ICOT Technical Report: TR-729

TR-729

CAL: A Constraint Logic Programming Language Its Enhancement for Application to Handling Robots

by

A. Aiba, S. Sato, S. Terasaki, M. Sakata
& K. Machida

January, 1992

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191 ~5
Telex ICOT J32964

Institute for New Generation Computer Technology

CAL: A Constraint Logic Programming Language Its Enhancement for Application to Handling Robots

Akira Aiba Shinichi Sato Satoshi Terasaki
Institute for New Generation Computer Technology*

Masahiro Sakata
Software Engineering Development Laboratory, NEC Corporation

Kazuhiro Machida
NEC Scientific Information System Development,Ltd.

Abstract

At ICOT, we have been researching into the Constraint Logic Programming Language CAL (*Contrainte Avec Logique*) since 1987 [1, 7].

CAL was originally designed as a CLP language with a constraint solver for algebraic polynomial equation constraints employing the Buchberger algorithm. As part of our research into CAL applications we have been investigating a design support system for handling robots.

Since the application needs several extra functions besides its original functions of computing Gröbner bases, we have extended the CAL language processor to version 2.1.

This paper describes these additional functions for the application: a facility to approximate the real roots of univariate equations in the algebraic constraint solver employing the Buchberger algorithm, handling multiple solutions for each non-linear univariate equation by introducing the concept of context.

Through this research and development, we are aiming to establish a very high level programming language for problem solving, based on concept of the constraint logic programming.

1 Introduction

A programming paradigm called *Constraint Logic Programming* (CLP) was proposed by A. Colmerauer [3], and J. Jaffar and J-L. Lassez [5] as an extension of logic programming. Jaffar and Lassez showed that CLP is a generalization of logic programming in the sense that it possesses coinciding logical, functional, and operational semantics in the same way as logic programming [11].

At the Institute for New Generation Computer Technology (ICOT) we have been working on a CLP language named CAL (*Contrainte Avec Logique*) since 1987 [1, 7]. One of the significant differences between CAL and other CLP languages is its constraint solver. That is, since CAL employs the Buchberger algorithm as its constraint solving algorithm, it can deal with non-linear polynomial equation constraints. We refer to this constraint solver as the "*algebraic constraint solver*".

*21F, Mita Kokusai Building, 4-18, Mita 1-Chome, Minato-ku, Tokyo 108, Japan

Since then, we have been studying the application of CAL with algebraic constraints to handling robots. Our goal was a design support system for handling robots that calls for the enhancement of CAL.

This paper describes our enhanced CAL, Version 2.1, and the two programs that constitute the central functions of the design support system: kinematics and statics of handling robots.

The advantages of using CAL for handling robot kinematics and statics are as follows. Firstly, kinematics consists of two main areas. These are "direct kinematics" for computing a position and the orientation of the end effector from given arm lengths and joint rotation angles, and "inverse kinematics" for computing desired arm lengths and joint rotation angles when moving the end effector to a given position with a given orientation. By using CAL, both direct kinematics and inverse kinematics can be handlued by a single program. Secondly, this program can deal with any configuration of handling robot. The configuration is reflected into the structure of the query.

Since the algebraic constraint solver computes the Gröbner bases of a given set of constraints, even if those constraints contain sufficient information to determine the values of all variables, a CAL program can return a set of equations, Gröbner base. As the simplest example, let us consider the equation $x^2 = 4$ be the only constraint. In this case, the value of x can be determined to 2 or -2, however, the Gröbner base of it is $x^2 = 4$.

To determine the values of variables, we adopt Strum's theorem to approximate the real roots of uni-variate equations. Since these approximated values contain computational errors, we have to modify the Buchberger algorithm to enable calculation using approximated values. For the real roots of univariate non-linear equations, one variable may have more than two values. For example, for the equation $x^2 = 4$, the values of x are -2, and 2. To deal with such situations, we introduce a concept of context.

The function to approximate real roots is necessary in the handling robot kinematics and statics especially when they are used in the context of the design support system. As presented in the section 4, because of the degree of freedom of handling robot, there will be two or more solutions of problems in kinematics represented by non-linear equations. And since each value gives different torque, thus, they will give different results in the further evaluation.

