations, and goals called within a clause, should be kept (some suggestions on how this information is gathered, and where it is kept, can be found in Section 5.5). Several actions should be possible while displaying this tree, namely: stopping the execution, stepping forward and backward, and abstracting parts of the execution in order to avoid overwhelming the programmer with too many details (see Section 5.2.1).

  In the labeling search view, the nodes are those corresponding to selections performed within the labeling procedure. At each node the user can control how this strategy is good. This combined with variable domain visualization provide him a good view how the search space is reduced. Similar abstracting facilities apply.

- In a further elaboration, clicking on the nodes will give a (possibly graphical) view of the state of relevant variables at that point. This will be performed by calling the corresponding visualizers (see Sections 5.2 and 5.3) with the appropriate data.

It is also interesting to have the possibility of switching between an event-oriented view and a time-oriented view. In an event-oriented view, every execution event (basically, every node of the tree) takes the same space in the output. In a time-oriented view, the placement of the nodes is scaled in order to reflect the actual time that node has used (note that, for example, traversing a head may trigger complex constraint propagation and solving in the engine, which cannot be obviated). This dual view was also implemented in VisAndOr [CGH93], and shown to be useful in practice.

5.2 Showing Values of Variables

The values of variables are the drivers of the execution and its ultimate objective. Thus, it is of utmost importance knowing them in order to uncover any possible correctness or performance bug. In the case of traditional languages this is a comparatively easy task, but in the realm of constraint languages the situation is more involved: definite values are now transformed into constraints which relate the values of the variables and which restrict the values which a variable can take.

In principle all the variables could be visualized, regardless of whether they are reachable or not from the current clause invocation. Selecting precisely the variables which are reachable from a given program point can be done by hand (inspecting the program) or by using the tool(s) described in Section 5.1.2. Program inspection by hand can be done, for example, with the addition to the program of meta-predicates which specify which variables should be visualized. Such meta-predicates do not change the meaning of the program at all, and should thus be easily removable.

A tool to depict the values of variables (taking into account its constrained nature) is also perceived as very useful. The depiction for finite domain variables may have points in common with that proposed by Micha Meier [Mei96] for the Eclipse system [Eur93].

A first implementation of a prototype could follow the steps below:

1. Define a data format, closely related to the metalanguage to represent constraints and variable values.
2. Develop a tool which can accept data in this format and show (a subset of) the variables.

Using a separate data format to communicate debuggers and programming systems has already shown to be successful [Duc91, Duc92, CGH93]. As with the control-oriented tool, several actions should be possible within this tool: stopping, stepping forward and backwards, etc.

5.2.1 Abstracting

In executions of large problems, the debugging process has to cope with sizeable execution steps and with a large number of variables and possible values, which can overwhelm the programmer. Thus, it is highly desirable to have methods which help to deal with these cases. Abstraction methods, amenable to be applied to as many cases as possible, would give the programmer a more friendly interface.

In the case of control depiction (Section 5.1.2), a clear possibility is abstracting parts of the search tree, maybe by collapsing them [Sch97]. The user might select which parts of the tree can be collapsed (perhaps those which correspond to parts of the program known to be correct). However, methods should be devised in order not to lose important information when performing this abstraction. s

In the case of the representation of (finite) variable domains, feasible techniques have to be adopted for the cases in which there are either too many variables, or the domains of certain variables are too large.

Some of this techniques (e.g., projections, domain refinement) can be devised so as to be applied automatically by the visualization tool. Other, more advanced possibilities, can involve human interaction, where the user defines (possibly by using a language related to that of the constraint program) which values should be taken out of the variable representation.

5.3 Showing Constraints

Imagine a C(L)P program composed of one single rule, to which is associated a huge and/or complex set of constraints. When the resulting answer is "unexpected" (in any sense) it is obvious that no analysis of the program structure, execution, etc. will help at all.

We will call *constraint debugging* the process of debugging programs by examining the constraint store(s) as opposed to examining the rule-structure (and the execution) of the program.

It is generally admitted that, while programs and data are structured in may ways (procedures, predicates and rules, objects, tree structures, etc.), the store is a mere flat and huge collection of formulas with no structure whatsoever. Moreover, in modern constraint languages, cooperation of solvers generate even more complex, heterogeneous and intricate store structures (e.g., in Prolog IV).

