CLP performance analysis: Search-tree Visualisation and Abstraction

C. Aillaud
Christophe.Aillaud@inria.fr
P. Deransart
Pierre.Deransart@inria.fr

INRIA-Rocquencourt, Projet LOCO, BP 105, F-78153 Le Chesnay Cedex, France.

Abstract

This paper presents the principles of a Constraint Logic Program debugging module, currently implemented in the DiSCiPl project [3], which concerns performance debuggging and helps to understand program behaviour in particular in complex combinatorial problems. It is based on search-tree abstractions. It defines the notion of abstract search-tree and a method to specify it. The originality of the approach lies in its genericity (different kinds of abstractions may be defined to analyse the computation space) and the way to specify the abstractions, in particular, using program predicate properties.

1 Introduction

Constraint logic programs are very hard to debug. One of the most important and difficult aspect of CLP debugging is "performance analysis", i.e understanding properties of the search space. The combinatorial complexity of the problems solved by CLP makes classical methods like trace analysis hopeless. Different tools have been proposed for performance analysis which focus on dynamic constraints display (like GANT Charts) [10] or interactive restricted [1, 16] or unrestricted search space display [13]. All existing systems have predefined views of the search space with limited "collapsing" functionalities.

This paper presents a generalisation of collapsing where synthetic views of the computation space can be defined by program properties. More synthetic views of the search space help to analyse the complexity of the computations and the used strategies, and therefore to improve performances.

Our approach to performance analysis is based on search-tree abstraction. The search-tree is a way to display the operational behaviour of CLP, i.e. a way to represent the search space. Search-trees are usually much too large to be displayed. We therefore present a method to display simplified versions of the search tree, called "abstract search-tree". We explain what is the abstract search tree and a method to specify it. Simple examples of abstract search-trees are sub search-trees or labellings. The originality of the approach lies in its genericity (different kinds of abstractions may be defined) and the way to specify the abstractions, in particular, using program predicate properties.

The paper has four main parts: in Section 2 we give the basic elements of the operational semantics of CLP when implemented with standard Prolog computation rule and its representation by CLP search-tree. It also describes assertional approach used for specification of abstract search-trees. Section 3 defines reduction of ordered tree in a manner which is independent from their meaning. It gives a declarative definition of the reduced tree corresponding to a subset of selected nodes and arcs, and describes how the reduced tree can be computed. In Sections 4 and 5 reduction is applied to CLP search-tree to obtain search-tree abstractions. A language for node and arc selection is introduced. Finally Sections 6 and 7 deal with implementation and examples.

2 CLP: Syntax and Semantics

In this section we recall the syntax and the semantics of CLP programs. We focus on Operational semantics.

2.1 Syntax

A CLP program is built over finite sets of function symbols, predicate symbols and denumerable set of variables. Among the predicates we distinguish a special class of constraint predicates[1], including the standard equality. The function symbols are divided into constructors and function names.

The atomic formulae constructed with the constraint predicates will be called basic constraints. The other atomic formulae will be called atoms. An atom is said to be normalised if it is of the form $p(x_1, .., x_n)$ where x_i's are distinct variables.

A program is a set of clauses of the form $h\ :-\ \{c|\ b\}^*$. ,where c is a basic constraint and h and b are normalised atoms. System predicates of the existing CLP systems will be abstracted as predicates.

A *unit clause* is a clause in which the body has constraints only.

A goal is a clause with no head.

Empty body is denoted *true* and may be omitted. A clause with empty body can be just a head.

The clauses in the examples may include unnormalized atoms as a notational shorthand for normalized ones.

2.2 Constraint domains

The constraints of a CLP program are interpreted over some structure D. The language of constraints together with the interpretation form a constraint domain. In practice a single CLP language may include several domains and some interaction between them may be possible. Typically the CLP languages make it possible to combine the use of interpreted function symbols belonging to some specific constraint domain and term constructors from the Herbrand constraint domain. Domains of variables can be inferred from the constraints in which they occur.

[1] Constraints are a subset of the built-in predicates in a CLP system.

Example 2.1 Here is a CLP(FD) program in Calypso[2]

```
Pr1    sp1(N,[P|T],[Q|R]) :- N #= P*Q+Np, sp1(Np,T,R).
       sp1(0,[],[]).

goal(R) :- R = [_,_], fd_domain(R,0,10), sp1(10, [5, 2], R), fd_labeling(R).
```

Variables `N`, `P`, `Q`, `Np` denote finite domain variables (positive integers), variables `T`, `R`, lists of integers, `+,*` integer operations with finite domain variables. □

The intended meaning of the program is that `sp1` describes the scalar product relation where the first argument is an integer representing the scalar product of two vectors represented as lists (of integers).

2.3 Constrained atom and D-atom

A *constrained atom* is a pair (σ, a), where a is an atom and σ is a finite set of constraints. For example

`({X#=1,Y#=X+T, fd_domain(T,1,2), fd_domain(Z,2,3)}, sorted([X,Y,Z]))`

is a constrained atom in Calypso.

For a given constraint domain D a D-atom is an object of the form $p(d_1, ..., d_n)$ where p is a predicate and $d_1, ..., d_n$ are elements of the underlying structure.

Given a set of constraint σ we denote by $Sat(\sigma)$ the set of all valuations satisfying the conjunction of the constraints in σ in the interpretation of the constraint domain.

A constrained atom $a = (\sigma,\ p(t_1, ..., t_n))$, which is a syntactic object, represents the following set a_D of D-atoms:

$$a_D = \{p(\mu(t_1), .., \mu(t_n)) | \mu \in\ Sat(\sigma)\}$$

If D is the finite domain then the example constrained atom above corresponds to the following set of the D-atoms:

`{sorted([1,2,2]), sorted([1,2,3]), sorted([1,3,2]), sorted([1,3,3])}`

A constrained atom a is said to be *ground* iff a_D is a singleton.