That is, the following enhanced functions are offered:

- facility to approximate real roots of univariate equations in the algebraic constraint solver,
- modification of Buchberger algorithm to enable handling of approximated values, and
- handling of multiple solutions for each non-linear univariate equation by introducing the concept of contet.

The structure of this paper is as follows. In Section 2, the CAL language is briefly summarized, and the construction of the CAL language processor is described in Section 3. In Section 4, an application of CAL to handling robots is described, and other constraint solvers are sketched in Section 5.

2 CAL language

This section summarizes the syntax of CAL. For a detailed description of CAL syntax, refer to the CAL User's Manual [4].

The syntax of CAL is very similar to that of Prolog, except for its constraints. A CAL program features two types of variables: logical variables denoted by a sequence of alpha-numeric characters starting with a capital letter (same as usual Prolog variables), and constraint variables denoted by a sequence of alpha-numeric characters starting with a lower-case letter. Constraint variables are treated as global variables in CAL, while logical variables are treated as local variables. This distinction has been introduced to enable easy incremental query.

The following is an example CAL program that features algebraic constraints. This program gives the new property of a triangle, the relation of which holds among the lengths of the three edges of the triangle and its surface area, from three known properties of a triangle.

```
:- public triangle/4.

surface_area(H,L,S) :- alg:L*H=2*S.
right(A,B,C) :- alg:A^2+B^2=C^2.
triangle(A,B,C,S) :-
        alg:C=CA+CB,
        right(CA,H,A),
        right(CB,H,B),
        surface_area(H,C,S).
```

The first clause "surface_area" expresses the formula for computing the surface area of a triangle S from its height H and its baseline length L. The second is the Pythagorean theorem for a right-angled triangle. The third asserts that every triangle can be divided into two right-angled triangles. (See Figure-1.)

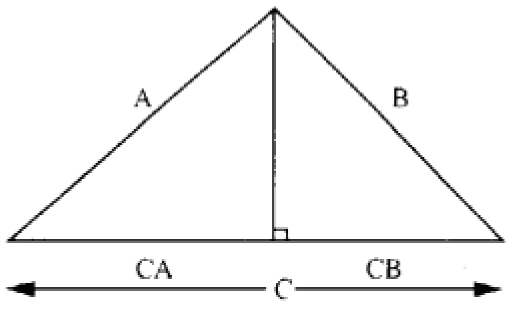


Figure-1: The third clause

In the following query, "heron:" is the name of the CAL source file in which the desired program is defined.

```
?- alg:pre(s,10), heron:triangle(a, b, c, s).
```

This query asks for the general relationship that holds among the lengths of three edges of a triangle and its surface area.

The invocation of "alg:pre(s,10)" defines the precedence of the variable s to be 10. Since the algebraic constraint solver employs Buchberger algorithm, ordering among monomials is essential for its computation. This command changes the precedence of variables. Initially, all variables are assigned a precedence of 0. So, in this case, the precedence of variable s is upped.

To this query, the system responding with the following equation:

```
s^2 = -1/16*b^4+1/8*a^2*b^2-1/16*a^4+1/8*c^2*b^2
      +1/8*c^2*a^2-1/16*c^4.
```

This equation is, actually, a developed form of Heron's formula.

3 CAL system

In this section, we will introduce the overall structure of a CAL system: system configuration, the approximation of real roots of univariate polynomials in an algebraic solver, and context.

3.1 System Configuration

The CAL language processor consists of the following three subsystems: *Translator*, *Inference Engine*, and *Constraint Solvers*. These subsystems are connected as shown in Figure-2.

