Debugging Systems for Constraint Programming

ESPRIT 22532

Task T.WP3.2: **Declarative Debugging in Constraint Programming**

Delivrable D.WP3.2.M2.1

Implementation of the Declarative Diagnoser module for Calypso

Alexandre Tessier

LIFO, rue Léonard de Vinci, BP 6759, 45067 Orléans Cedex 2, France
Alexandre.Tessier@inria.fr
http://www.univ-orleans.fr/SCIENCES/LIFO/Members/tessier/

`http://discipl.inria.fr/`

Contents

List of Figures

List of Algorithms

1 Introduction

The general principle of declarative diagnosis of constraint logic programs is explained in [1]. An extension of the negative part to any computation rule is decribed in [5]. This report clarify how it has been implemented in Calypso, the INRIA plate-forme.

We briefly remind the reader of this principle. For constraint logic programs two semantics may be considered:

Positive Semantics

A *positive answer* to a goal $\leftarrow g$ is the traditional notion of answer, that is the store (set of contraints) S computed by a finite success SLD derivation from the goal $\leftarrow g$.

The notion of *positive computation* is the notion of SLD-derivation.

The relation between the goal $\leftarrow g$ and a positive answer S is formalized using the implication $S \rightarrow g$.

Negative Semantics

The *negative answer* to a goal $\leftarrow g$ is the disjunction of the positive answers to the goal $\leftarrow g$, that is the set of stores $\mathcal{Z}$ computed by a finite SLD tree (the set of stores associated to the success leaves of the SLD tree).

The notion of *negative computation* is the notion of SLD-tree.

The relation between the goal $\leftarrow g$ and the negative answer $\mathcal{Z}$ is formalized using the implication $g \rightarrow \mathcal{Z}$.

Computations are described using the notion of *proof tree skeleton* (see [2, 8]). A positive computation is described by a sequence of proof tree skeletons. A negative computation is a tree whose nodes are proof tree skeletons.

A notion of (positive or negative) proof tree is associated to each (positive or negative) answer.

Sometimes, a (positive or negative) computation provides a symptom: that is, a (positive or negative) answer which is not expeceted. There is two kinds of symptom:

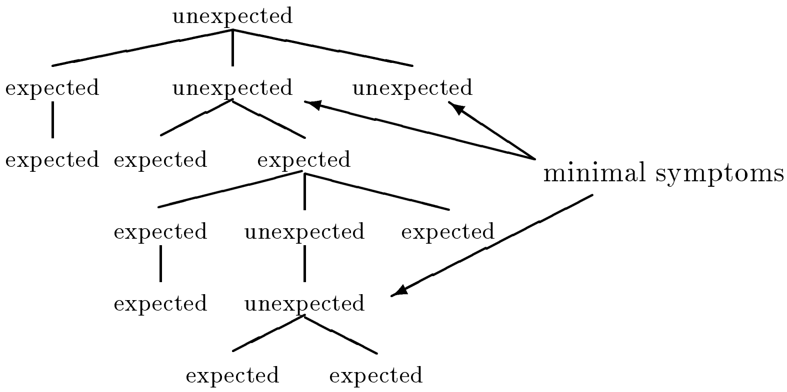

Figure 1: Notion of minimal symptom in a proof tree

Positive Symptom
: when $S \to g$ is not in the expected semantics (positive symptom correspond to the classical notion of wrong answer);

Negative Symptom
: when $g \to \mathcal{Z}$ is not in the expected semantics (negative symptom correspond to the classical notion of missing answer).

Let us consider the proof tree T which corresponds to a (positive or negative) symptom.

To each node N of the proof tree is associated a formula $\mathcal{F}(N)$. Let N be a node of the proof tree and $N_1, \ldots, N_n$ be the children of N; the point is that: $\mathcal{F}(N_1) \wedge \cdots \wedge \mathcal{F}(N_n) \to \mathcal{F}(N)$ is a consequence of the program (more precisely, the if-and-only-if part of the program, see [1]). The formula associated to the root of the proof tree is the (positive or negative) unexpected answer.

Some formulas associated to the nodes of the proof tree are unexpected (others are expected), they are also (positive or negative) symptoms.

The goal of declarative diagnosis is to find a minimal symptom (wrt the parent relation in the proof tree) and then to show to the user the piece of code used to demonstrate that $\mathcal{F}(N_1) \wedge \cdots \wedge \mathcal{F}(N_n) \to \mathcal{F}(N)$, where N is the node associated to the minimal symptom and $N_1, \ldots, N_n$ are its children. Figure 1 shows a proof tree with 3 minimal symptoms.

The point is that there is no difference between positive and negative diagnosis, except the notion of proof tree, the notion of formula associated to a node and the notion of error associated to a minimal symptom. In the implementation, a wide part of the module is common to the positive and negative diagnosis.

Another point is that we just store the SLD-tree (in fact, mainly its structure, some informations are recomputed when needed).