2.4 Operational semantics

A CLP program is used to answer queries. We assume without loss of generality that the query has the form of a constrained atom (σ, a). The computed answer may also be seen as a constrained atom (σ', a), where σ' is called *the computed constraint.*

Example 2.2 For the scalar product program `Pr1` a possible query is e.g.:

```
:- X = [_,_,_], fd_domain(X,0,20), sp1(20, [4, 3, 10], X), fd_labeling(X).
```

[2]Calypso is the successor of `CLP(FD)` developed at INRIA-Rocquencourt and Paris I (ftp://ftp.inria.fr/INRIA/Projects/loco/calypso/) [2].

in which case the constrained atom obtained before labelling is:

```
({20 #= 4*X1+3*X2+10*X3, X = [X1, X2, X3],
fd_domain(X1,0,5), fd_domain(X2,0,6), fd_domain(X3,0,2)³},
sp1(20, [4, 3, 10], X)).
```

□

A state of computation can be characterized by a *resolvent* defined as a pair (σ, G) where G is a sequence of atoms and basic constraints, and σ is a set of constraints, called the *constraint store*.

The initial state is determined by the query (σ_0, g_0). The operational semantics is defined by a transition function on the states.

At each step a satisfiability test of the possibly new constraint store is performed. The satisfiability check is implementation dependent. It is usually based on a sufficient condition, so that some unsatisfiable constraint stores may pass the test. A store is said *acceptable* if it passes the satisfiability test.

The transition relation on states determines for a state (σ, G) the new state (σ', G') defined as follows:

- If G is empty the computation terminates without changing the state. The computed answer constraint is then obtained as a simplification of the formula $\exists \overline{x} \sigma$ where $\overline{x}$ denotes the free variables of σ that do not appear in the initial query. The simplification is an implementation dependent transformation preserving the logical equivalence.

- For a non-empty G with the first element a and tail T ($G = [a|T]$) three cases are possible:

  - if a is a constraint and $\sigma \cup \{a\}$ is an acceptable store then define $\sigma' = \sigma \cup \{a\}$, and $G' = T$,
  - if a is an atom $p(\overline{t})$ then a renamed variant of a clause of the form $p(\overline{t'})$:-B is created (if any), $\sigma' = \sigma \cup \{\overline{t} = \overline{t'}\}$ and G' is obtained from G by replacement of a by B,
  - if a is a constraint such that $\sigma \cup \{a\}$ is unsatisfiable or a is an atom such that there is no clause defining it then the computation halts with failure.

  (σ, a) is called the *selected (constrained) atom.*

The scalar product example shows the simplified answer constraints as constructed by Calypso.

Notice that, in the case of the Herbrand constraint domain with most general unifier used both as satisfiability check and constraint simplifier, the above described operational semantics can be seen as LD-resolution, or SLD resolution with (ISO)-Prolog computation rule [5].

[3] The `fd_domain/3` constraints do not occur in the program, but are in the store as a result of values propagation.

2.5 The CLP search-tree

The different computations defined by the operational semantic transition function may be represented by a tree called the *CLP search-tree* (in short search-tree) that we define as follows.

- Each node is labelled by a resolvent (σ, G), such that, if G is not empty, the first element of G is an atom a formed with a user defined predicate.
- There are two kinds of leaf-nodes:
  - Nodes whose goal label G is empty (or `true`), called *success nodes.*
  - Nodes with a nonempty goal label, but which cannot be expanded with any acceptable store, called *failure nodes.*
- A non-leaf node labelled (σ, G) has as many children labelled (σ', G') as there are clauses l leading to acceptable constraint store as follows.

  If $G = [a|T]$ and l is a rule, i.e. $l:\ h$:- $c_0{}^4, a_1, B.$, then $\sigma' = \sigma \cup c_0$, and $G' = [a_1, B|T]$ flattened.

  If $G = [a, c_1, a_1|T]$ and l is a fact, i.e. $l:\ h$:- c^4 , then $\sigma' = \sigma \cup c \cup c_1$, and $G' = [a_1|T]$ flattened.

  The children are in the same order as the clauses in the database. A *left-to-right order* of the children will be assumed in a two dimensional representation of the search-tree.

Notice that one arc in the search tree may correspond to several computation steps. In order to limit the number of nodes in the search tree, successive computation steps with store accumulation only are compacted into one single step.

There are three kinds of branches: *success*, *failed*, *infinite.*

- A *success branch* corresponds to a success node.
- A *failed branch* corresponds to a failure node.
- An *infinite branch* corresponds to a nonterminating computation following the Prolog computation rule.

If there is no infinite branch in a search-tree, it is a *finite search-tree.*

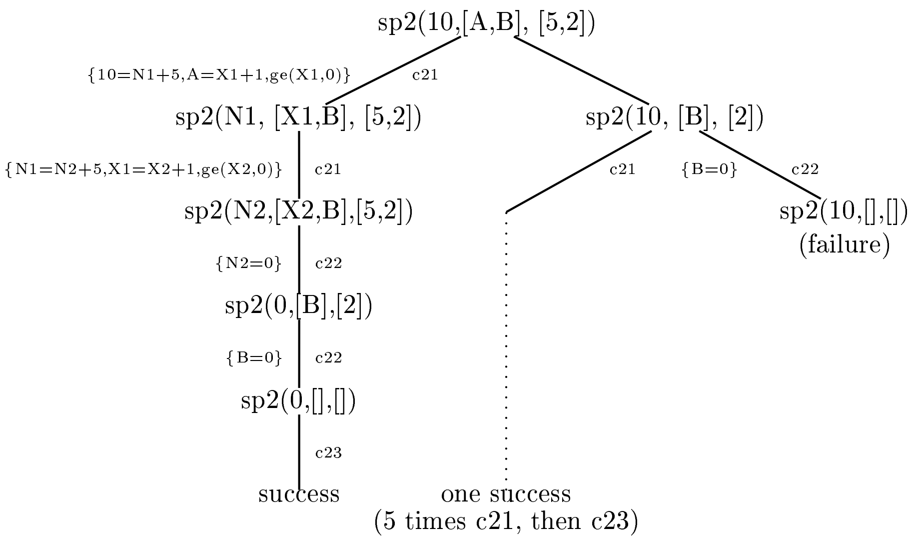
At every node the label store corresponds to the current store. To every success branch there corresponds an *answer constraint* which is the current store labelling the (success) leaf.