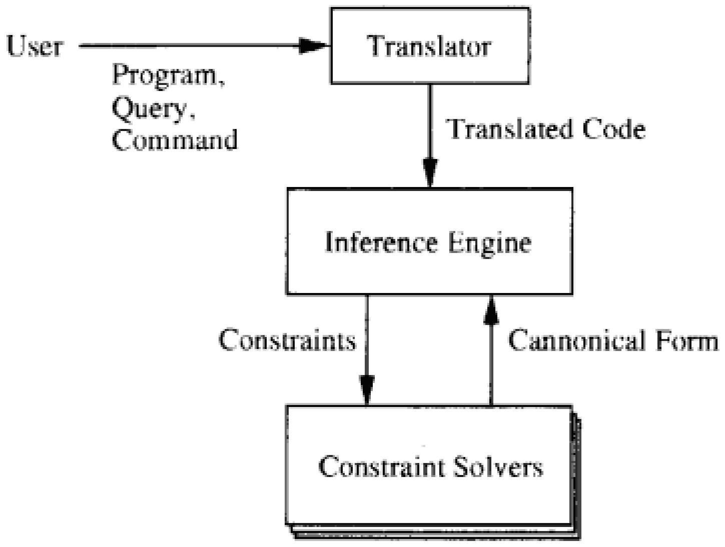


Figure-2: Overall structure of CAL language processor

A CAL system runs on the *PSI* (Personal Sequence Inference) machine, developed in Fifth Generation Computer System project. The language used on the PSI machine is called ESP (Extended Self-contained Prolog) [2], an object-oriented logic programming language.

The translator receives all inputs from a user, then translates them into ESP code. Thus, a CAL source program is translated into the corresponding ESP program. The inference engine in Figure-2 utilizes the ESP language processor itself, and a translated program invokes a constraint solver whenever the language processor finds a constraint during execution.

The constraint solver adds a newly obtained constraint to the set of canonical forms of the former constraints, then returns a new set of canonical forms.

At present, CAL Version 2.1 offers the following five constraint solvers for various domains:

1. Algebraic Solver employing Buchberger Algorithm[1, 7],
2. Boolean Solver employing Boolean Buchberger Algorithm[9, 1, 7],
3. Boolean Solver employing Incremental Boolean Elimination Algorithm[6]
4. Linear Solver employing Simplex method, and
5. Set Solver employing Modified Boolean Buchberger Algorithm[10]

3.2 Approximation of real roots

Since the "Algebraic solver" employs the Buchberger algorithm, the solver does not always return a value for each variable. For example, if we input the following query to the CAL program shown in Section 2,

```
?- heron:triangle(3,4,5,s)
```

then the CAL system calculates the surface area of a triangle whose edges have lengths 3, 4, and 5. The CAL processor, however, returns the following answer since this is the Gröbner base of a given set of equations:

```
s^2 = 36
```

Therefore, we cannot obtain an answer of `s = 6`.

The user, however, will want to know the value of variable `s` in this case.

If a variable has finitely many values in the whole solutions, there is a way of obtaining univariate equation with the variable in the Gröbner base. Therefore, if we can add a functionality that enables us to extract approximate values for a variable from univariate non-linear equations, we can approximate all possible real solutions.

For this purpose, we implemented a means of approximating the real roots of univariate non-linear polynomials, based on Strum's theorem. In CAL, by applying Strum's theorem, all real roots of univariate polynomials are isolated by obtaining a set of intervals, where each interval contain one real root. Then, each isolated real root is approximated by the given precision.

There are several well-known methods of approximating the real roots of univariate non-linear equations, such as the Newton Raphson method, or the bisection method. However, since the real roots computed using these methods are approximated values, the algebraic constraint solver should be modified to be capable of handling of approximated values when these approximate real roots functionalities are combined into the solver.

By combining these functionalities into the solver, the values of variables can be obtained when univariate equations are included into a set of answer constraints by estimating the approximated values for their real roots. Furthermore, these approximated values can also be used to reduce other constraints to produce other univariate equations. Thus, if all variables have finitely many values, then by performing this iteration all combinations of values of all variables can be obtained.

The number of iterations is relatively small with the Newton-Raphson method. But, if the initial value is not appropriate then the iteration may never terminate. On the other hand, in the bisection method, the number of iterations is greater than in the Newton-Raphson method, and detection of multiple roots is required. However, estimation of the initial value is not required. Because of the stability on the selection of the initial value, the bisection method is employed to approximate the real roots of univariate non-linear equations in CAL.