A possible starting point for the design of a workable store structure should come out after the completion of tasks T.WP2.3 (abstract constraint solving) and T.WP3.4 (complex constraint abstraction) as well as their graphical debugging-related counterpart T.WP2.3 (domain independent visualization paradigm).

5.3.1 Showing Relationships

The tools described in Section 5.2 do not give any direct insight into the relationships / mutual influence among variables. As the values of variables are updated by adding constraints (which restrict a variable's domain, or relate different variables), exploring which constraints are active at a given point is also very interesting. In general, it is useful to show some / all the constraints in which a subset of variables are involved. Several tools which may help the user to view the active constraints can be devised. Obvious constraint representations include text-based displays of (projections of) constraints, but this may be not always appropriate if too many constraints / variables are involved.

It may be possible to develop graphical representations of the relationships among variables: for example, 2-D grids in which 2-D points which belong to the domains of two given variables are highlighted. This would show the combined effect of all the constraints on the two selected variables. The practicality and difficulty (both from the implementation and computational view) of such an approach is still unknown, and thus it will be addressed as an academic experiment.

An appealing possibility is allowing the user to interactively change values of variables, and see what are the effects on the other variables (this should be orthogonal to the way in which variable values are depicted). This would give an intuitive understanding of the way the variables relate to each other, and, thus, of the constraints placed on them. This is again a tentative proposal: there are computational complexity issues involved (the whole constraint system might need to be solved from scratch each time the user changes a variable value) and experiments are needed to determine whether such depictions give the desired feedback. Thus, it will be explored at research level.

5.3.2 Controlling size and complexity of the store (S-boxes)

As mentioned above, no reasoning can be done on a flat representation of the store. This can be addressed by allowing the programmer to *structure the store by modifying the constraint granularity.* This would be achievable by structuring the store as a hierarchy of sub-stores, organized themselves in sub-stores etc. and giving the programmer the necessary tools to:

- create and modify the hierarchy,
- concentrate on a local view of selected sub-parts,
- navigate in two directions (search tree and propagation process) in these sub-stores

Taking advantage of properties of the constraint narrowing operators and of the main algorithm computing the overall domain reductions, it is then possible to generate arbitrary levels of abstractions of the store by considering subsets of the store as global constraints (in order to simplify the representation).

Our preliminary proposition in this direction is to allow the user to add structure information to the source code by *marking selected goals in selected rule bodies.* The set of "interesting" constraints (e.g., finite domain and/or interval constraints) defined by the corresponding predicates would then form a sub-store (box). Hierarchical organization of the store is then associated to the program structure.

Contrarily, decomposition of (some) global constraints would help refining the visualization of crucial sub-parts of the store. In the case of pre-defined global constraints (especially in finite domain-based systems), domain modifications coming from global constraints may be incomprehensible to the programmer if he has no precise knowledge of the underlying algorithms. Actual decomposition might not be very useful in this case.

We present below a graphical tool which implements a "divide and conquer" method reducing the store to a manageable size.

The store-as-a-box view The key idea of our method is to consider a store as a closed box (referred as *S-box* in the following) with no connection with the outside. Moreover, the store may gather only a subset of all constraints already processed. By default, there is only one box which contains the whole store at a node of the search tree.

One way to create S-boxes is to highlight some parts of the CLP program. Action depends of what is highlighted:

- some constraints: they are all linked in one S-box;
- a goal: all constraints created when resolving the goal are included in one S-box;
- head of a clause: all constraints created while in the clause are included in one S-box.

Note that the three cases may be mixed. For example, highlighting two constraints (c_1, c_2) and one goal g_1 creates one box gathering c_1, c_2 and all the constraints created while resolving the goal. S-boxes are thus organized in a hierarchical way.

From the propagation view-point, a S-box acts as a big constraint which is the conjunction of all constraints it embeds (see section 5.3.2). Note that it implies a modification of the propagation process (all constraints pertaining to the same S-box must be awaken in such a way that the whole propagation appears as atomic from the outside).

