Automatic Algorithm Synthesis Applied to Arc Consistency : A Case Study

Jean-Luc Dormoy
Electricité De France R&D Center
1, avenue du général De Gaulle
92141 Clamart Cedex - FRANCE
Jean-Luc.Dormoy@der.edf.fr
Tel: 33 1 47 65 59 06
Fax: 33 1 47 65 54 28

Content Areas : Automatic Programming, Constraint Satisfaction.

Automatic Algorithm Synthesis Applied to Arc Consistency : A Case Study

Abstract : We are currently designing and implementing a knowledge-based automatic program generator, called Descartes. We show in this paper how Descartes applies to a specification of arc consistency and manages to "discover" an optimal worst-case algorithm for this problem.
Finding and improving algorithms for arc consistency in CSPs has been the topic of active research for twenty years. After several attempts, an optimal worst-case algorithm, AC4, was published in 1986. Since then, several improvements have been performed, in particular AC6 last year, which improves the average-case complexity. We show how an optimal worst-case algorithm, though not as elegant as, but "trickier" than AC4, can be generated. Nevertheless, many difficulties remain, which we discuss. In particular, AC6, which is better in average-case complexity than AC4, remains out of reach.

1 Introduction

Arc consistency in CSPs was specified by Waltz in 1972. After several proposals, in particular AC3 in 1985 [Macworth & Freuder 85], Mohr & Henderson [1986] published an optimal algorithm for solving this problem. This algorithm was improved in the average case by Bessières and Cordier last year [1993].

Besides, automatic program generation using AI methods has been the topic of active research for more than ten years. We are currently developing a knowledge-based system for automatic program generation, which integrates various techniques such as formal differentiation, theorem-proving and simplification, and data organization. So, it was tantalizing to apply Descartes to AC. The result is amazing: an optimal worst-case algorithm can be generated.

After showing how arc concistency is specified in Descartes and giving an outline of the generation process in Section 2, we describe in greater details the main steps of the algorithm generation in Sections 3 to 7. We conclude in Section 8 by a discussion, and by suggesting some potential future research directions.

2 Specification of arc-consistency

2.1 Arc consistency

A CSP is defined by giving a set of variables V taking their values in a set of objects A, a set of arcs K[1] between variables, and a set C linking couples of variables $(v,w) \in K$ to sets of couples of objects (a,b). A solution of a CSP is a binding of the variables of V such that all the constraints are satisfied.

It is well-known that solving a CSP is NP-complete, so several polynomial constraint propagation methods have been proposed to make a given CSP simpler, among them arc consistency. Arc consistency consists of removing from the domain of a variable

[1] We limit our study to binary CSPs.

v any object a such that, for another variable w linked to v by K, there is no possible b such that (a,b) belongs to C(v,w).

2.2 Specifying AC in Descartes

The Descartes language is based on functions and sets. It has been kept as close as possible to mathematical statements.

To specify AC in Descartes, we need to be somewhat more formal. We shall call Var the set of variables, Obj the set of objects, Dom = Set(Var x Obj), where x is the cartesian product and Set is the set of subsets, Arc = Set(Var x Var), and Co = Set(Var x Var x Obj x Obj). Var, Obj, Dom, Arc and Co are used as types. If $D \in$ Dom represents the current domains, $(v,a) \in D$ means that a (still) belongs to the domain of v. If $C \in$ Co and $K \in$ Arc, $(v,w) \in K$ and $(v,w,a,b) \in C$ have a similar interpretation.