In the computation of the Gröbner base, redundant monomials whose coefficients are zero, and redundant equations whose all monomials are zero are removed. If an approximate value is introduced into the computation of the Gröbner base without changing the algorithm, redundant monomials/equations may not be recognized as being redundant, or non-redundant monomials/equations may be wrongly recognized as being redundant, because of a computational error.

Therefore, the monomial redaundancy check is changed from detecting the zero coefficient to comparing the coefficient with precision Err. That is, if the absolute value of a coefficient is smaller than Err, then the monomial is recognized as being redundant. If the numerical computation of coefficients is carried out using floating point number, then errors such as rounding errors and cut-off errors should be cared. Therefore, in CAL 2.1, the computation of coefficients is carried out using infinite precision rational numbers.

The following illustrates the finding real roots of the equation `s^2 = 36`.

To approximate real roots, the query

```
?- heron:triangle(3,4,5,s).
```

should be replaced with

```
?- alg:set_out_mode(float),
   alg:set_error1(1/1000000),
   alg:set_error2(1/100000000),
   heron:triangle(3,4,5,s),
   alg:get_result(eq,1,nonlin,R),
   alg:find(R,S),
   alg:constr(S).
```

The first line of the above "**alg:set_out_mode**" sets the output mode to "**float**". Without this, approximate values are output as fractions.

The second line of the above "**alg:set_error1**" specifies the precision used to compare coefficients in the computation of the Gröbner base. The third line "**set_error2**" specifies the precision used to approximate real roots by the bisection method. Usually, the value of `set_error2` is much smaller than `set_error1`.

The fourth line "**heron:triangle**" is the invocation of the user defined predicate.

The fifth line "**alg:get_result**" selects appropriate equations from the Gröbner base. In this case, univariate (specified by `1`) non-linear (specified by `nonlin`) equations (specified by `eq`) are selected, and unified to a variable `R`.

`R` is then passed to "**alg:find**" to approximate the real roots of equations in `R`. Such real roots are obtained in the variable `S`.

Then, `S` is again input as constraint to reduce other constraints in the Gröbner base.

For the above query, the CAL system outputs the following answer.

```
R = [s^2 = 36].
S = [s = real(-, [5121, 7921, 1030], [9184, 7986, 171])]
  = [s = -6.000000099].
s = -6.000000099
```

and

```
R = [s^2 = 36].
S = [s = real(+, [5121, 7921, 1030], [9184, 7986, 171])]
  = [s = +6.000000099].
s = 6.000000099
```

Thus, the first solution is `s = -6.000000099`, and the second solution is `s = 6.000000099` with tiny errors.

3.3 Context

3.3.1 Manipulating multiple values

To deal with a situation where a variable may have plural values, as in the above example, we have introduce the concept of "*context*", and "*context tree*".

A context is a constraint set. A new context is created whenever the set of constraints is changed. In CAL 2.1, contexts are manipulated by the "context tree". A "current context" is a node of the context tree, and is the target of the context manipulation.

A set of constraints is changed in the following cases:

1. A goal execution:
   A new context is created as a child-node of the current context in the context tree.

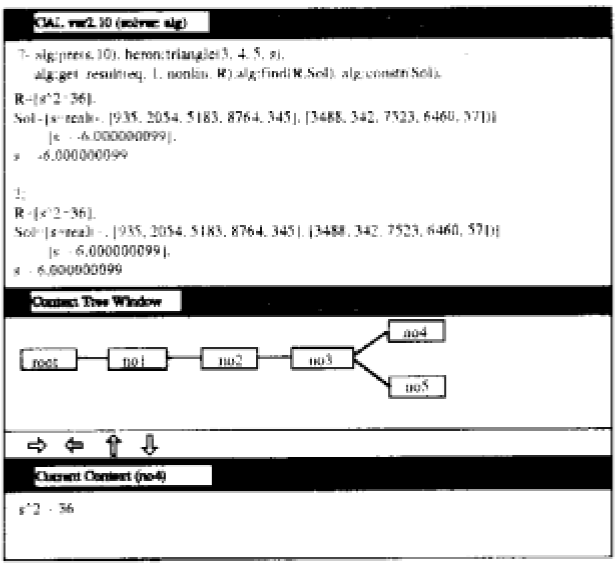
2. Creation of new constraint set by requiring another answers for a goal:
   A new context is created as a sibling node of the current context in the context tree.