Notice that the notion of search-tree depends only on the predication-choice (which is imposed here by the "Prolog computation rule"). The clause-choice (not specified yet) specifies the way it is visited. Given a search-tree, the execution of a goal in the context of a database, with the Prolog computation rule, may be represented by a depth-first left-to-right visit of the search-tree. This

[4] c_0, c_1 and c below denote a conjunction of basic constraints
4

visit defines the visit order of the nodes of the search-tree, hence the order of execution of goals and subgoals. It is a total order (see below). If the search-tree has infinite branches, there is no way to visit beyond the first one, which will be explored indefinitely. This explains why the execution does not terminate when the traversal visits an infinite branch. It also explains why not all solutions may be computed if there is an infinite branch with some success branches afterwards.

An example of search-tree is depicted bellow for program `Pr2` and the goal `({N=10, P=[A,B], Q=[5,2]}, sp2(N, P, Q))`



In the scheme above, some informations only are displayed: each non leaf node is labelled with the first atom in the resolvent and each arc is labelled with the clause used to produce it and the set of constraints added to the store in this step. These informations are the most useful local ones which may be displayed on user's demand.

Arcs in a search-tree correspond to a computation step formalised by the use of a clause instance where the head is the chosen atom of the parent node and the first atom in the body (if any) is the chosen atom in the child node. The constraints added to the store in this step are all the constraints before the first atom in the body. In a step where some fact is selected (its body is a conjunction of constraints), then the constraints added to the store in this step are all the constraints of the fact with the constraints in the parent resolvent after this fact and before the following atom in the resolvent (if there is none, the parent resolvent of the parent is considered... etc).

It is important to observe that the tree traversal induced by the standard Prolog computation rule defines a total order on the nodes of the search-tree. If the nodes are coded using a Dewey-like notation (root is **1**, successive children are **11**, **12**,... and so on). The total order corresponds to lexicographic ordering of the nodes. This order will be denoted *family ordering*. It consists also of two

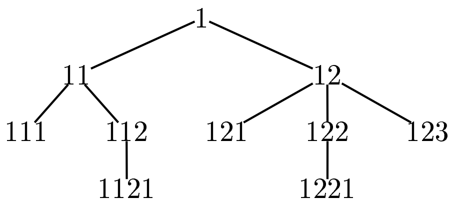


Figure 1: Example CST with nodes total order (family order)

partial orders (refered also as "family orders": the "child" relation[5] (a parent, or ancestor node is the prefix of a child or a descendent) and the "brother" relation (ordering the nodes with a common parent). If two nodes are not "descendent" neither "ancestors", we will say they are "gran cousin". The brother relation follows the order in which the clauses have been chosen to build the brother nodes. The family orders extend trivially to arcs. Figure 1 illustrates the relations: node **112** is descendent of node **11**, but node **122** (**12**) is gran cousin of node **11** (**112**).

2.6 Labelling

CLP systems over finite domains use "labelling" procedures to enumerate solutions when the current store is not sufficient to fix all finite domain variables. These procedures, with different strategies are part of CLP processors and the user may call them with a "labelling" system predicate. The user can also define its own labellings. A labelling may be visualised also as search-tree. In this case all the successful branches have the same size, corresponding to the number of finite domain variables to be enumerated.

Labelling search-trees are the most important parts of computations to be visualised: it is purely combinatorial and is the corner-stone of possible poor performances. Performance analysis needs therefore to focus on this part. It is thus particularly useful to be able to extract from the search tree the labelling part. This may not be easy when the labelling is user-written in CLP. Search-tree abstraction will allow to do this.

2.7 Predicate properties

To specify abstract search-tree, properties of chosen atom of a node, among other kinds of search-tree properties, will be used. If the current store is σ, to verify that the chosen atom $a(X)$ satisfies the property $p(X)$ associated with a, is to prove that

$D \models \sigma \rightarrow p(X).$

A property will be specified by a predicate expressed in the same CLP language.

For efficiency reasons, its proof is implemented just by trying to execute the goal $p(X)$ in the context of the current store σ. Two cases are thus possible

[5] Transitive closure is "descendent", opposite is "parent", opposite transitive closure is "ancestor".

- The goal $p(X)$ fails. This shows that $\sigma \wedge p(X)$ is insatisfiable and therefore that

  $D \models \sigma \rightarrow \neg p(X)$,

  the property is false.

- The goals $p(X)$ succeeds. Even if the constraint solver is complete[6], it does not necessarily show that the property holds, but only that

  $D \models \exists (\sigma \wedge p(X))$.

  In this case one will consider that the property holds. The results will be an over approximation of the abstraction (it will be in some sense "less" abstracted than expected).

Here is a small example. Assume the current store is σ :`{X#>=Y,X#\=Y}` and the property p(X,Y): `X#<=Y`. A finite domain (incomplete) solver may find that `{X#>=Y,X#\=Y,X#<=Y}` is satisfiable, hence the property true, even if it is obviously not implied by the store.

In many cases however the truth of the properties will be correctly computed.
If the store consists in
`fd_domain([X,Y,Z],1,3),Y#=X+1,Z#=Y+1` and the property is `fd_size(X,1)`,
the solver succeeds on the store
`fd_domain([X,Y,Z],1,3),Y#=X+1,Z#=Y+1,fd_size(X,1).`
which is a way to test whether a finite domain variable is "ground".

Term properties may also be tested that way, testing for example that the predicate name of a predication `Term` is `user_labeling/1` can be expressed with the property

```
p(Term) :- functor(Term,user_labeling,1)
```

If the solver is complete, the implication problem becomes decidable. If each property p may have a negative cunterpart np such that always one of the property is true, when both cannot be true together, it is thus sufficient to test unsatisfiability of $\sigma \wedge p$ or $\sigma \wedge np$ (one of the formula is) to know whether p or np holds[7].

3 Tree Reduction

Search-tree abstractions may be obtained after several successive abstraction operations. "Abstraction" is viewed here as an operation which takes an initial tree and produces a *unique minimal* one in which the family orders are preserved. Minimality means that a minimal number of nodes is kept. Uniqueness means that for a given selection of nodes and arcs the "minimal" tree is unique.