Arc consistency is defined in Descartes by the following functions :

ArcCo : Co --> Dom
C --> AC(V,D,K,C)
With $V = \{v \mid \exists\ w, a, b \quad (v,w,a,b) \in C \quad \text{or} \quad (w,v,a,b) \in C\}$
With $D = \{(v,a) \mid \exists\ w, b \quad (v,w,a,b) \in C\}$
With $K = \{(v,w) \mid \exists\ a, b \quad (v,w,a,b) \in C\}$

AC : Set(Var) x Dom x Arc x Co --> Dom
(V,D,K,C) --> If $\Theta D = \emptyset$ then D
Else AC(V,D - ΘD,K,C)
With $\Theta D = \{(v,a) \in D \mid \exists\ w \quad (v,w) \in K \quad \text{and}$
$\forall\ b \quad (w,b) \in D \Rightarrow (v,w,a,b) \notin C\}$

The "With" statement is a syntactic facility which avoids repeatedly using complex expressions. ArcCo is the function to be computed. It is an intermediate to specify the links between K, C, V and D. ΘD is the subset of the domains which has to be removed. If nothing has to be removed (Ø is the empty set), then domains D are already arc consistent, otherwise, AC should be re-applied to the new domains ("-" is the set difference). We cannot think of a simpler formal expression of arc consistency (except syntactic details). In particular, it should be clear that no algorithm, let alone an optimal one, is explicitely or implicitely contained in this specification.

The Descartes system starts from this specification and acts with no human intervention.

Remark: In the following, we shall focus on function AC and use a simpler specification, which does not mention the set of arcs K. This changes nothing to the results, but makes the expressions more readable. Anyway, this trick has been used

in the past by people studying arc consistency: one considers that, if $(v,w) \notin K$, then C(v,w) is Obj x Obj.

2.3 Outline of generation

For generating an as efficient as possible program, Descartes uses several techniques and knowledge sources. The first one is formal differentiation. Formal differentiation consists of transforming a "static" program into an incremental one when it must be called several times in similar contexts. It is the case here of the "ΘD" expression, which must be evaluated at each recursive call of AC.

More precisely, a new parameter T is added to function AC, while the differential expression Θ'D of ΘD with respect to D (which is the "recursively moving set") is computed. T represents the current value of ΘD. The new value of T is computed by means of Θ'D. So, AC is transformed into a new function AC' looking like (types and V and C parameters have been omitted):

AC' : (D,T) --> If T = Ø then D
Else AC(D - T,f(T,Θ'D))

where f(T,Θ'D) is the proper combination of T and Θ'D, the result of which is the "new" T. We shall see that this expression looks like:

$\{(v,a) \mid (v,a) \in D$ and $[\exists w \ (v,w) \in K$ and
$(\exists b \ (w,b) \in T$ and
$(v,w,a,b) \in C)$ and
$\neg(\exists b' \ (w,b') \in D$ and
$(v,w,a,b') \in C)]\}$

The reader familiar with arc consistency can see that this set is exactly the set of domain instances which should be removed "because of" T.

The second step consists of symplifying the expression f(T,Θ'D), which is a first-order formula. This simplification requires the proof of some theorems involving the sets D, T, Θ'D, etc. We shall briefly present the weak theorem-proving strategy we have implemented.

The third step consists of selecting the proper way to represent data, in order to reach a good complexity. In particular, "tuples" of sets K and C can be represented by using *indices*. For example, C will not be represented as a list of tuples, but as the index (w,b) --> $\{(v,a) \mid (v,w,a,b) \in C\}$. These indices can be "read" in expression f(T,Θ'D) by using proper knowledge. Previous index is well-known to the readers familiar with AC4: it is the *support sets*.

The fourth step consists of generating a high-level algorithm. The main task is to transform recursion and set operations into loops, and to properly order the statements of the algorithm.

The fifth step consists of selecting concrete data structures (i.e. lists, pointers, arrays, …) for representing the sets and indices mentioned in the algorithm. This selection

depends on the operations they are involved in. Finally, a low-level algorithm, close to a C program, is generated. This low-level algorithm is then translated into a common language (e.g. C) by a suitable compiler.