3. Changing the precedence:
   A new context is created as a child-node of the current context in the context tree.

In the all case, a new set of constraints is stored in the newly created context. And the newly created context is set as the current context. A root of the context tree is called "*a root context*".

Several commands are provided to manipulate contexts and the context tree: a command to display the contents of a context, a command to set a context as the current context, delete the sub-tree of contexts from the context tree, and others.

The following figure shows the CAL processor windows including the current context window, and the context tree window.



```
?- alg:pres(s,10), heron:triangle(3, 4, 5, s),
   alg:get_result(eq, 1, nonlin, R),alg:find(R,Sol), alg:constr(Sol).
R=[s^2=36],
Sol=[s=real(-, [935, 2054, 5183, 8764, 345], [3488, 342, 7523, 6460, 571])]
      [s = -6.000000099],
s = -6.000000099

;
R=[s^2=36],
Sol=[s=real(+, [935, 2054, 5183, 8764, 345], [3488, 342, 7523, 6460, 571])]
      [s = 6.000000099],
s = 6.000000099
```

```
s^2 = 36
```

Figure-3: CAL system windows

4 Application of CAL to Robotics

In this section, we will describe the application of CAL to handling robots.

A handling robot is a kind of industrial robot that picks up an object with its end-effector (grip) and changes its position and orientation by rotating itsh joints and changing the length of its arms. A typical 3-joint handling robot is shown in the following figure.

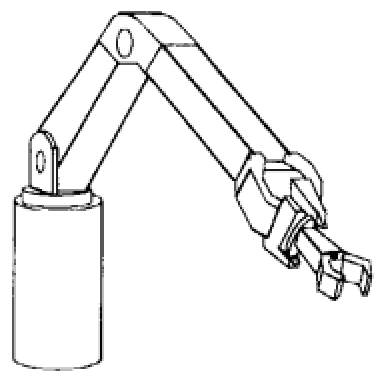


Figure 4: Handling robot

Currently, the design of handling robots relies greatly on human engineer's effort. The design process for the structure of handling robots can be decomposed into the following sequence:

1. Determining the configuration of the handling robot.
2. Deriving the relationship among arm lengths, joint rotation angles, position of the end effector, and its orientation.
3. Obtaining the Jacobian matrix of position parameters and orientation parameters with respect to joint rotation angles.
4. Obtaining singular points where the determinant of the Jacobian matrixis equal to 0.
5. Evaluating the configuration.

In the following, we will consider two central programs in 2 of the above, the "*Kinematics*" and "*Statics*" of handling robots, by using CAL programs.

Kinematics represents the relation among the target position and the target orientation of the end-effector, and the extension of each arm and the rotation angle of each joint. Kinematics consists of two problems: one is the calculation of the position and the orientation of the end effector when the length of each arm and the rotation angle of each joint are given. The second is the calculation of the desired arm lengths and rotation angles to move the end effector to given position with a given orientation. The former is called "*direct kinematics*". The latter is called "*inverse kinematics*".

Statics represents the relation among the torque acting on each joint, the force acting on the end-effector, and the position of each joint that represents the shape of the handling robot.

By applying constraint logic programming to this field of application, we can realize the following advantages:

1. A user writes a program by stating only the mathematical relationships among the position and orientation of the end effector, arm lengths, and rotation angles.

2. Any type of handling robots, having any configuration, can be handled by a single program.
3. A single CAL program can handle both direct kinematics and inverse kinematics.
4. For statics, bi-directional computation, the same as for the kinematics program, can be employed. That is, for example, if the shape and the torque are given then the force working on the end-effector is determined. And if the shape and the force on the end-effector are given then the torque is determined. Furthermore, if the force on the end-effector is given then the relation between the shape and the torque is obtained.