The set of constraints of the active S-box is displayed in a graph form: nodes are constraints[4] and edges link constraints sharing variables. Moreover, S-boxes at a deeper level than the active S-box will be materialized in a way to be defined (by means of color or figures embedding constraints, ...). A S-box can be promoted to the active one by clicking on it. Its contents is then "zoomed" and it becomes the active store , that is all links with S-boxes at the same or higher level disappear. All is done as if constraints outside the active S-box did not exist. This allows the user to focus on a sub-part of its problem. The user has the ability to create sub-boxes in the active S-box by clicking on the constraints he wants to gather.

The hierarchy of existing S-boxes will appear in a tree form allowing the user to step backward and forward in the "zooming process".

As it can easily be seen, our approach is similar to extracting constraints from the main CLP program, then gathering them in clauses to form a new structured Prolog program. An intended benefit of this would be to allow declarative debugging of CLP programs with no structure (such as the one goal program) using methods such as those described in [FT97].

5.3.3 Constraint propagation

Tracing of the propagation process is as important a task in the CLP scheme as the tracing of instructions in imperative languages.

This is very important for correctness but can also be of crucial interest to detect slow convergence or inefficient constraint stacking. It is not conceivable to trace step by step the whole fixed point computation or even worse a whole sequence of fixed point computations. A possible solution is to give the programmer the possibility to set breakpoints based on certain properties of the execution state. For example a breakpoint can be set when:

1. inconsistency is detected,
2. for a given variable there is a domain modification, an absolute reduction (a given size/number of values is obtained, particularly fixed variables and variables with canonical domains) or a relative reduction (a certain ratio of its previous/initial set of values is computed),
3. a given constraint, or a constraint belonging to a given set is stacked, actually used or actually used to reduce the domain of a particular variable

It will then be possible to let the propagation run until a breakpoint or a fix-point is reached or alternatively to trace step-by-step each propagation. When in step-by-step mode, the user will have the ability to record every modification such that it will be possible to step backward or delete a constraint.

From a mid/long term point of view it would be very desirable to be able to enhance the debugger with facilities to propose courses of action, to provide selected information, possibly to ask questions and more generally to try to compute short cuts, thus avoiding the programmer lengthy investigations and possible irrelevant search. This could be done by extracting possible relevant information from the store during the execution process, depending on store/domain modifications, previous actions or answers given by the programmer. On the basis that our hierarchical representation of the store could be represented as a CLP program, ongoing work on declarative debugging in the project could provide ideas and starting points in this prospective direction.

[4]Note that the graph contains only user constraints, that is decomposed constraints are reconstructed in order to be displayed. This will induce modification in the core of host systems.

5.3.4 Symbolic processing of the store

If debugging facilities are extended to general, domain and solver-independent C(L)P languages, the domain-oriented approach presented below is not sufficient. Many solvers dynamically modify the constraint store by rewriting formulas. Except for specialists, that is constraint solver designers, tracking resolution steps for these representations of constraints (a situation that is very similar to global constraint debugging) is of little interest (choice of the pivoting variable, etc). Some facilities can be nevertheless shared with domain-based solvers like dynamic modifications of the store.

Specific functions can also be envisaged. For example, one natural function the programmer would like to be granted in this case is the capability to simplify the store (redundancies elimination, computation of compact constraints) and to project the store over a subset of the variables. Unfortunately, as it was shown in the Boolean [Ben93] and in the linear case [LMH93], these techniques are very difficult to design and implement (and also very time-consuming).

Symbolic processing can also be considered as an important feature of domain-based constraint solving, at least for one important reason, linked to performance debugging. As it is well known from constraint-based application developers, substantial gains in efficiency can be achieved by *adding* redundant constraints.

5.4 Relating Debugging to the Source Program

A desirable feature of most debugging tools is the ability of relating the execution state to some part of the source program. This is because the programmer is usually familiar with the program and this can greatly help the user to understand the program execution.

Concerning the (apparent) independence between the constraints appearing in the store and the source code, the problem is itself quite hard to tackle. Possible investigations include:

- intuitive renaming of fresh variables,
- careful track keeping of relations between variables and their “home-rules”,
- design of constraint structures with no primitive constraint decomposition, that is globalization of the representation as opposed to the “resolution”
- global processing of constraints *a la* Newton [BMV94, VMB97]

This problem is, in our opinion, crucial. No serious debugging can be done in presence of (numerous) variables apparently unrelated to the program source. Note that the envisaged solutions to this problem involve major modifications of the solving algorithms themselves.