3 Formal differentiation

3.1 Set-differentiation

Formal differentiation is a well-known technique in program transformation. The underlying idea is to avoid computing an expression $f(x)$ when $f(x_0)$ has already been computed, and when x is "close" to x_0. To do this, a differential df of function f is formally computed, which must satisfy the formula $f(x) = f(x_0)$ op $df(x_0).dx$, where op is a suitable operator and dx is the "difference" between x and x_0. Let's notice that df is a function which, for any x_0, yields a function $df(x_0)$. If the expression $f(x_0)$ op $df(x_0).dx$ is judged as being simpler than $f(x)$, given that $f(x_0)$ is already known, then it should be substituted.

This kind of formal processing is possible in particular with set-functions, i.e. functions which transform sets into sets. To do this, one considers sets E, F, G, …, and positive and negative variations d^+E, d^-E, …, of these sets. When set E "moves" from E_1 to E_2, then $d^+E = E_2 - E_1$ is the set of elements added to E, and $d^-E = E_1 - E_2$ is the set of elements removed from E. Differentials of the usual set operations are defined by formulae, some of which are:

$d^+(E \leftrightarrow F) = (d^+E \leftrightarrow F) \approx (d^+F \leftrightarrow E) \approx (d^+E \leftrightarrow d^+F)$

$d^-(E \leftrightarrow F) = (d^-E \leftrightarrow F) \approx (d^-F \leftrightarrow E)$

$d^+(\pi(E)) = \pi(d^+E) - \pi(E)$

$d^-(\pi(E)) = \pi(d^-E) - [\pi(E - d^-E) \approx \pi(d^+E)]$

$d(\#E) = \#d^+E - \#d^-E$

π is a projection along a given axis, and # is the cardinal.

Now, it is well-known, in particular to database practitionners, that set formulae (when including cartesian product and projection) have the same expressive power than first-order formulae, and algorithms exist, which transform these formulae into each other. So, rules of set-derivation have a counterpart for first-order expressions, which we shall not give here. In the following, we freely use any of these two forms of set- or predicate-derivation.

3.2 Set differentiation applied to arc consistency

Descartes knows the rules of set differentiation, and attempts to use them whenever a first-order formula has to be computed several times in the same context. More precisely, Descartes focuses on first-order expressions nested in a recursion.

AC is obviously recursive. It is also obvious that the expression $\Theta D = \{(v,a) \in D \mid \exists w \in V \ \forall b \ (w,b) \in D \Rightarrow (v,w,a,b) \notin C\}$ has to be computed for each recursive call. The only parameter which changes between two computations is the set D. So, Descartes decides to differentiate the expression ΘD with respect to D. It knows, as the recursion is called with $D - \Theta D$ in place of D, that $d^+D = \emptyset$ in this context. After having normalized ΘD into an internal canonical form, it gets the differentials :

$d^+\Theta D = \{(v,a) \mid (v,a) \in D \text{ and } [\exists\, w \quad (\exists\, b \ (w,b) \in d^-D \text{ and } (v,w,a,b) \in C) \text{ and } \neg(\exists\, b' \ (w,b') \in D \text{ and } (v,w,a,b') \in C \text{ and } (w,b') \notin d^-D)] \text{ and } (v,a) \notin \Theta D\}$

$d^-\Theta D = \{(v,a) \mid (v,a) \in d^-D \text{ and } (v,a) \in \Theta D\}$

4 Using the differential

Once the differential is computed, it is used to transform the specification into a new one. Basically, this relies on the following theorem.

Theorem :

Let $f(D) =$ If $a(D) = \emptyset$ then $b(D)$
Else $f(D \,\Delta\, \Theta(D))$

and

$f'(D,A,T) =$ If $A = \emptyset$ then $b(D)$
Else $f'(D \,\Delta\, T, A \,\Delta\, da(D).T, T \,\Delta\, d\Theta(D).T)$

Then, for all D, $f(D) = f'(D,a(D),\Theta(D))$

Δ is the set symmetric difference. In f', the new parameters A and T respectively memorize the current values of a(D) and Θ(D). Functions a and Θ are only initially computed, and A and T are maintained by computing the differentials da(D) and $d\Theta$(D).