4.1 Kinematics

To formulate kinematics, the following matrix is used. This matrix is widely used in robotics, and is covered in detail in many text books in the field. We call it the *position-orientation matrix*.

The position of an object is expressed by a vector (p_x, p_y, p_z) and its orientation is expressed by two unit vectors (a_x, a_y, a_z) and (b_x, b_y, b_z) perpendicular to each other. These two unit vectors are fixed on the object and their directions are changed by rotation.

Among the components of these vectors, the following relations hold.

$$
\begin{aligned}
{a_x}^2 + {a_y}^2 + {a_z}^2 &= 1 &(1)\\
{b_x}^2 + {b_y}^2 + {b_z}^2 &= 1 &(2)\\
a_x * b_x + a_y * b_y + a_z * b_z &= 0 &(3)
\end{aligned}
$$

To represent position and orientation together, the following position-orientation matrix (abbreviated to *p-o matrix*) $[P^{**}]$ is used.

$$
[P^{**}] = \begin{pmatrix} p_x & a_x & b_x \\ p_y & a_y & b_y \\ p_z & a_z & b_z \\ 1 & 0 & 0 \end{pmatrix}.
$$

Any position and any orientation of the object can be represented by this matrix. Note that there are three redundancies among the components of the orientation vectors because of the above relationship.

The 1 and 0's in the fourth row of the matrix are added to make manipulation of rotation and straight movement easier.

Note that any transformation of the object in three dimensional space can be expressed by a combination of straight movement and rotation. If the object of state $[P^{**}]$ is rotated through angle θ around an axis W $(= (W_x, W_y, W_z))$ through its center, then moved by dP $(= (d_x, d_y, d_z))$, then the new *p-o matrix* for the object will be

$$
\begin{aligned}
[P'^{**}] &= [M][P^{**}] &(4)\\
[M] &= \begin{pmatrix} E & dP \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} e_{11} & e_{12} & e_{13} & d_x \\ e_{21} & e_{22} & e_{23} & d_y \\ e_{31} & e_{32} & e_{33} & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} &(5)
\end{aligned}
$$

The matrix $[M]$ above is called a *transformation matrix*. This represents both straight movement and rotation. A sub-matrix E, called the *rotation matrix*, in the transformation matrix represents rotation. Its elements are determined by the rotation axis W and the rotation angle θ.

Movement of the end effector of a handling robot can also be represented by a combination of straight movement and rotation. The transformation of an object that is held by the end effector is also represented by the transformation matrix $[M]$.

Generally, in a handling robot having m joints and m arms, resulting the *p-o matrix* can be expressed as follows:

$$\begin{aligned}[][P^{**}] &= \begin{pmatrix} E_1 & P_1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} E_2 & P_2 \\ 0 & 1 \end{pmatrix} \cdots \begin{pmatrix} E_i & P_i \\ 0 & 1 \end{pmatrix} \cdots \begin{pmatrix} E_m & P_m \\ 0 & 1 \end{pmatrix} \begin{pmatrix} G_0 & a_0 & b_0 \\ 1 & 0 & 0 \end{pmatrix} \\ &= [M_1][M_2]\ldots[M_i]\ldots[M_m][G^{**}], \end{aligned} \tag{6}$$

where E_i is the rotation matrix of the i-th joint. P_i is the vector $(P_{xi}, P_{y_i}, P_{zi})$ of the i-th arm before rotation. G_0 is the initial vector of the end-effector (G_x, G_y, G_z). a_0, b_0 is the initial orientation of the end-effector. (a_{x0}, a_{y0}, a_{z0}). (b_{x0}, b_{y0}, b_{z0}). and m is the number of joints (rank number).

We let W_i be the initial direction vector of the rotation axis of the i-th joint $(W_{xi}, W_{y_i}, W_{zi})$, and θ_i be the rotation angle of the i-th joint.