5.5 Gathering Data about the Execution

Data about the execution need to be communicated to the tools. This data can be received by the tools as execution proceeds (on-line tools), or after the execution has finished (off-line tools). In principle, this distinction should not affect the format of the data used to interface the tools with the CLP system. However, if the tool has to react with the CLP system (e.g., to give hints about labeling heuristics), then it should be on-line.

The exact format of these trace files is still to be clearly determined, and will depend on further developments in the project. From an implementation point of view, the generation of this file can, in principle, be performed in several ways:

- Using a meta-interpreter, which detects branching / constraint addition / propagation steps / etc., and dumps the corresponding info to the trace file.
- Using a variant of the existent engine, which performs basically the same operations as above, but at a lower level.
- Using an annotated program, possibly accessing ad-hoc primitives available from the engine. Insertion of these annotations can be done by the user or, alternatively, automatically, as a rewriting of the original program.

All three options appear interesting. The last one seems to be particularly attractive given that it offers the advantages of not having to change the engine or constraint solver implementation too much, and not imposing the high execution overheads of a meta-interpreter.

Profiling A simpler approach is profiling. The execution engine is augmented in order to store pieces of information regarding program execution. These data may be used afterwards as hints of the program behavior, especially for performance debugging. Examples of data stored are:

- number of fixed point steps,
- number of reduction steps,
- unused constraints
- constraints which were (often) stacked with no effect on domain reductions,
- constraints which were (often) stacked with little effect on domain reductions (slow convergence),
- variables whose domain was not reduced,

5.6 Run-Time Checking (Dynamic Diagnosis)

We expect that it will be quite often the case that provers (tools A2) are neither able to prove nor to disprove that a certain program property holds. This is because proving algorithms are usually incomplete. In such a case it will be useful to provide a tool for dynamic checking, i.e., at run-time. Such a tool would test whether a given property is satisfied in particular test runs of the program. It will be implemented by:

- Defining run-time checks in the source language suitable for several forms of assertions.
- Enhancing the static checker to annotate the program where assertions have not been proven and to discard proven assertions.
- Incorporating to the static checker a back-end which translates previous annotations into the suitable run-time checks.
- Providing tools to localise erroneous clauses, starting from run-time symptoms.

6 Actual debugging tools to be developed

This section summarizes the intermediate basic or advanced tools and the contents of the platforms which will include different tools.

6.1 Intermediate modules

Intermediate modules correspond to interfaces or algorithms developed by one partner which are of interest for other partners. They should be usable with the languages defined in Task T.WP1.2.

6.1.1 Static analysis tools

A generic tool for assertion generation based on abstract interpretation (AI) techniques. This tool takes as input a program and generates a set of assertions derived from the program. It will be implemented by:

- Designing the overall language of assertions (using ideas from [BCHP96] and [BDM97]).
- Designing a sub-language of assertions which includes special properties suitable for static inference through abstract interpretation.
- Modifying the CIAO analyser to produce output in this form.
- Integrating the assertion generation module into the public-domain CIAO system.

6.1.2 Declarative diagnosis algorithms and dialog modules

The declarative diagnoser will be produced in such a modular form which will permit to export main functionalities to other constraint systems.

The “exportable” tools to be designed will include:

- A navigation module to explore constrained skeleton for localization of minimal incorrectness error.
- A constrained atom presentation module for different kinds of queries (wrong and missing answers) and representations.
- A navigation module to explore finite LD-tree for localization of minimal incompleteness error.
- A dialog module to organize DD session: selection of symptoms, navigation, query presentation, answers and possible storage.

We intend to consider combination of DD sessions with assertions provided by the user and/or generated automatically.

The modules will be parameterized by the constraints domains. The three domains which will be considered in the tools are term domains, finite domains and intervals. Combination with other constraints domains will be further investigated.

6.1.3 Visualization functions library

The visualization library is designed to consist of several independent tools, which can be used together thanks to the design of the appropriate interfaces. These interfaces will serve to allow communication of the tools among them and with the program being debugged. The tools to be implemented include:

Graphical interface for classical debuggers: A GUI for the box model, augmented with capabilities to represent control flow of CLP(Intervals) programs.