Indeed, another form is used in Descartes. In f', the change of D into $D \,\Delta\, T$ is performed only after f' has been called, while it could be done at once. So, Descartes uses rules which correspond to the theorem (here Θ and a are equal):

Theorem :

Let $f(D) =$ If $\Theta(D) = \emptyset$ then $b(D)$
Else $f(D \,\Delta\, \Theta(D))$

and

$f'(D,A,T) =$ If $T = \emptyset$ then $b(D)$
Else $f'(D \,\Delta\, (T \,\Delta\, d\Theta(D \approx T).T), T \,\Delta\, d\Theta(D \approx T).T)$

Then, for all D, $f(D) = f'(D \,\Delta\, \Theta(D), \Theta(D))$

Moreover, Descartes knows that, in this transformation, d^-D, T and Θ(D) can be identified. So, the expression of the differential becomes

$d^+\Theta D = \{(v,a) \mid (v,a) \in D \approx T \text{ and } [\exists\, w \quad (\exists\, b \ (w,b) \in T \text{ and } (v,w,a,b) \in C) \text{ and } \neg(\exists\, b' \ (w,b') \in D \approx T \text{ and } (v,w,a,b') \in C \text{ and}$

$(w,b') \notin T)]$

and $(v,a) \notin T\}$

$d^{-}\Theta D = \{(v,a) \mid (v,a) \in T$ and $(v,a) \in T\}$

Moreover, these expressions can be simplified (cf. Section 5), and finally, the use of formal differentiation gives rise to the new function AC_2:

$AC_2 : (D,T) \rightarrow$ If $T = \emptyset$ then D

Else $AC_2(D - d^{+}\Theta D, d^{+}\Theta D)$

With $d^{+}\Theta D = \{(v,a) \mid (v,a) \in D$ and $[\exists w$

$(\exists b \quad (w,b) \in T$ and

$(v,w,a,b) \in C)$ and

$\neg(\exists b' \quad (w,b') \in D$ and

$(v,w,a,b') \in C)]\}$

5 Theorem-proving and simplification of expressions

Descartes possesses knowledge for proving theorems about the specification. They make it possible to simplify expressions (this section) and to perform good choices in the algorithm implementation and in the selection of indices (Section 6). We shall make here a simple description of this knowledge. A more complete account of this work can be found in [D 94].

Firstly, Descartes contains more than 300 rewriting rules. Secondly, Descartes has some means of proving local theorems on the specification, which in turn make it possible to transform the specification by taking into account the environment of an expression.

This knowledge is divided in three main parts. First, there are some rules which deduce local theorems. They implement what we call *logical and set constraint propagation* (LSCP) rules. Second, another set of rules performs transformations. They are context-dependant rewriting (CDR) rules, but mention in their condition local theorems. Third, some theorem management (TM) rules combine local theorems to deduce new ones, or suppress unuseful ones. The main computational property of these rules is that they cannot deduce more than a polynomial number of theorems (as a function of the specification size).

Local theorems essentially state that, if e is a boolean expression involved in the specification, then $e \Rightarrow p$ or $p \Rightarrow e$ is true, with p being a simpler expression. If e is a set expression, local theorems have the form $e \subset p$ or $p \subset e$.

These rules make it possible to perform the simplifications leading to AC_2, and local theorems will again intervene in Section 6.

6 Algorithm generation and selection of data indices

The algorithms AC4 and AC6 begin to appear behind the new form of the specification, and in particular in the expression $d^{+}\Theta D$. At this step, Descartes

decides to remove recursion and to transform the first-order expressions occurring in the specification into an algorithm. The basic idea is to transform each dumb variable (i.e. x in $\{x \mid C(x)\}$ or $\exists x$) into a loop.

We have here tail-recursion, which is easy to transform. Let's notice that Descartes possesses knowledge based on clichés for removing complex patterns of recursion (more precisely, for transforming them into tail-recursion).