The proof tree, in both cases (positive and negative), is known by its root which corresponds to a node of the SLD-tree and its subtrees which also corresponds to nodes of the SLD-tree. We explain in section 2.1 how to compute the parent relation of a (positive or negative) proof tree from the SLD-tree.

The parts which are not common to positive and negative diagnosis concerns:

- the queries (formulas) associated to the nodes of a proof tree;
- the error associated to a minimal symptom in the proof tree;
- the windows used by the `tk` interface.

So, perhaps 80% or 90% of the module are common to positive and negative error diagnosis.

2 Calypso Declarative Diagnoser

We discuss in this section the main features of the Declarative Diagnoser module of Calypso.

2.1 Proof Tree

Proof tree is the central notion of declarative diagnosis. In the next sections, we confuse, in general, the notion of node and tree, that is, a proof tree and its root are denoted by the same object.

A node of a proof tree contains several informations:

- the list of its children (a list of proof trees is called a forest);
- a field which indicate if the node is expected, the possible values of this field are:

  `expected(user)`
  : when the user said that the node is expected;

  `expected(list)`
  : when the predicates is in the correct predicate list given by the user;

  `expected(ass(N))`
  : when the assertion `N` implies that the node is expected;

  `expected(system)`
  : when the predicates is built-in;

  `unexpected(user)`
  : when the user said that the node is unexpected;

  `unexpected(ass(N))`
  : when the assertion `N` implies that the node is unexpected;

  `dontask(user)`
  : when the user said that this node must not be questioned;

  `dontask(list)`
  : when the predicates is in the list of predicates that the user does not want to answer (the list is given by the user);

`unknown(user)`
: when the predicate is not a built-in, any current assertion implies that the node is expected or unexpected and the user has not been questioned about the node.

This list of values may be extended in the future.

- a flag which can be used by the diagnoser to determine the chosen node according to some strategy.

As we said before, (positive or negative) proof trees are not built by the declarative diagnoser. The nodes of a (positive or negative) proof tree corresponds to some nodes of the SLD-tree. The problem is to determine

1. the nodes of the SLD-tree which corresponds to the children of a node of the proof tree;

2. the formula associated to a node of the SLD-tree.

The computation rule is assumed to be without co-routining (e.g. the standard computation rule of Prolog). The extension presented in [5] is not actually implemented. We will see that the main notion useful for the first point is the notion of *erasing*, that is, a node X of the SLD-tree being given, let $a_1, \ldots, a_n$ be the resolvant associated to the node and a_i be the chosen atom, the question is: which are the descendants of X where a_i is fully solved (the resolvant of these nodes are $a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n$).

Definition 1 *Let N be a node of the SLD-tree.*

1. *we denote by* `store(`N`)` *the set of constraints accumulated from the root of the SLD-tree to the node N;*

2. *we denote by* `select(`N`)` *the atom selectioned at the node N;*

3. *we denote by* `erased(`N`)` *the set of nodes where* `select(`N`)` *is erased from the resolvant.*

The parent relation of a (positive or negative) proof tree and the formulas assoicated to its nodes are computed on demand from the SLD tree using the previous functions.

2.1.1 Positive Proof Tree

A node of a positive proof tree is labeled by a constrained atom $a \leftarrow S$ (see [1] for more details). The store S is the same for each node of a positive proof tree, it is the store of the unexpected answer; thus the store is recorded once. Atom a corresponds to the selected atom of the corresponding node in the SLD tree.

The query associated to the node is: "$S \rightarrow a$ expected?". If the free variables of S which are not free in a are $\tilde{y}$ the query can be simplified into: "$\exists \tilde{y}\, S \rightarrow a$ expected?".

The store $\exists \tilde{y}\, S$ should be simplified. Several techniques may be used (see [3], English abstract [4]):

- quantifier eliminations;
- redundant constraint eliminations;
- symbolic transformations.

How to determinate the parent relation of a positive proof tree from a success branch of the SLD-tree? Let $N_0, N_1, \ldots, N_n$ be the sequence of nodes of the success branch of the SLD-tree.

The root of the positive proof tree is the root of the SLD-tree, that is N_0. The children of a node N_i are:

1. if the chosen atom at the node N_i is erased at the node N_{i+1} then N_i has no child in the positive proof tree
2. else the first child of N_i is N_{i+1}. In the following, we denote by X the last child founded for N_i. Actually X is N_{i+1}.
   (a) if the node where the chosen atom in X is erased is the node where the chosen atom in N_i is erased then N has no other child
   (b) else, the node where the chosen atom in X is erased is the next child of N_i (the next X is this node).