We study here the definition of the unique minimal ordered tree obtained from an ordered tree in which nodes and arcs (without extremity node) have been selected. Such transformation, called here *tree reduction*, is studied on ordered trees independently from the nature of the labels of the nodes.

[6] A solver is complete iff when success implies the existence of a solution which satisfies the resulting constraint store.

[7] If $\sigma \wedge p$ is unsatisfiable, np holds, hence $\neg p$, otherwise if $\sigma \wedge np$ is unsatisfiable, p holds. See [14, 18] for assertion checkers with complete solvers.

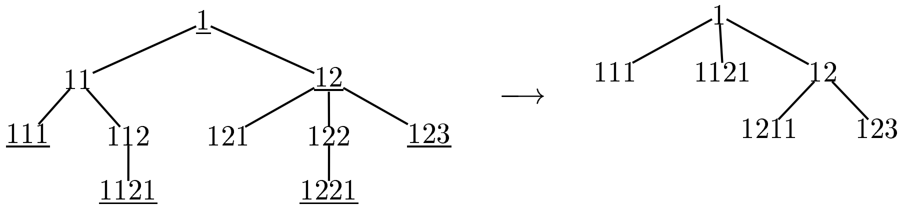


Figure 2: Preservation of family ordering (selected nodes are underlined)

3.1 Unique minimal Ordered Tree

An ordered tree is a tree with a family ordering. This ordering is completely defined by the lexical ordering defined on the set of nodes coded by a Dewey like notation. The "descendent" relation is defined by the prefix ordering (node uv is descendent of node u).

It is well-known that any subset of such coded nodes defines an ordered tree such that the filliation relations are preserved between the nodes of the subset (i.e. "descendent", "brother" or "gran cousin" nodes are still "descendent", "brother" or "gran cousin" in the subset).

A first transformation consists thus in selecting a subset of nodes and to take this subset as the (trivially) unique minimal tree. It is "minimal" in the sense of set ordering by inclusion. Figure 2 illustrates how the family order is preserved in this case (selected nodes are underlined).

Selecting nodes only is not sufficient: it may be more convenient to select arcs too (or instead). The reason is that it is sometimes easier to identify an interesting transition rather than an interesting computation state. We will see that selecting arcs (without selecting extremity nodes) is in fact a selection of nodes. If for example all arcs of a tree are selected, then the resulting tree is the same, i.e. all nodes are selected. Now if on a path only two non adjacent arcs are selected (without their extremity nodes), one would like to see in the resulting tree two adjacent (and descendent) arcs only, in order to satisfy the "minimality" objective. The question now is which node should be selected common to both arcs? Since there were assumed several nodes between the two arcs, there are several candidates.

In order to guarantee "minimality" and "uniqueness" of the result, one node only (according to the family ordering) among possible candidates will be selected.

Figure 3 shows the possible minimal tree when nodes and arcs are selected.

Notice that in Figure 3 non selected node **11** remains, to preserve the "gran cousin" relation, and there is an arc **(11, 1113)** which corresponds to a path in the original tree which contains non selected arc only. There is a 1-1 correspondence between the arcs of the initial tree and a subset of arcs of the reduced tree.

Notice also than one could have chosen node **111** instead of node **11**. The minimal ordered tree is not unique. There is however a "natural" choice which

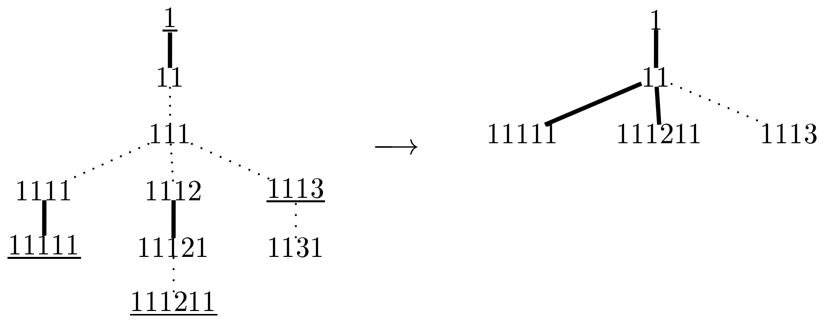


Figure 3: OT reduction with arc selection (selected arcs are bold).

guarantees uniqueness of the result: to choose the least node in the family ordering, i.e. the lower node of the upper arc.

The following definition summarizes the ideas above and is a declarative definition of minimal ordered tree obtained from nodes and arcs selection.

Definition 3.1 Reduced ordered tree

Given an OT and a subset of selected nodes (the root of the OT is always selected) and arcs, the reduced OT is the least subset of nodes such that:

1. all selected nodes are in the reduced OT,
2. to each original arc corresponds an arc in the reduced tree and there is a 1-1 correspondence between the set of original arcs and the set of their image in the reduced OT.
3. the filiation orders of the nodes and the arcs are preserved,

□

As we have seen, the reduced tree, as defined, is not always unique. There are several possible nodes between two non adjacent descendent arcs whose extremity nodes have not been selected. Considering the fact that the family orders must be preserved, candidate nodes are always nodes of a portion of path in the initial ordered tree. The problem arises only when there is an arc whose lower node is not selected. If there are several branches issued from that node with selected elements all nodes on the section common to all pathes are candidate (Nodes of path (**11,111**) in Figure 3). In order to make the reduced ordered tree unique, the least node (i.e. the lower node of the arc) will be selected[8].

If we define now an ordering on the subtrees as follows

[8]This choice is arbitrary. Any other candidate node could be chosen and condition 3 could be even better understood using lower node instead of the upper one. The upper node has been chosen here because it is attached to a unique selected arc (the lower node may not belong to any selected arcs or be common to several ones). We did not study yet the practical impact of a different choice on the abstractions.

Definition 3.2 Ordered Tree Ordering

Given an OT and two subset of nodes s1 and s2 defining respectively OT t1 and t2, t1 $<$ t2 iff

s1 has less elements that s2, or

s1 and s2 have same cardinality and for all elements of s1 and s2 which are different, say s1' and s2', for each element of s1' there is an element of s2' which is greater according to the total family ordering.

□

Hence the definition.