So, the main problem here consists of precisely determining the domains which the loop variables stemming from a dumb variable must go through, and to decide how the loops must be nested.

6.1 Indices on " " expression

A first-order expression is a sequence of intertwined "$\forall$" ("$\neg\exists$" in the normalized form) and "$\exists$" quantifiers. Each time a "$\forall$" appears in the scope of a "$\exists$" expression, it is separated out and has to be computed alone.

For any "$\forall$" expression, a new index is built. In general, a "$\forall$" expression can be put into the form $\forall\ x \quad P(x,y) \Rightarrow Q(x,y)$, where P and Q are positive in x and y represent the variables quantified over x. A condition is said positive for a variable if it determines a finite set to which this variable must belong. We shall just say that, in this general case, an index $P_1(y) = \{x \mid P(x,y)\}$ is built.

We have in the AC case a particular case: the expression Q is "empty". In this particular case, the expression is equivalent to $\{x \mid P(x,y)\} = \emptyset$. Then, Descartes decides to build the index $NP_2(y) = \#\{x \mid P(x,y)\}$, and to substitute the condition $NP_2(y) = 0$ for $\neg\ \exists\ x \quad P(x,y)$. This new index becomes a new parameter of function AC_2. Moreover, formal differentiation applies to this new parameter. This means that the new index is "maintained" instead of being recomputed at each recursive step.

In the AC case, we have the index $N(v,w,a) = \#\{b' \mid (w,b') \in D \text{ and } (v,w,a,b') \in C\}$. N is defined on the C-projection $\{(v,w,a) \mid \exists\ b''\ (v,w,a,b'') \in C\}$. Its differential is $dN(v,w,a) = -\ \#\{b' \mid (w,b') \in d^{-}D \text{ and } (v,w,a,b') \in C\}$. Eventually, we get the new expression AC_3

$AC_3 : (D,T,N) \dashrightarrow$ If $T = \emptyset$ then D
Else $AC_3(D - d^{+}\Theta D, d^{+}\Theta D, N + dN)$
With $d^{+}\Theta D = \{(v,a) \mid (v,a) \in D$ and $[\exists\ w$
$(\exists\ b \quad (w,b) \in T$ and
$(v,w,a,b) \in C)$ and
$N(v,w,a) = 0]\}$
With $dN(v,w,a) = -\ \#\{b' \mid (w,b') \in T \text{ and } (v,w,a,b') \in C\}$

6.2 Loop nesting

After the transformations described in 6.1, Descartes has to nest loops corresponding to $x_1,\ldots x_n$ in expression $\exists\ x_1,\ldots x_n$.

First, Descartes uses again the local theorems it has deduced to determine the domains of the dumb variables. Let's take for example variables v and a in $d^+\Theta D$. Descartes knows that these variables must verify the *positive* conditions "$(v,a) \in D$" and "$(v,v_0,a,a_0) \in C$".

Then, a set of preferences is computed. The variable domains are compared by various means. Suppose for example that we know that variables x and y must respectively belong to domains A and B, and that local theorems include $A \subset B$. Then, Descartes will prefer to nest the y-loop into the x-loop, according to the well-known heuristic that more constrained variables should be considered first. Another set of preferences stem from formal differentiation. Formal differentiation of a set D is "reasonable" when its variation dD is "small" when compared to D. This reasoning is "induced" in Descartes to become "choose first variables representing variations of sets". So, in the AC case, variables having T as their domains should be preferred over variables having D as their domains. This means that the w-loop should include the v- and a-loops.

Finally, after having chosen according to the preferences, Descartes nests the loops in the order (w,b) --> (v,a) to give rise to the proto-algorithm for evaluating $d^+\Theta D$

```
For all (w,b) ∈ T do
    For all (v,a) such that (v,a) ∈ D and (v,w,a,b) ∈ C do
        …
```

6.3 Domain ordering

Now, when a loop variable belongs to the intersection of several domains, these domains must be ordered. A single domain must be kept as the loop domain, and the other domains will simply be checked in an "if…" statement.