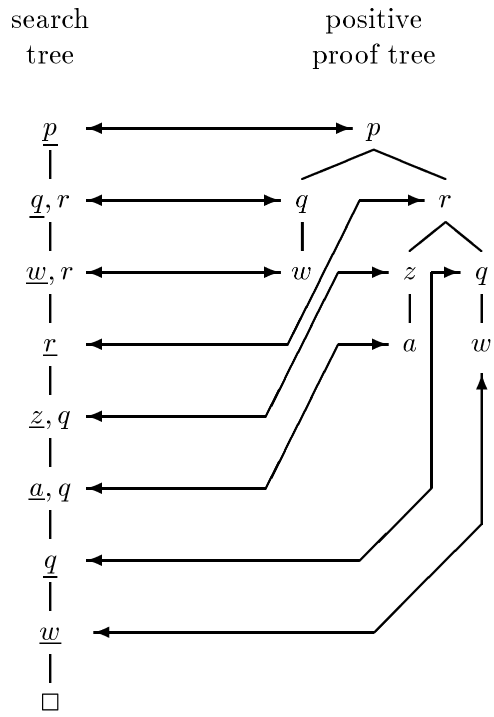

Figure 2: From a success branch of the SLD-tree to a positive proof tree

Example 2.1 From SLD-tree to positive proof tree

Let us consider the following program:

$$
\begin{array}{l}
p \leftarrow q, r \\
r \leftarrow z, q \\
q \leftarrow w \\
z \leftarrow a \\
w. \\
a.
\end{array}
$$

Figure 2 shows the SLD-tree associated to the goal $\leftarrow p$, using the standard computation rule. The selected atom of a node is underlined. And it shows the positive proof tree associated to the success branch of the SLD-tree.

How to determinate the query associated to a node of the positive proof tree from the SLD-tree? The query associated to a node N of the positive proof tree is $S \rightarrow a$ where S is the store associated to the success leaf of the branch of the SLD-tree which corrsponds to the positive proof tree, and a is the selected atom at the node of the success branch which correspond to N.

2.1.2 Negative Proof Tree

A node of a negative proof tree is a local cover of atom $S \wedge a \rightarrow \mathcal{Z}$ (see [1]). To each node of the negative proof tree corresponds a node of the SLD-tree (and vice-versa, except for the success leaves).

Let $S \wedge A$ be the resolvant which labels a given node N of the SLD-tree (S is the store and A is the conjunction of atoms), and a be the selected atom in A. The node of the negative proof tree which corresponds to N is labeled by $S \wedge a \rightarrow \mathcal{Z}$ where $\mathcal{Z}$ is the disjunction of the answer to the goal $S \wedge a$.

When the computation rule used is without co-routining the member of the disjunction $\mathcal{Z}$ can be found in the SLD-tree, let A' be the conjunction obtained when a is removed from A: $\mathcal{Z} = \bigvee\{S' \mid S' \wedge A'$ is the label of a descendant N' of N in the SLD-tree, and there exists no node in the path from N to N' labeled by $S'' \wedge A'$ where S'' is a store$\}$. That is, the members of $\mathcal{Z}$ are the stores associated to the nodes where a is erased from the resolvant.

The query associated to the node of the negative proof tree which corresponds to N is: "$S \wedge a \rightarrow \mathcal{Z}$ expected?" or, in other words, "does an answer to $S \wedge a$ is missing in $\mathcal{Z}$?". Again, the stores of $\mathcal{Z}$ should be simplified.

How to determinate the parent relation of the negative proof tree from the SLD-tree? The main differences with respect to [1] is that only one kind of node are in the negative proof tree: the nodes whose associated formula has one atom, the other nodes are skipped. This is not a problem because, as explained in [1], the other nodes does not depend on the expected semantics, they cannot be minimal symptoms.

The root of the negative proof tree is the root of the SLD tree. The children

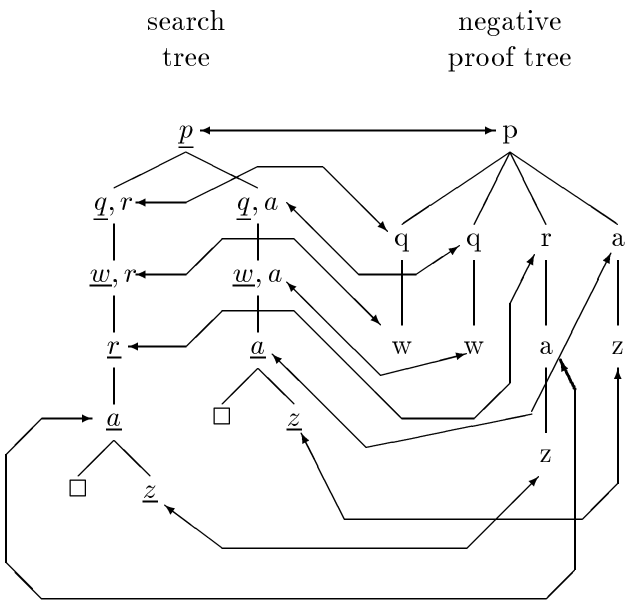

Figure 3: From the SLD-tree to the negative proof tree

of a node N of the negative proof tree are:

1. the children $N_1, \ldots, N_n$ in the SLD tree of the node which corresponds to N except those where the selected atom in N is erased;
2. and, for each N_i, let $a_1, \ldots, a_k$ be the atoms introduced in the resolvant associated to N_i in the SLD tree and a_j be the selected atom in $a_1, \ldots, a_k$, the nodes where $a_1, \ldots, a_{j-1}, a_{j+1}, \ldots, a_k$ are selected.