Definition 3.3 Minimal Reduced OT

Given an OT and a nonempty set of selected nodes (the root of the OT is always selected) and arcs, the minimal reduced OT is the least subtree such that:

1. all selected nodes are in the reduced OT,
2. to each original arc corresponds an arc in the reduced tree and there is a 1-1 correspondence between the set of original arcs and the set of their image in the reduced OT.
3. the filiation orders of the nodes and arcs are preserved,

□

It follows from the definition and explanations above that there is a unique minimal reduced OT (e.g. in Figure 3, the node **11** is chosen among candidates {**11**,**111**}).

3.2 Minimal Reduced Tree Construction

Proposition 3.1 *Construction of the minimal reduced OT Given an OT and a nonempty subset of selected nodes and arcs, there is a one-pass algorithm to compute the nodes of the minimal reduced OT.*

We assume that a subset of nodes and a subset of arcs is selected. If the lower node of a selected arc is selected too, there is no need to further action since there will be a corresponding arc in the reduced OT. So if all selected arcs have their lower node selected, there is nothing to do more than building a new tree from the subset of selected nodes.

A question arises if the lower node of an arc is not selected. Some descendent node should exist in order to ensure that there will be a corresponding arc in the reduced OT. Two cases only need to be considered

- There is one selected descendent node which is before all other selected elements (according to the total family ordering) in the subtree issued from that arc. In this case this selected node will serve as lower node for the corresponding arc in the reduced OT.

- In all other cases (for example two descendent arcs with no selected element between them or no selected elements in the subtree issued from the arc, or two selected nodes direct descendent in different pathes issued from that arc, ...), the lower node of the arc will be selected.

The purpose of the following algorithm is to decide whether the lower node of a selected arc must be selected. When this is known, the (unique) minimal reduced OT is exactly the subset of selected nodes. The construction of the minimal reduced OT is thus processed according to the following steps

1. node and arc selection,
2. selection of the lower nodes of selected arcs,
3. construction of the reduced tree from the set of selected nodes.

Each phase needs at least one visit (complete in the worst case) of the initial OT. For the first phase the selection criterion is applied to each node (and additionally arc). In the second phase, a complete (bottom-up) visit of the subtree is needed (in the worst case) to decide whether the lower node of a selected arc is selected or not. The third phase corresponds to a re-numbering of the nodes. Its efficiency depends on the used data structure, but actions will at least follow the total ordering of the nodes.

Algorithm

Each node has two flags denoted **Fa** and **Fb**. The values of **Fa** result from node and arc selection, the values of **Fb** are thus computed.

Fa $\subseteq$ {+,-,?}, respectively "selected", "not selected" and "to be decided" (this value corresponds to a lower node of a selected arc).

Fb $\subseteq$ {n,y,o}, respectively "nothing selected in the subtree", "one selected node only in direct descendence", "other".

The problem is to replace each "?" by "+" or "-".

Flag **Fa** is initialised in the first phase and updated in the second, flag **Fb** is set in the second phase.

We describe the different cases:

- leaf node:

  + $\rightarrow$ y

  - $\rightarrow$ n

  ? $\rightarrow$ +,n

- nonleaf node:

  + $\rightarrow$ y

  - and all children nodes flagged n $\rightarrow$ n

  - and one child only flagged y others flagged n $\rightarrow$ y

  - otherwise $\rightarrow$ o

  ? and one child only flagged y others flagged n $\rightarrow$ -,y

  ? otherwise $\rightarrow$ +,y

The point now is to observe that

1. given the values of **Fa**, the values of **Fb** can be computed in one (bottom-up) visit of the OT, hence property 3.1;
2. if node and arcs can be selected during a first visit, first two phases can be performed together in one complete visit of the initial OT.

It is here assumed that the value of flags **Fb** of the children of a node can be accessed with no significant cost from any nonleaf node.

4 Concrete search-tree Abstraction

We apply here the previous reduction to abstraction of concrete search-tree. We first introduce the notion of *CLP Choice-tree* which generalizes the notion of search-tree.

Definition 4.1 CLP Choice-tree
A CLP Choice-tree (CT) is a finite ordered labelled tree defined as follows.

- Each node is labelled by a constrained goal (σ, G).
- Each edge issued from a non-leaf node u labelled (σ, G) and with extremity node v labelled (σ', G') is labelled by a "local" store μ such that $\sigma' = \sigma \cup \mu$.
- There are three kinds of leaf-nodes:
  - *success*: successful computation.
  - *failure*: failed computation.
  - *nonterminated* otherwise.
- All nodes are totally ordered by a family ordering (with the two family partial orders: "descendent" and "brother").

There are three kinds of branches: *success*, *failed*, *nonterminated.*

- A *success branch* ends with a success leaf.
- A *failed branch* ends with a failure leaf.
- A *nonterminated branch* ends with a nonterminated leaf.

□

According to this definition, any finite CLP search-tree can be viewed as a CLP choice-tree. Labels of the nodes are resolvents, labels of the arcs are local stores. Success (resp. failed) leaves are marked success (resp. failed). All nodes are totally ordered by the visit order.

Assuming now that a subset of nodes and arcs of a CLP choice-tree has been selected, the corresponding reduced CLP choice-tree is defined as follows:

Definition 4.2 Reduced CLP Choice-tree

Given a CLP Choice-tree and a subset of selected nodes (the root is always selected) and arcs, the reduced choice-tree is the corresponding reduced ordered tree such that:

- Each node is labelled by the initial constrained goal (σ, G).
- Each edge issued from a non-leaf node u labelled (σ, G) and with extremity node v labelled (σ', G') is labelled by a "local" store μ such that $\sigma' = \sigma \cup \mu$.
- Leaf-nodes u are marked
  - *success* if u is root of a success branch in the initial choice-tree.
  - *failure* if u is root of a finitely failed subchoice-tree in the initial choice-tree.
  - *nonterminated* otherwise.

□

Proposition 4.1 *properties CST reductions*

Any choice-tree obtained after a finite number of reductions satisfies the properties:

- *All the nodes are nodes of the initial tree,*
- *Every arc corresponds to a path in the initial tree and is labelled by the union of local stores of this path,*
- *Every leaf marked "success" corresponds to a node whose subchoice-tree has at least one success branch,*
- *Every leaf marked "failure" corresponds to a node whose subchoice-tree is finitely failed,*
- *Every leaf marked "nonterminated" corresponds to a subchoice-tree with no success branch and at least one nonterminated branch.*

5 Node and arc selection

We finally turn to the question how to specify nodes and arcs selection. We present first a non exhaustive collection of criteria whose combinations allow to define sophisticated abstractions[9].