Depiction of the search tree: The box model only shows a node of the computation at a time (but with great detail). Sometimes it is advisable to have a global look at the whole computation, showing all the nodes in the tree. Optionally, a more detailed view of every node can be selected.

For the programmed search part we propose to start this work by extending preliminary existing prototypes [Lue97], based partially on the Transparent Prolog Machine [EB88]. For the labeling search we propose to use an or-tree depiction, combined with historical views on variable evolution (see Section 5.2).

Visualization in debugging: A typical finite domain program consists of three, well separated parts: variable definition, constraint statement and search procedure. Our intention is to provide a tool which help the user to understand all three parts with special emphasis on the visualization of the search procedure.

In the search procedure the constraint program is run to generate information about the search tree which is then displayed in a graphical form. The user can interact with this representation of the search tree and re-instantiate the state of the search process at a given point in the search tree. The visualization tool provides a number of views into the search procedure:

1. **Variable view:** This view focuses on showing the values of variables. Several different views are possible:
   (a) **Update view:** This shows the change of the variables on the different steps on the current path from root to the selected node.
   (b) **State view:** This view shows the domains of all variables at a given state.
   (c) **Spy view:** This view allows to follow the change of a single domain variable through the path from root to current selection.
2. **Constraint view:** The constraint views show evolution of constraints within the search process. Specific views will be developed to take into account global constraints in CHIP.
3. **Propagation view:** This view should display active constraints at a given point.
4. **Assertion display:** The use of assertions forms an important aspect of declarative debugging. In the visualization tool, assertions can be used to rapidly find particular nodes in the search tree. As defined in Section 3.2.1, assertions are properties expressed in a language. A language of assertions will be added to the kernel CLP language.

Showing relationships among variables: Depicting the relationships among variables gives an intuition of the constraints which hold in the program. However, it seems that performing this kind of representation is very difficult, and the research performed so far in this field is very little. Therefore, the tools proposed here will be mainly of academic nature, and subject to possible further changes as the project develops.

Abstracting views: Sometimes the size or nature of the program being debugged makes detailed views to be useless. In this case, a means of performing an abstraction is needed. This abstraction should be general in that it can be applied to different constraint systems (e.g., projection on constraints, on variables, collapsing parts of a tree... even user-defined filters).

A relevant practical issue is which low-level graphical library will be used in the development of the tools. The three most interesting possibilities which have emerged from previous discussions are Tcl/Tk, Java and XGIP. Because all these libraries are interesting and because tools written with different libraries can coexist (since they would communicate at the Prolog level, not at the graphical interface level) we do not make any project-level commitments regarding this issue.

6.2 CLP platforms with debugging tools

6.2.1 CLP(FD), CLP(FD+Interval)

INRIA will develop a complete debugging environment for CLP(FD) and CLP(FD+Interval). This environment will include:

- Type checking and verification with assertions using some abstract domains.
- Declarative diagnoser with assertions and visualization.
- A classical debugger connected with visualization.
- Search tree visualization.
- Constraint debugging.

INRIA plans also to develop a constraint debugger as described in Section 5.3 over clp(FD+Intervals) using Tcl/Tk. It will be designed in order to let the user debug a compiled program as well as an interpreted one (for this, we will create two compilers, one adding debugging informations to the compiled clp program. This will be transparent from the user point of view by means of a common front-end).

In order to be portable on every constraint solving system, the design of this debugger will rely on the definition of an interfacing language/protocol as a corner-stone. This protocol will define a minimal set of functionalities to be provided by each host attempting to use the debugger.

6.2.2 The CIAO System

UPM will devise several toolsets which will be added to the CIAO system. The result will be a relatively comprehensive, public domain correctness and performance validation and diagnosis environment:

- An integrated correctness validation and debugging toolset. Central to the system will be the development of an assertion language which will allow expressing both intended and actual properties of programs. Properties supported will be quite general (essentially, those that can be expressed in the source language). Some (abstract) properties will be predefined. The environment will support:
  - Dynamic (i.e., run-time) checking of assertions in debug mode.
  - Static checking (verification) of programs annotated with assertions (using predefined/abstract properties) which express user requirements.
  - Static generation of assertions (i.e., actual properties of the program at hand) for unannotated or partially annotated programs.
- An integrated performance validation and debugging toolset. This will include:
  - A generic search tree viewer, which will be an extension of the traditional box model and capable of depicting the search tree and the node contents in several ways.
  - A set of experimental constraint visualization modules, which will be linked to the search tree viewer and, possibly, to program assertions.