Example 2.2 From SLD-tree to negative proof tree ______________________

Let us consider the following program:

$$
\begin{array}{l}
p \leftarrow q, r \\
p \leftarrow q, a \\
r \leftarrow a \\
q \leftarrow w \\
w. \\
a. \\
a \leftarrow z
\end{array}
$$

Figure 3 shows the SLD-tree associated to the goal $\leftarrow p$, using the standard computation rule. The selected atom of a node is underlined. And it shows the negative proof tree associated to the SLD-tree, and the correspondence between the nodes of the trees.

How to determinate the query associated to a node of the negative proof tree from the SLD-tree? The query associated to a node N of the negative proof tree is $a \to \mathcal{Z}$ where a is the selected atom at the nodes of the SLD-tree which correspond to N and $\mathcal{Z}$ is the set of stores associated to each node of the SLD-tree where a is erased (starting from the correspondant of N).

2.2 List of Correct Predicates

Built-in predicates are known as correct predicates, but also user predicates could be known as correct. For example, the static analysis proved there correctness. Another example is when the user is convinced that some predicates are correct. The nodes whose predicate is in the list of correct predicates are labeled by `expected(list)`.

The point is that the user can change dynamically the list of correct predicates.

If the user add a predicate, each `unknown(_)` or `dontask(_)` node concerning the predicate is labeled by `expected(list)`; if a node is `unexpected(user)` then the problem is given to the user: either remove the predicate of the correct predicate list or label the node as `expected(user)`; if a node is `unexpected(ass(N))` the problem is also given to the user: either remove the predicate of the correct predicate list or remove the assertion N and then remove all the expected nodes due to this assertion.

When the user removes a predicate of the list, each `expected(list)` node concerning the predicate is labeled by `unknown(user)`.

2.3 List of Predicates which must not be questioned

It is possible that the user does not want to be questioned on some predicates. For example, the user does not know if a predicate is correct but she is sure that the predicate is not responsible for the current symptom. Another example is when the semantics of the predicate is very intricate and the user want to suspend the queries on the predicate as long as possible.

Again the user can change dynamically the list of predicates which must not be questioned.

If the user add a predicate to the list, each `unknown(_)` node concerning the predicate becomes a `dontask(list)` node.

If the user remove a predicate of the list, each `dontask(list)` node concerning the predicate becomes an `unknown(user)` node.

2.4 Assertions used by the Declarative Diagnoser

There is four kinds of assertions:

`:- inmodel A <= S`
: all solution of the store `S` is a solution of the atom `A` in the (positive) expected semantics;

`:- outmodel A <= S`
: there exists a solution of the store `S` which is not a solution of the atom `A` in the (positive) expected semantics;

`:- inmodel A => Z`
: for each solution of the atom `A` in the (negative) expected semantics there exists a store `S` of the disjunction of stores `Z` which is satisfied by the solution;

`:- outmodel A => Z`
: there exists a solution of the atom `A` in the (negative) expected semantics which is solution of no store of the disjunction of stores `Z`.

The `inmodel` assertions are described in [6]. We add the `outmodel` assertions.

Of course, these kinds of assertions can answer to the queries of the declarative diagnoser:

- Let us consider a query "$S \to a$ expected?"
  If there exists an assertion "`:- inmodel a <= S1`" such that $S \to$ `S1` then $S \to a$ is expected.
  If there exists an assertion "`:- outmodel a <= S1`" such that `S1` $\to S$ then $S \to a$ is unexpected.
- Let us consider a query "$a \to \mathcal{Z}$ expected?"
  If there exists an assertion "`:- inmodel A => Z1`" such that `Z1` $\to \mathcal{Z}$ then $a \to \mathcal{Z}$ is expected.
  If there exists an assertion "`:- outmodel A => Z1`" such that $\mathcal{Z} \to$ `Z1` then $a \to \mathcal{Z}$ is unexpected.

Obviously, we assume that the set of assertions is consistent, that is, there exists a model of the set (the intended model should be a model of the set of assertions).

The main problem in this part is the entailment problem. Actually it is implemented when the two stores are syntactically equal (up to a variable renaming). We see again how much it is important to spend research on the entailment problem.

2.5 Diagnosis Strategies

As explained in [1] any strategy can be used in order to localyse a minimal sympton in the (positive or negative) proof tree.

The strategies actually implemented are

`td`
: the top-down strategy, see section 4.1

`bu`
: the bottom-up strategy, see section 4.2

`dq`
: the divide-and-query strategy, see section 4.3

`ne`

: the nearby-error strategy, see section 4.4

`ug`

: the user-guided strategy, see section 4.5

Other strategies are implemented, but not described in this document (they have not been sufficiently tested), for example:

`tm`

: the top-most strategy, similar to the top-down strategy except that it deals with `dontask(_)` nodes;

`bm`

: the bottom-most strategy, similar to the bottom-up strategy except that it deals with `dontask(_)` nodes.