A similar reasoning leading to preferences is used. Here, condition $(v,w,a,b) \in C$ is preferred over $(v,a) \in D$ for a rather complex heuristic reason. Descartes knows (local theorem) that D is included into the C-projection $\{(v,a) \mid \exists\ w\ \exists\ b \quad (v,w,a,b) \in C\}$. From this, it assumes that these sets are of the same order of magnitude. So, as the condition $(v,w,a,b) \in C$ is a "cut" of this projection, it is probably smaller than D.

The evaluation of expression $d^+\Theta D$ becomes

```
For all (w,b) ∈ T do
    For all (v,a) such that (v,w,a,b) ∈ C do
        If (v,a) ∈ D then
        …
```

6.4 Selection of indices

Now, indices can be clearly "read". The domain condition of the (v,a) loop would be simpler if we had at hand for each (w,b) the sets $C_1(w,b) = \{(v,a) \mid (v,w,a,b) \in C\}$. These sets are the so-called supports in AC4. We now have

```
For all (w,b) ∈ T do
    For all (v,a) ∈ C1(w,b) do
        If (v,a) ∈ D then
        …
```

6.5 High-level algorithm for AC

After a few transformations that we do not describe, the high-level algorithm generated is:

```
While T ? Ø do
    Θ'DT ← Ø
    For all (w,b) ∈ T do
        For all (v,a) ∈ C1(w,b) do
            If N(v,w,a) = 0 then
                If (v,a) ∈ D then
                    Θ'DT ← Θ'DT + (v,a)
    D ← D - Θ'DT
    T ← Θ'DT
    For all (w,b'') ∈ Θ'DT do
        For all (v,a) ∈ C1(w,b'') do
            N(v,w,a) ← N(v,w,a) - 1
```

7 From high-level to low-level algorithm

The high-level algorithm still mentions sets. There is no uniform way to efficiently represent sets on a computer, but a collection of possibilities. On one hand, a set can be involved in several operations. It can be initialized, elements can be added and/or removed, it can be involved in set operations such as union, one can search elements in it, … Moreover, one can know things about a set: it can be included in another set, for example. On the other hand, a set can be represented by a list, or a matrix, or by a more complex structure such as a hash table, a binary or a B-tree, etc.

Descartes possesses knowledge which links the operations performed on a set and the various data structures available. We shall not give details about this knowledge, instead we shall show how it is used in the AC case.

The high-level algorithm mentions five variables: T, Θ'DT, D, N and C_1. Let's first consider D and T. We know that T is the domain of a loop and assigned by another variable, and that elements are searched in and removed from D. So, the conclusion is that T should be represented as a list, and D as a matrix of booleans. These structures are the fastest for the operations involved, and are sufficient.

Let's now turn to Θ'DT. Θ'DT is the domain of a loop, is initialized by a constant quantity, and elements are added to it. So, a list would be suitable for Θ'DT as a loop domain and for the initialization. But it is not for adding elements, since the occurrence of the elements to be added should be searched prior to the adding. In this case, it is suitable to add a second redundant structure, a matrix, which makes it possible to know in constant time whether an element is member of the set. Now, it is not suitable to initialize a matrix, in particular when the initialization is nested in a loop. Here, Descartes uses a trick. Instead of using an array of booleans for representing the matrix, it uses an array of integers A, and it adds a counter n in the loop. x will be an element of the set iff A[x] = n. This is possible because Θ'DT is initialized to the empty set at the beginning of the loop. Obviously, the matrix and the counter should be reinitialized each time the capacity of integers (more than 4.10^9 on 4-bytes computers) is overflowed, which is a rare event in a pratical way.