[9] The question whether any abstraction can be specified with the proposed collection of criteria is not considered here.

5.1 Node selection

Can be selected all nodes such that

- *Predicate indicator*: predicate name and arity. The chosen atom of the resolvent corresponds to this predicate.

  It is expressed by `[not_]node_name(n,l)` where `n` is a search-tree node and `l` a list of predicate indicators [5].

- *A property of a predicate*: The chosen atom a of the resolvent satisfies the given property: if σ is the current store it is a property of the constrained atom (σ, a). Example: a given argument is of type "finite domain or list of finite domain variables".

  It is expressed by `[not_]node_property(n,l)`, where `n` is a search-tree node and `l` a list of properties, understood as their disjunction. A property is defined by the user as a unary predicate whose single argument is the chosen atom of the resolvent of node `n`.

- *success leaf*: to be a success leaf, expressed by `success_leaf(n)`.

- *failure leaf*: to be a failed leaf, expressed by `failure_leaf(n)`.

- *success node*: to be a node, root of a success branch, expressed by `success_node(n)`. It includes the success leaves.

- *failure node*: to be root of a failed subsearch-tree, expressed by `failure_node(n)`. It includes the failure leaves.

- *leaf*: to be a leaf, expressed by `leaf(n)`.

- *nonleaf*: not to be a leaf, expressed by `nonleaf(n)`.

- *least failed tree root*: to be root of a "bigest" failed subtree, expressed by `least_failed_tree_root(n)`.

- *node of some selected arc*: there is the possibility to specify which nodes are selected with a selected arc: (*no node, upper, lower node* or *both nodes*). The way to express it is with arc selection.

- *ancestor*: to have an ancestor whose chosen atom of the resolvent satisfies the given property (subtree selection).

  It is expressed by `[not_]ancestor_property(n,l)`, where `n` is a search-tree node and `l` a list of properties, understood as their disjunction.

A node can be selected positively (e.g. `success_leaf(n)`) or negatively (e.g. `not(success_leaf(n))` which is equivalent to `failure_leaf(n)`). Each criterion has a corresponding negative criterion which is equivalent to its negation in Prolog (not provable).

5.2 Arc selection

Can be selected all arcs such that

- *A property of the chosen atom of the lower node*: the chosen atom of the resolvent of the lower node satisfies the property.

  It is expressed by `in_arc(n,l,s)`, where `n` is a search-tree node, `l` a list of properties, and `s` $\in$ { `n,u,l,b` }, called *node selector*, (`n,u,l,b` stand respectively for "no node", "upper", "lower" and "both nodes").

- *A property of the chosen atom a of the upper node with local constraints*: if the current store of the lower node is σ, the chosen atom a is such that the constrained atom (σ, a) satisfies the given property.

  It is expressed by `out_arc(n,l,s)`, where `n` is a search-tree node, `l` a list of properties of the parent node, and `s` the node selector.

- *A label property*: the label of the arc satisfies some property (the store includes some constraint or the local store has some property).

  It is expressed by `label_property(l,s)`, where `l` a list of properties, and `s` the node selector. Properties depends on the nature of the labels.

- *A program arc*: a clause and a body (non constraint) predication number (first predication corresponds to the arc $h...a_1$, others are $a_i...a_{i+1}$ if a_i is defined by facts only).

  It is expressed by `clause_arc(c,i1,i2,l,k,s)`, where `c` denotes a clause of the user's program, `i1` and `i2` are two successive integers between 0 and the number of atoms in the body of the clause, `l` is a list of properties, `k` denotes the node to which this property applies (`k` $\in$ {`u,l`}), and `s` the node selector.

There is no negative counterpart for arc selection (an arc characterizes a computation step and its negation has no particular meaning).

5.3 Specification of abstractions

Finally criteria can be combined in boolean expressions: conjunction ($c1 \vee c2$) means that nodes and arcs satisfying both criteria are selected, disjunction ($c1 \wedge c2$) means that the union of nodes and arcs satisfying one of the criteria is selected. Negation (*not* c) may also be used. Expressions should be written in such a way that no arc selection criterion occurs negatively in the expression and no arc and node criteria are related by a conjunction (the simplest way is to restrict to a disjunction of two sub expressions). The semantics is as follows: any expression is considered as equivalent to a disjunctive normal form in which all negative properties occurrences are replaced by a positive occurrence of the negative counterpart and no conjunction contains node and arc criteria.

Here are some examples:

`success_node(N)`: selects all nodes which lead to a success branch only.

`success_node(N)` $\wedge$ `leaf(N)`: defines the tree with the root and as many children they are successes (equivalent to `success_leaf(N)`).

`node_name(N,[go/1])` $\vee$ `out_arc(N,[prop],n)`
with prop/1 possibly defined by the user as:

```
prop(X) :- X=..[u_labeling,A,_,_],
           integer(A).
```

specifies that are selected either nodes whose chosen atom corresponds to predicate `go/1` or arcs whose chosen atom of the upper node corresponds to predicate `u_labeling/3` such that its first argument is an integer.

Abstraction may be composed (i.e. a tree abstraction may need some further reduction). This is needed since arc selection may re-introduce nodes which have not been explicitely selected. For example if one selects
`success_node(N)` $\vee$ `out_arc(N,[prop],n)`
We may obtain some failure nodes corresponding to arcs selected in failed subsearch-tree. Thus the way to obtain success nodes only in the final abstraction is to reduce again the search-tree with criterion `success_node(N)`. A simple way to define non redundantly such abstraction is by composing abstractions first with criterion `out_arc(N,[prop],n)` then with `success_node(N)`.

6 Implementation

We started an implementation of search tree visualisation and abstraction for Calypso, our CLP(FD) platform.

The initial CLP choice-tree corresponds to the CLP search-tree as described in Section 2.1. We call it the “concrete search-tree” (CST). This CST may not be finite or just too big to be tractable. Therefore we need some “termination” criteria. Such criteria are also a way to impose some limitations to the size of the CST in case of non terminating or too long computation.