3 Interface between the declarative diagnoser and calypso

The main predicates of the declaretive diagnoser called from calypso are:

`'$dd_init'(Tree)`

This predicate initialize the declarative diagnoser.

It opens, if necessary, the tk windows (initialize the declarative diagnoser interface).

Next, it initializes the proof tree `Tree`:

1. the root of `Tree` is `unexpected(user)` (because the proof tree is a computed symptom);
2. the built-in predicates are `expected(system)` (they are always assumed to be correct);
3. the nodes concerning a predicate of the list of correct predicates given by the user are `expected(list)`;
4. the nodes concerning a predicate of the list of predicates which must not be questioned are `dontask(list)`;
5. when the assertion `N` implies that a node of the `Tree` is expected, then the node is `expected(ass(N))`;
6. when the assertion `N` implies that a node of the `Tree` is unexpected, then the node is `unexpected(ass(N))`.

Finally, it searches a minimal symptom in `Tree`. If it found a minimal symptom, it shows the associated error.

`'$dd'(Tree)`

This predicate search the chosen node in `Tree` according to the current strategy.

If the current strategy cannot choose a node, it tells the user to change the strategy. Otherwise, it questions the user about the node.

If the user answers that the node is `expected` or `unexpected`, it add the corresponding assertion, and checks if this assertion implies some answer for other `unknown` nodes.

Finally, it searches a minimal symptom in `Tree`. If it found a minimal symptom, it shows the associated error.

`'$dd_close'`

It closes the declarative diagnoser (essentially closes the tk windows).

4 Strategies actually implemented

In the figures of this section the nodes of the proof trees are labeled by:

`Y`
: when the node is `expected(_)`;

`N`
: when the node is `unexpected(_)`;

`X`
: when the node is `dontask(_)`;

`?`
: when the node is `unknown(_)`.

The main predicate is: `'$dd_query'(Tree, NodeFound)`. The predicate chooses an `unknown` node in `Tree` according to the current strategy and ask the user if the node is expected. `NodeFound` correspond to the fact that the user has been questioned on a node.

4.1 Top-Down Strategy

```
top-down(Tree)
    if the root of Tree is unknown then return Tree
    elseif the root of Tree is unexpected then
        Forest ← children(Tree)
        while Forest is not empty
            let Tree1 be the first tree of Forest
            remove Tree1 from Forest
            SubTree ← top-down(Tree1)
            if SubTree ≠ notfound then return SubTree
    return notfound
```

Algorithm 1: Top-Down Strategy

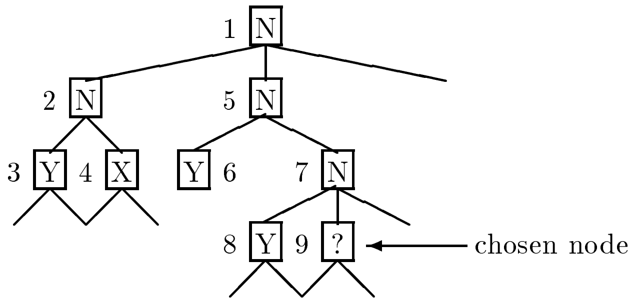

Figure 4: Top-Down Strategy

The top-down strategy follows a path of `unexpected` nodes from the root until it reaches an `unknown` node. Because of the `dontask` nodes, the strategy does not always conclude.

The top-down strategy is explained by algorithm 1, where the function `top-down` returns either a tree rooted by the chosen node or `notfound`. The function `top-down` returns `notfound` when it cannot conclude, that is it cannot choose a node because the user does not want to be questioned on some nodes (`dontask` nodes).

Example 4.1 Top-Down Strategy ______________________________

Figure 4 shows an example. The numbers are the order of the node visit.

The search start from the root of the proof tree:

the root (node 1) is always `unexpected`, next node is the 1st child of the root (node 2);

node 2 is `unexpected`, next node is the 1st child of node 2 (node 3);

node 3 is `expected`, next node is the 2nd child of node 2 (node 4);

node 4 is `dontask`, node 2 have no other child, next is the 2nd child of the root (node 5);

node 5 is `unexpected`, next node is the 1st child of node 5 (node 6);

node 6 is `expected`, next node is the 2nd child of node 5 (node 7);

node 7 is `unexpected`, next node is the 1st child of node 7 (node 8);

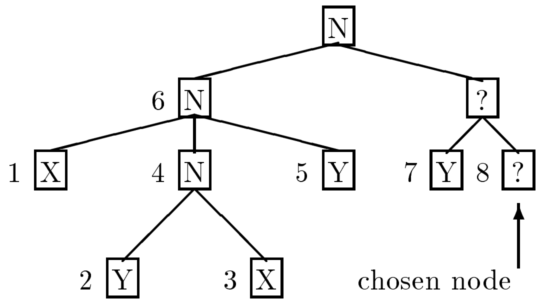

Figure 5: Bottom-Up Strategy

node 8 is `expected`, next node is the 2^{nd} child of node 7 (node 9);

node 9 is `unknown`, node 9 is the chosen node.