Finally, after introducing the right data structures, pointer variables, and so on, the low-level algorithm is generated:

```
n ← 0; LΘ' ← nil;  MΘ' ← 0;
While LT ? nil do
    pLT ← LT; Inc(n,MΘ'); Free(LΘ');  LΘ' ← nil;
    While pLT ? nil do
        (w,b) ← pLT.
        pC ← C1(w,b)
        While pC ? nil do
            (v,a) ← pC.
            If MD(v,a) = 1 then
                If N(v,w,a) = 0 then
                    If MΘ'(v,a) < n then
                        MΘ'(v,a) ← n
                        LΘ' ← cons((v,a),LΘ')
            pC ← pC.next
        pLT ← pLT.next
    pLΘ' ← LΘ'
    While LΘ' ? nil do
        (w,b'') ← pLΘ'.; pC ← C1(w,b'')
        While pC ? nil do
            (v,a) ← pC.
            N(v,w,a) ← N(v,w,a) - 1
            pC ← pC.next
        MD(w,b) ← 0
        Free(T); T ← LΘ';
```

8 Discussion

8.1 What Descartes is and is not

First, it must be pointed out that Descartes is a general system, not devoted to any particular algorithm. We have along the paper tried to show that the transformations

it applies in the AC case have general causes, not particular ones. One could even say that the knowledge elicited in Descartes could also be used by human beings, in particular the rules of formal differentiation. It simply turns out that its knowledge is sufficient for properly applying to AC. We must also confess that AC has been a source of inspiration for us. But Descartes, while still in an experimental shape, has also successfully been applied to different problems.

Second, Descartes possesses sophisticated knowledge for transforming specifications and programs, but its control is very primitive. Indeed, the transformations described here are applied in a fixed order. Another example: the weak theorem-proving capabilities of Descartes have been designed so that they can blindly apply in polynomial time. So, Descartes is not a problem solver having to tackle a large, or infinite search space. Instead, it blindly applies operators to a limited space.

8.2 Comparison with AC4 and AC6

First, the reader can check that previous algorithm is optimal, provided that the number of steps performed in the "While LT ? nil" loop does not overflow the computer integer capacity. The proof published in [Mohr & Henderson 86] can be adapted here.

Nevertheless, one can see that Descartes' algorithm is somewhat trickier and less elegant (indeed, this is a general property of machine-generated algorithms!). The main difference lies in the fact that elements of T, i.e. the elements that must be removed from the domains, are not processed one by one, but all together, while in AC4, as soon as an element that should be removed is found, it is actually removed from the domains. Indeed, this means that our AC_2 function, which has the form

AC_2 : (D,T) --> If T = Ø then D
Else AC_2(D - Θ'(D,T),Θ'(D,T))

is also equal to the function (where Elem(T) selects any element of T)

AC_4 : (D,T) --> If T = Ø then D
Else AC_4(D - Θ'(D,{x}),T - {x} ≈ Θ'(D,{x}))
With x = Elem(T)

This is an occurrence of what could be called "data commutativity ", or "data parallelization".

We could have added to Descartes the proper knowledge to apply this transformation to the particular AC case. However, as already said, Descartes is supposed to be general, not devoted to a particular algorithm. This issue of proving the commutativity of actions performed on data and then reordering them is important, but we have not yet found a proper general framework for tackling it.

However, this difference must not hide that the essential features of AC4 have been found, in particular the representation of C as support sets, and the counters N.

Besides, we have not presented here the initialization of the data structures used by AC (supports and counters). Descartes generates the related program separately, which is similar to AC4 initialization.

Now, Descartes' algorithm is still far away from AC6. There are several reasons for this. First of all, Descartes exhibits a heavy trend to prefer time efficiency against memory. Second, Descartes has no capability of precisely analyzing the constituents of the sets and relations involved in the specification. It considers them as a whole, and cannot see beyond indices for tuples. It turns out that the starting point of AC6 lies on this kind of precise analysis. Third, and this is the main reason, AC6 requires some *invention* from its designer. In particular, arbitrary orders are imposed to sets (in particular the domains D) to avoid the memorizing of the whole support sets and counters. As previously said, Descartes is "stupid" in some sense, since it attempts to apply transformations in a fixed order, and certainly invents nothing. It is limited to a very directed search in a limited space. So, AC6 is currently definitely out of reach.