  Gradual improvements will be made to the generic search tree and constraint viewers, with abstraction capabilities, so that selected portions of the visualized objects can be abstracted away.

There will be strong collaboration between Linköping, INRIA, and UPM on the development of the basic technologies required for the tools mentioned above. Linköping, in collaboration with UPM, will extend the existing global analysis module in CIAO and perform the necessary improvements to realize type inference and checking of CHIP programs.

6.2.3 Prolog IV Beta version

A first application of the results of this work will be integrated into the Prolog IV compiler.

- Assertions:
  - Ability to use interval predicates in assertions
  - Static checking at the clause level
  - Dynamic checking in debug mode
- Visual Debugging Tools:
  - Enhancements in the Prolog IV Source Level Debugger and Box Model Debugger, mostly by adding support for calls/constraints (locally and globally) at the debugger "prompt" level.
  - The Control Tree Search Debugger, both the PrologIA's prototype and the UPM's coming tool, (probably with a final merge of both will be in Prolog IV). Integrating a "full tool" will be tried if its "graphical" implantation language is TCL/TK.

6.2.4 CHIP V5 Beta version

COSYTEC will develop the following modules which will be integrated in its existing constraint programming environment CHIP V5:

- connection of the its debugger to the graphical visualizer,
- search tree visualizer,
- graphical visualizer of some variants of global constraints,
- assertions visualizer.

References

[AB97] A. Aggoun and N. Beldiceanu. Extending chip in order to solve complex scheduling techniques. *Journal of Mathematical and Computer Modelling*, 17(7):57–73, 1997.

[BC93] N. Beldiceanu and E. Contejean. Introducing global constraints in chip. *Journal of Mathematical and Computer Modelling*, 20(12):97–123, 1993.

[BCHP96] 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.

[BDD+97] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linkoping, Sweden, May 1997. U. of Linkoping Press.

[BDM97] J. Boye, W. Drabent, and J. Małuszyński. Declarative diagnosis of constraint programs: an assertion-based approach, May 1997.

[Ben93] Frédéric Benhamou. Boolean Algorithms in Prolog III. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 307–325. MIT Press, 1993.

[Ben96] F. Benhamou. Heterogeneous Constraint Solving. In *Proceedings of the fifth International Conference on Algebraic and Logic Programming (ALP'96)*, volume 1139 of *LNCS*, pages 62–76, Aachen, Germany, 1996. Springer-Verlag.

[BFL+95] Michel Bergère, Gérard Ferrand, François Le Berre, Bernard Malfon, and Alexandre Tessier. La Programmation Logique avec Contraintes revisitée en termes d'arbres de preuve et de squelettes. Technical Report 95/06, LIFO, University of Orléans, 1995.

[BM97] J. Boye and J. Małuszyński. Directional types and the annotation method. *Journal of Logic Programming*, 1997. (in print).

[BMV94] Frédéric Benhamou, David McAllester, and Pascal Van Hentenryck. CLP(Intervals) Revisited. In *Proceedings of ILPS'94*, pages 1–21, Ithaca, NY, USA, 1994. MIT Press.

[BO97] Frédéric Benhamou and William J. Older. Applying Interval Arithmetic to Real, Integer and Boolean Constraints. *Journal of Logic Programming*, 1997. (Forthcoming).

[BT95] Frédéric Benhamou and Touraïvane. Prolog IV: langage et algorithmes. In *JFPL'95: IVèmes Journées Francophones de Programmation en Logique*, pages 51–65, Dijon, France, 1995. Teknea.

[Byr80] L. Byrd. Understanding the control flow of prolog programs. In S.-A. Tärnlund, editor, *Proc. of the Workshop on Logic Programming*, Debrecen, 1980.

[CC77] 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.

[CGH93] M. Carro, L. Gómez, and M. Hermenegildo. Some Paradigms for Visualizing Parallel Execution of Logic Programs. In *1993 International Conference on Logic Programming*, pages 184–201. MIT Press, June 1993.