4.2 Bottom-Up Strategy

```
bottom-up(Tree)
    Forest ← children(Tree)
    while Forest is not empty
        let Tree1 be the first tree of Forest
        remove Tree1 from Forest
        SubTree ← bottom-up(Tree1)
        if SubTree ≠ notfound then return SubTree
    if the root of Tree is unknown then return Tree
    return notfound
```

Algorithm 2: Bottom-Up Strategy

The bottom-up strategy starts from the leaves. It is a post-fix traversal of the tree which returns the first `unknown` node found.

The bottom-up strategy is explained by algorithm 2, where the function `bottom-up` returns either a tree rooted by the chosen node or `notfound`. The function `bottom-up` returns `notfound` when it cannot conclude, that is it cannot choose a node because the user does not want to be questioned on some nodes (`dontask` nodes). Note that when `notfound` is returned the proof tree does not contain `unknown` nodes.

Example 4.2 Bottom-Up Strategy

Figure 5 shows an example. The numbers shows the order of the node visit.

The search start from the left-most leaf of the proof tree:

node 1 is `dontask`, next node is the 2nd leaf (node 2);

node 2 is `expected`, next node is the 3rd leaf (node 3);

node 3 is `dontask`, next node is the parent of node 3 (node 4);

node 4 is `unexpected`, next node is the 4th leaf (node 5);

node 5 is `expected`, node 6 have no other child, next node is its parent (node 6);

node 6 is `unexpected`, next node is the 5th leaf (node 7);

node 7 is `expected`, next node is the 6th leaf (node 8);

node 8 is `unknown`, node 8 is the chosen node.

4.3 Divide-and-Query Strategy

```
divide-and-query(Tree)
    SubTree ← compute-weight(Tree)
    if SubTree = notfound then return notfound
    elseif a minimal symptom is possible in SubTree then
        return middle-node(SubTree, (weight(SubTree) + 1)/2)
    return notfound
```

Algorithm 3: Divide-and-Query Strategy

The divide-and-query strategy choose the best node to divide the search space:

```
compute-weight(Tree)
    Forest ← children(Tree)
    SubTree ← notfound
    while Forest is not empty
        let Tree1 be the first tree of Forest
        remove Tree1 from Forest
        SubTree1 ← compute-weight(Tree1)
        if SubTree1 ≠ notfound then
            if SubTree = notfound then SubTree ← SubTree1
            elseif weight(SubTree1) < weight(SubTree) then
                SubTree ← SubTree1
    if the root of Tree is unknown or dontask then
        weight(Tree) ← 1 + Σ_{T∈children(Tree)} weight(T)
        return SubTree
    elseif the root of Tree is expected then
        weight(Tree) ← 0
        return SubTree
    /* the root of Tree is unexpected */
    weight(Tree) ← Σ_{T∈children(Tree)} weight(T)
    if SubTree ≠ notfound then
        if weight(SubTree) < weight(Tree) then
            return SubTree
    return Tree
```

Algorithm 4: Divide-and-Query Compute Weight

- when a node is `expected` the subtree rooted by this node can be removed from the search;
- when a node is `unexpected` each node which is not a descendant of this node can be removed from the search.

The point is to choose a node such that there is as much `unknown` nodes in its subtree as out of its subtree (considering that all subtrees rooted by an `expected` node are removed).

```
middle-node(Tree, MiddleWeight)
    Forest ← children(Tree)
    SubTree ← notfound
    while Forest is not empty
        let Tree1 be the first tree of Forest
        remove Tree1 from Forest
        SubTree1 ← middle-node(Tree1, MiddleWeight)
        if SubTree1 ≠ notfound then
            if SubTree = notfound then SubTree ← SubTree1
            elseif weight(SubTree1) < weight(SubTree) then
                SubTree ← SubTree1
    if the root of Tree is unknown then
        weight(Tree) ← abs(weight(Tree) − MiddleWeight)
        if SubTree ≠ notfound then
            if weight(SubTree) < weight(Tree) then
                return SubTree
        return Tree
    return SubTree
```

Algorithm 5: Divide-and-Query Middle Node

The tree considered is a tree rooted by an `unexpected` node with the minimal number of `unknown` and `dontask` nodes.

Algorithm 4 shows the function `compute-weight(Tree)` which associates to each node of the `Tree` the number of `unknown` and `dontask` nodes in its subtree (removing the subtrees rooted by an `expected` node). This information, called `weight(Tree)` is stored in the flag field of the nodes.

Moreover, the function `compute-weight(Tree)` returns the subtree of `Tree` with the minimal number of `unknown` and `dontask` nodes (or `notfound` as usual).