8.3 Future directions

First, Descartes is still in an experimental stage, and much work remains to make it easily usable on any specification. It is currently implemented as a set of rule bases. We intend to specify it in its own language, and to try to bootstrap it. However, this will be a very difficult task, since the specification of Descartes will be much longer than the AC example presented here. It remains to see if we will "just" have to tackle combinatorial difficulties (i.e., the specification to be processed simply is very large), or conceptual problems (i.e. more sophisticated knowledge is required).

Nevertheless, Descartes is already a complex system, and probably already difficult to qualitatively improve. So, we shall probably have to tackle new problems.

A first problem is control. We have mentionned that Descartes has been designed in order to limit control issues. In particular, Descartes is unable to judge the interestingness of the transformations it applies. For example, it is unable to assess the final algorithm it generates (Descartes does not know that it has generated an optimal worst-case algorithm for AC!). Would this capability exist, we could extend the search space visited by Descartes. A research direction could be [Minton 93].

Another problem is "completeness". We certainly do not hope to design a totally general program generator, but Descartes' capabilities still have to be assessed. In particular, Descartes only possesses clichés on various aspects (recursion removal, simplifications, use of formal differentiation). It turns out that its knowledge comes from a common core made up of, say, naive set theory. So, it would be interesting to design another system which would generate some of Descartes' knowledge, and assess its utility. Examples of this kind of system, within different frameworks, are [Pitrat 66] and [Pastre 89]. Anyway, this is a long-term research direction.

References

[Barstow, 79] *Knowledge-based program construction*. Elsevier North Holland, New-York.

[Bessières & Cordier, 93] *Arc consistency and arc consistency again*. AAAI 93.

[Burstall, Darlington, 76] *A system which automatically improves programs*. Acta Inf. 6. pp 41-60.

[Burstall, Darlington, 77] *A transformation system for developing recursive programs*. J. ACM 24, 1, pp 44-67.

[D, 94] *Program transformation and theorem proving as constraint propagation*. Internal report.

[Ginoux, Lagrange, 89a] *An expert system approach to program synthesis*. AAAI Spring Symposium Series 1989, Artificial Intelligence and Software Engineering, Standford, Ca. USA.

[Ginoux, Lagrange, 89b] *Synthesis of simple programs which handle complex data.*, IJCAI'89 Workshop on Automating Software Design, Detroit, Mich. USA.

[Kant, Barstow, 81] *The refinement paradigm : The interaction of coding and efficiency knowledge in program synthesis*. IEEE Trans. Softw. Eng. 7, 458-471.

[Mackworth & Freuder 85] *The complexity of some polynomial network consistency algorithms for constraint satisfaction problems*. AI J. 25-1, pp. 65-74.

[Minton 93] *Integrating heuristics for constraint satisfaction problems: a case study*. AAAI 93.

[Mohr & Henderson, 86] *Arc and path consistency revisited*. AI J. 28-2, pp. 225-233.

[Mostow, 91] *A transformational approach to knowledge compilation*. In Lowry & McCartney, Eds, *Automating software design*.

[Pastre, 89] *MUSCADET: An automatic theorem proving system using knowledge and metaknowledge in mathematics*. AI J. 38, 3, pp 257-318.

[Pitrat 66] *A theorem-proving program using heuristic methods*. PhD dissertation, Paris 6 University, 1966.

[Smith, 91] *KIDS: A semiautomatic program development system*. In Lowry & McCartney, Eds, *Automating software design*.

[Steier, Anderson, 89] *Algorithm synthesis : A comparative study*. Springer-Verlag. New-York.