[CLMV96] 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.

[Col96] Alain Colmerauer. Specifications of Prolog IV. Draft, 1996.

[dlBH93] M. García de la Banda and M. Hermenegildo. A Practical Approach to the Global Analysis of Constraint Logic Programs. In *1993 International Logic Programming Symposium*, pages 437–455. MIT Press, Cambridge, MA, October 1993.

[dlBHB+96] 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.

[DM93] P. Deransart and J. Małuszyński. *A Grammatical View of Logic Programming*. The MIT Press, 1993.

[Duc91] M. Ducassé. Abstract views of prolog executions in opium. In *Proc. International Logic Programming Symposium*, San Diego, 1991. MIT Press.

[Duc92] M. Ducassé. A general trace query mechanism based on prolog. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 400–414. Springer LNCS 631, 1992.

[EB88] M. Eisenstadt and M. Brayshaw. The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming. *Journal of Logic Programming*, 5(4), 1988.

[Eur93] European Computer Research Center. *Eclipse User's Guide*, 1993.

[Fag96] François Fages. *Programmation Logique par Contraintes*. Ellipses, X-Ecole Polytechnique, France, 1996.

[FT97] Gérard Ferrand and Alexandre Tessier. Positive and Negative Diagnosis for Constraint Logic Programs in terms of proof skeletons. In *International Workshop on Automated Debugging*, 1997. (To appear).

[Gou95] F. Goualard. Icons - compilation d'un langage pour la résolution de contraintes par propagation d'intervalles. Master's thesis, University of Orléans, 1995. (Rapport de DEA - in French).

[HL94] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[HPMS95] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Logic Programs. In *International Conference on Logic Programming*, pages 797–811. MIT Press, June 1995.

[JB92] G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):205–258, July 1992.

[JM94] J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 13/20:503–581, 1994.

[LMH93] J.L. Lassez, K. McAloon, and T. Huhnh. Simplification and Elimination of Redundant Linear Arithmetic Constraints. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*. MIT Press, 1993.

[Lue97] Angel López Luengo. APT: implementación de un visualizador gráfico de la ejecución de prolog. Master's thesis, T.U. of Madrid (UPM), Facultad de Informática, Madrid, 28660, 1997. In Preparation.

[Mei96] M. Meier. Grace User Manual, 1996. Available at `http://www.ecrc.de/eclipse/html/grace/grace.html` http://www.ecrc.de/eclipse/html/grace/grace.html.

[OB93] William Older and Frédéric Benhamou. Programming in CLP(BNR). In Paris Kanellakis, Jean-Louis Lassez, and Vijay Saraswat, editors, *PPCP'93: First Workshop on Principles and Practice of Constraint Programming*, Providence RI, 1993.

[OV93] William Older and André Vellino. Constraint Arithmetic on Real Intervals. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*. MIT Press, 1993.

[SC95a] H. Simonis and T. Cornelissens. Modelling producer/consumen constraints. In *PPCP'95: Second Workshop on Principles and Practice of Constraint Programming*, volume LNCS 976, pages 449–463, Cassis, France, 9 1995. Springer Verlag.

[SC95b] H. Simonis and T. Cornelissens. Modelling Producer/Consumer Constraints. In *Principles and Practice of Constraint Programming, CP'95*, volume 976 of *Lecture Notes in Computer Science*, pages 449–463. Springer Verlag, September 1995.

[Sch97] Christian Schulte. Oz Explorer: A Visual Constraint Programming Tool. In Lee naish, editor, *ICLP'97*. MIT Press, July 1997.

[SG95] H. Saglam and J. Gallagher. Approximating Constraint Logic Programs Using Polymorphic Types and Regular Descriptions. Technical Report CSTR-95-017, Department of Computer Science, University of Bristol, 1995.

[Vet94] E. Vetillard. *Utilisation de Declarations en Programmation Logique avec Contraintes*. PhD thesis, U. of Aix-Marseilles II, 1994.

[VMB97] P. Van Hentenryck, L. Michel, and F. Benhamou. `Newton`: Constraint Programming over Non-linear Constraints. *Scientific Programming*, 1997. (Forthcoming).