Next, if a minimal symptom is possible in the subtree returned by the function `compute-weight(Tree)`, the function then `middle-node(Tree, MiddleWeight)` (algorithm 5) returns the subtree rooted by the `unknown` node whose weight is nearest the weight of the root divided by two.

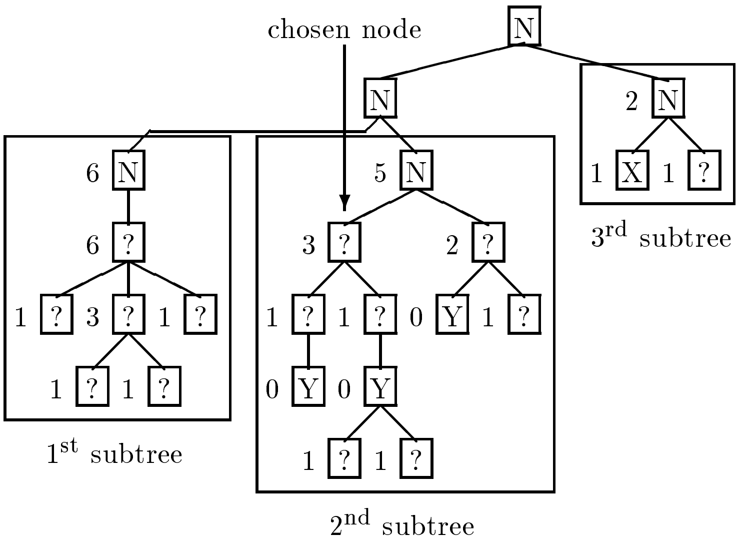

Figure 6: Divide-and-Query Strategy

Algorithm 3 shows the general algorithm.

Example 4.3 Divide-and-Query Strategy

Figure 6 shows an example of divide-and-query strategy.

The number associated to each node is the weight associated to the node by `compute-weight(Tree)` (number of `unknown` and `dontask` node in the subtree).

The search is made in the second subtree because the first subtree has weight of 6 (6 `unknown` and `dontask` nodes) and no minimal symptom is possible in the third subtree (because of the `dontask` node) while a minimal symptom is possible in the second subtree and its weight is 5 (note that the subtrees rooted by an `expected` node are removed: their weights are 0).

The chosen node is the node with the weight of 3. Indeed, there is 2 `unknown` nodes in the subtree (weight of the node $-$ 1) and there is 2 `unknown` nodes outside the subtree (weight of the second subtree $-$ weight of the node): (weight of the second subtree $-$ 1)/2 = weight of the chosen node $-$ 1. We substract 1 because the chosen node will be known as expected or unexpected after.

The Algorithm proposed here for the Divide-and-Query Strategy is different of the traditional algorithm (for example in [7]) for two reasons:

1. The algorithm proposed in [7] is wrong: it does not have the minimal average number of queries (it does not correspond to the complexity annonced).
2. The diagnoser allows the user to change the strategy during a diagnosis, some nodes are known as expected or unexpected wrt the assertions, ... Then it will be incoherent to ignore a part of the information stored in the proof tree. Our algorithm, a proof tree beeing given, really chose the best node wrt the Divide-and-Query strategy.

4.4 Nearby-Error Strategy

```
nearby-error(Tree)
    compute-flags(Tree, n)
    return minimal-node(Tree)
```

Algorithm 6: Nearby-Error Strategy

The nearby-error strategy searches a node such that the number of queries to determine if the node or its parent is a minimal symptom is the lowest.

The number of queries to determine if a node is a minimal symptom is

- either `n` when the node cannot be a minimal symptom;
- or the number of unknown nodes among the node and its children.

The nearby-error-strategy is given by algorithm 6. As usual it returns the subtree rooted by the chosen node or `notfound`. When it returns notfound no minimal symptom is possible in the `Tree` because of the `dontask` nodes.

```
compute-flags(Tree, ParentFlag)
  if all the children of Tree are expected or unknown then
    if the root of Tree is unknown then
      Flag ← number of unknown children of Tree + 1
    elseif the root of Tree is unexpected then
      Flag ← number of unknown children of Tree
    else Flag ← n
  if ParentFlag = n then flag(Tree) ← [Flag, n]
  elseif Flag = n then flag(Tree) ← [ParentFlag, n]
  elseif ParentFlag < Flag then
    flag(Tree) ← [ParentFlag, Flag]
  else flag(Tree) ← [Flag, ParentFlag]
  Forest ← children(Tree)
  while Forest is not empty
    let Tree1 be the first tree of Forest
    remove Tree1 from Forest
    compute-flags(Tree1, Flag)
```