Termination criteria are:

- Maximal depth (default value: unlimited),
- Number of solutions (default value: unlimited),
- Number of failure nodes (default value: unlimited),
- Number of nodes (default value: unlimited)
- Number of leaves (default value: unlimited)
- Execution time (default value: unlimited)

- Ressource exhausted,
- User interrupt.

In order to be able to deal with large CST, only few informations are stored at each node (basically the chosen clause, and the chosen predication). To obtain any other information at some node (for example the current store) needs recomputation along the path to this node. Node and arc selection is performed for each different abstraction after the CST has been computed once.

Different abstractions can be thus derived from the CST, which may not be displayed.

Presently all node selection criteria except "ancestor" and all arc selection criteria except "label property". Both need further investigations to be implemented efficiently.

The entailment is implemented as described in Section 2.7. As the finite domain solver of Calypso is not complete, over appoximations of the abstractions are, in principle, computed.

7 Examples

The purpose these examples is to illustrate comprehensively the result of an abstraction: from "Prolog" concrete search-trees displayed in Figure 4 and corresponding to two programs (`go1/1` and `go2/1`) we obtain two abstractions, displayed in Figure 5, which are a representation of the labelling part in which each level corresponds to enumeration of one finite domain variable.

```
go1(L) :- L = [X1, X2, X3], fd_domain(L, 1, 5),
          user_labeling(L), X1 + X3 #<= X2.
go2(L) :- L = [X1, X2, X3], fd_domain(L, 1, 5),
          X1 + X3 #<= X2, user_labeling(L).

user_labeling([]).
user_labeling([X | L]) :- fd_min(X, Min), user_labeling_cont(X, Min, L).

user_labeling_cont(X, Min, L) :- X #= Min,  user_labeling(L).
user_labeling_cont(X, Min, L) :- X #> Min,  user_labeling([X | L]).
```

All (finite domain) variables have 3 possible values. Labelling is performed on unconstrained domains, hence leading to 81 possible solutions. The constraint finally posted eliminates some of them (failed branches).

To obtain an abstraction which represents a labelling CLP choice-tree, the easiest way is to select arcs corresponding to value assignment of finite domain variables only (i.e. arcs corresponding to computation steps where a finite domain variable becomes ground). The following criteria is thus used:

`out_arc(N,[prop],none)`

where

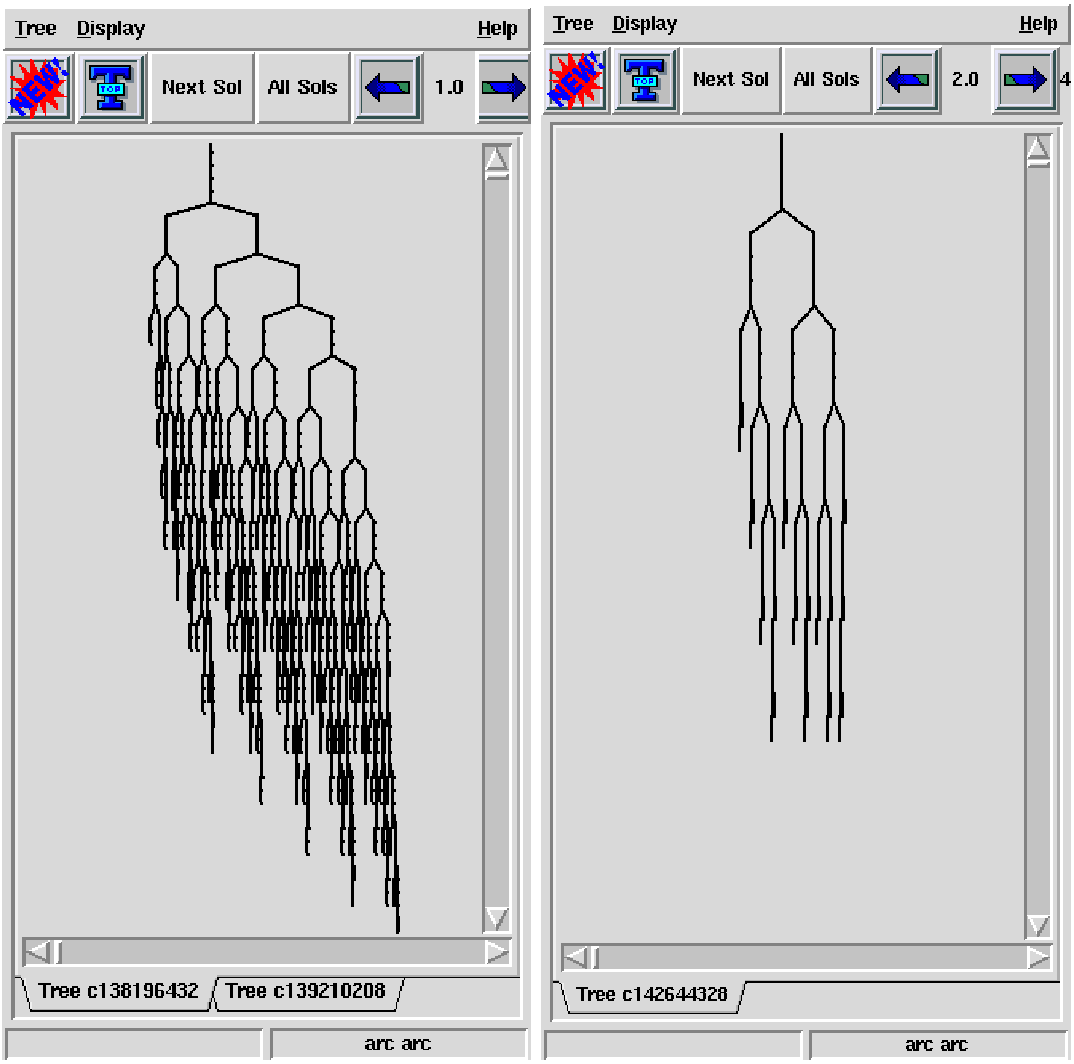


Figure 4: Concrete search-trees

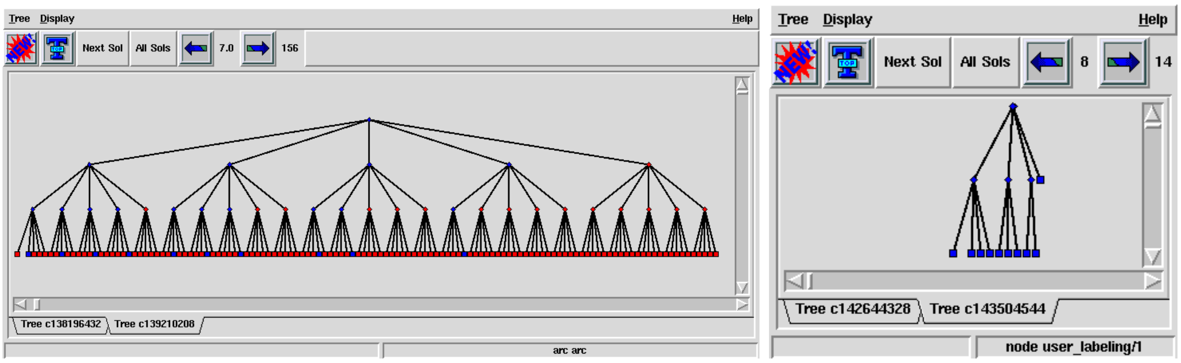


Figure 5: Abstract search-trees

```
prop(Term) :- functor(Term, 'user_labeling_cont', 3),
              arg(1, Term, L), integer(L).
```

It selects all arcs issued from `user_labeling_cont/3` and such that the finite domain variable is assigned to some value in the next computation step (i.e. it describes value assignment of finite domain variables: `X` assigned to `Min` or, if the domain of `X` has 2 values only, `X` assigned to the value greater than `Min`).

The fact to select arcs without node only makes automatic selection of lower nodes of the arcs (whose chosen atom is `user_labeling/1`). It would however not be possible to make the same selection using a property of `user_labeling/1`.

Both displayed abstractions show improvements resulting from propagation when constraints are known before labeling. It can be immediately observed in particular that labeling of two variables is sufficient to fix all values.

8 Conclusion

In this paper we have presented a method for abstracting search-trees of CLP programs and for specification of these abstractions. We have illustrated the approach by extracting labelling trees from CLP search-trees in which the labelling was user programmed.

The ideas presented here are currently implemented as part of a debugging environment for Calypso, allowing boolean expressions of criteria to specify abstractions. Implementation aspects, not described here, will be considered in further report, in particular possibilities to optimize the abstraction process in presence of complex expression of criteria.

This environment includes many other functionalities, not presented here, as facilities to display the AST, node informations, constraints store and finite domain variable evolution, etc, ... and other debugging systems. Principles and details can be found in deliverables of the DiSCiPl project[3, 4, 8]. Results on experimentations will be presented in forthcoming works. An interesting aspect of the AST is the possibility (not developed here) to compose abstractions.

Search-tree abstraction could be considered from a different point of view: search-tree of an abstracted program. This could be also an interesting approach. In fact some properties may be captured by an abstract program (like predicate types for example), or an abstract program may terminate when the original one does not, and allow to study some properties. We did not choose this approach since our purpose was to find possibilities to study concrete search-trees and the definition of abstract domains for performance debugging needs further investigations.

The use of predicate properties needs some efficient method for verification, as the number of nodes which must be tested may be enormous. If properties are expressed with constraints this leads to the entailment problem. Meta constraints in Calypso allow to solve this problem efficiently in some cases [9]. This introduces restrictions on the use of constraints in properties. We prefered to test directly properties with the risk to obtain over approximations of abstractions. First examples show the relevance of this approach in terms of efficiency and of adequation of the abstractions.

References

[1] Cosytec. *CHIP user Manual, beta release* Sept. 1998.

[2] D. Diaz. *Calypso user Manual* to appear.

[3] The DiSCiPl Consortium. Esprit LTR (Task 4.2) Project Nb 22532 *Debugging Systems for Constraint Programming.* http://discipl.inria.fr

[4] The DiSCiPl Consortium, ed. P. Deransart. *CP Debugging: Tools* (D.WP1.1.M1.1-2). July 1997. On request.

[5] P. Deransart and A. Ed-Dbali and L. Cervoni. *Prolog: The Standard, Reference Manual.* Springer Verlag, 1996.

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

[7] F. Bueno and P. Deransart and W. Drabent and G.Ferrand and M. Hermenegildo and J. Maluszynski. *A Systematic Approach to Debugging through Semantic Approximations*, (Extended Abstract). Workshop on Automated and Algorithmic Debugging, In Proceedings AADEBUG'97, 1997.

[8] F. Benhamou and F. Goualard. *Debugging Programs in the Constraint Programming Scheme.* DiSCiPl deliverable D.WP3.1.M2.1, October 1998.

[9] B. Carlson and M. Carlsson and D. Diaz *Entailment of Finite Domain Constraint* ICLP'? ftp://ftp.inria.fr/INRIA/Projects/loco/publications/entail.ps

[10] M. Meier. *Debugging Constraint Programs.* Procceedings of Principles and Practice of Constraint Programming (CP95), Cassis, pp 204–221, 1995.

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

[12] PrologIA. *Prolog IV Manual, Tutorial, Basic Concepts* 1997.

[13] PrologIA. *Prolog IV User Manual, beta release* Sept. 1998.

[14] PrologIA. Assertions (Chapter of the Prolog IV User Manual, beta release) DiSCiPl public deliverable D.WP2.2.M2.1-5, http://discipl.inria.fr, Oct. 1998.

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

[16] C. Schulte. *Oz Explorer: A Visual Constraint Programming Tool.* Proceedings of the Fourteenth International Conference on Logic Programming (ICLP'97), Leuven, The MIT Press, pp 286–300, July 1997.

[17] C. Schulte. *Programming Constraint Inference Engine.* Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (ICLP'97), Linz, Springer Verlag, LNCS 1330, pp 519–534, October 1997.

[18] K. Marriott, P. Stuckey. Programming with constraints, an introduction. The MIT Press, 1998.