Algorithm 7: Nearby-Error Compute Flags

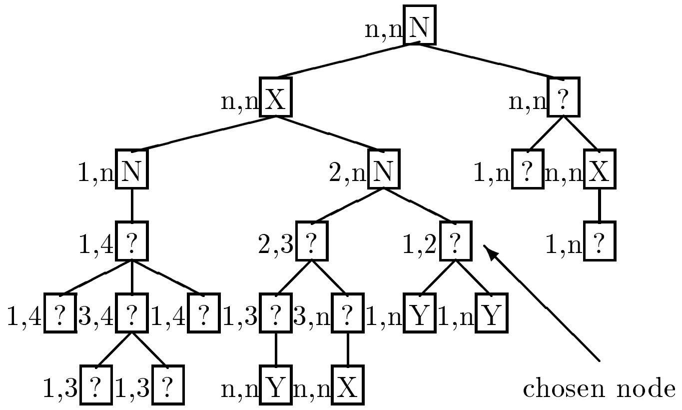

Figure 7: Nearby-Error Strategy

First, it computes the flag of each node of the `Tree` (algorithm 7). The flag of a node is a pair `[F1, F2]`, where `F1` < `F2` (we assume that `n` is greater than each integer). One member of the pair is the number of queries to determine

```
minimal-node(Tree)
    Forest ← children(Tree)
    SubTree ← notfound
    while Forest is not empty
        let Tree1 be the first tree of Forest
        remove Tree1 from Forest
        SubTree1 ← minimal-node(Tree1)
        if SubTree1 ≠ notfound then
            if SubTree = notfound then SubTree ← SubTree1
            elseif inf(flag(SubTree1), flag(SubTree)) then
                SubTree ← SubTree1
    if the root of Tree is unknown then
        if SubTree = notfound then SubTree ← Tree
        elseif inf(flag(Tree), flag(SubTree)) then
            SubTree ← Tree
    return SubTree
```

Algorithm 8: Nearby-Error Minimal Node

if the node is a minimal symptom, the other member is the number of queries to determine if the parent of the node is a minimal symptom.

Next, the function `minimal-node(Tree)` (algorithm 8) returns the chosen node (or `notfound`). The chosen node is a `unknown` node with the minimal flag `[F1, F2]` with respect to the lexicographical order (`n` is greater than each integer).

Example 4.4 Nearby-Error Strategy

Figure 7 shows an example of nearby-error strategy.

The pairs associated to the nodes in the figure are the flags computed by the function `compute-flags(Tree, ParentFlag)`.

Consider the chosen node. Its flag is [1,2]. That is, if the node is `unexpected` you find a minimal symptom (first member of its flag); if the node is `expected` you have to one other node to determine if its parent is a minimal symptom. So in the first case 1 query, in the second case 2 queries.

The nearby-error strategy could be used, for example, when no other strategy conclude or when there is few `unknown` nodes in the proof tree.

4.5 User-Guided Strategy

User-Guided strategy is just a `tk` function which allows the user to graphically chose a node on the proof tree with the mouse.

5 Conclusion

This report explains a part of how declarative error diagnosis is implemented in Calypso.

Some parts of the diagnoser are not implemented yet, like the full query simplification (store simplification), the use of assertions (entailment) and a part of the negative proof tree construction. So they are not detailled in the report.

A full declarative diagnoser will be available on calypso at the end of the project. Then a report will explain how to implement it on others CLP plate-formes: a wide part of the diagnoser is written in standard prolog, the interface is written in `tk`, the other part (mainly the SLD tree construction) is written in `c` and `c++`.

References

[1] Gérard Ferrand and Alexandre Tessier. Clarification of the bases of Declarative Diagnosers for CLP. Deliverable D.WP2.1.M1.1-1, 1997. Debugging Systems for Constraint Programming (ESPRIT 22532).

[2] Gerard Ferrand and Alexandre Tessier. Correction et complétude des sémantiques PLC revisitée par (co-)induction. In Olivier Ridoux, editor, *Journées Francophones de Programmation Logique et Programmation par Contraintes*, pages 19–38. HERMES, 1998.

[3] Abdelkhalek Goumairi. Simplification de contraintes sur les domaines finis. Technical Report , Mémoire de DEA, LIFO, University of Orléans, 1998.

[4] Abdelkhalek Goumairi and Alexandre Tessier. Simplifiaction of Finite Domain Constraints. Deliverable D.WP3.4.M2.1-3, 1998. Debugging Systems for Constraint Programming (ESPRIT 22532).

[5] Bernard Malfon and Alexandre Tessier. An Adaptation of Negative Declarative Error Diagnosis to any Computation Rule. Deliverable D.WP2.1.M2.1-1, 1998. Debugging Systems for Constraint Programming (ESPRIT 22532).

[6] Germán Puebla, Francisco Bueno, and Manuel Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. Deliverable D.WP1.2.M1.1-1, 1997. Debugging Systems for Constraint Programming (ESPRIT 22532).

[7] Ehud Y. Shapiro. *Algorithmic Program Debugging.* ACM Distinguished Dissertation. MIT Press, 1982.

[8] Alexandre Tessier. *Approche, en termes de squelettes de preuve, de la sémantique et du diagnostic d'erreur des programmes logiques avec contraintes.* PhD thesis, LIFO, University of Orléans, 1996.

This research was supported in part by LOCO Project, common to University of Orléans and INRIA Rocquencourt.