Parameters θ_i, $(W_{xi}, W_{y_i}, W_{zi})$, and $(R_{xi}, R_{y_i}, R_{zi})$ are called *transformation matrix* parameters because they determine the contents of each *transformation matrix*.

4.2 Statics

We will now introduce the statics of handling robots. When a force acts on the end effector, torque acts on the axis of each joint. Thus, the motor shaft of each joint has to be rotated against the torque acting on it. The calculation of the torque acting on each joint is an important issue since, if the torque is stronger than the torque generated by the motor at each joint, the joint can not be rotated. The torque acting on each joint can be expressed in terms of the position of each joint, and the position of the end effector. Here, the only force we consider is the weight of the object.

The torque acting on the i-th joint T_i can be represented as follows:

$$T_i = Z_i \cdot (P_m - P_i) \times F_m \tag{7}$$

where,

where Z_i is the direction vector of the rotation axis of the i-th joint, P_m is the vector from the root joint to the end-effector, P_i is the vector from the root joint to the i-th joint, and F_m is the force vector acting on the end-effector.

4.3 Programming in CAL

We developed two CAL programs [8]. One is for kinematics, the other for statics. The kinematics program calculates the value of each *transformation matrix* parameter from the target position and the target orientation parameters by applying the above expression. Then, the statics program calculates the position of each joint and torque acting on it from the values of the *transformation matrix* parameters by applying the above expression.

4.3.1 Kinematics Program

As we described, a kinematics program can handle the direct kinematics and inverse kinematics of any type of handling robot.

In the kinematics program, the head of the top level predicate is as follows.

```
robot(Mlist,Gx,Gy,Gz,Ax0,Ay0,Az0,Bx0,By0,Bz0,Px,Py,Pz,Ax,Ay,Az,Bx,By,Bz)
```

where `Mlist` is the list of *transformation matrix* parameters, as follows:

```
[[cosm, sinm, Rxm, Rym, Rzm, Wxm, Wym, Wzm],
      ...
      ...
 [cos2, sin2, Rx2, Ry2, Rz2, Wx2, Wy2, Wz2],
 [cos1, sin1, Rx1, Ry1, Rz1, Wx1, Wy1, Wz1]]
```

Other parameters are those defined above.

Since *transformation matrix* parameters can be expressed as lists, the program can handle any handling robot by manipulating their contents.

Let us consider, for example, a robot having 3 arms and 3 joints, expressed by the vector sketch in the following figure.

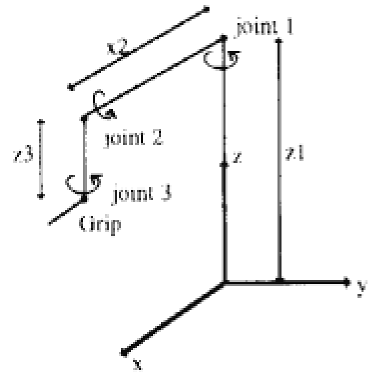


Figure-5: Vector sketch of example robot

The robot can rotate each joint and change the length of each arm. Therefore, the robot has 6 degrees of freedom. If we want to know the general relation among the target position and the target orientation, and the rotation angle of each joint and the length of each arm, we can obtain the result by issuing the following query.

```
?- robot:robot([[cos3, sin3, 0,  0, z3, 0, 0, 1],
                [cos2, sin2, x2, 0, 0,  1, 0, 0],
                [cos1, sin1, 0,  0, z1, 0, 0, 1]],
               5,0,0,1,0,0,0,1,0,
               px,py,pz,ax,ay,az,cx,cy,cz).
```

In the query, `px`, `py`, and `pz` represent the target position, and (`ax`, `ay`, `az`), and (`cx`, `cy`, `cz`) are two unit vectors that represent the orientation of the end effector. `sin`'s and `cos`'s represent the rotation angle of each joint, and `z3`, `x2`, and `z1` are the final lengths of each arm. Note that `sinn` and `conn` represent $sin\ \theta_n$, and $cos\ \theta_n$, respectively.

The answer to the query is as follows.

```
cos2^2   =  1-sin2^2
cos1^2   =  1-sin1^2
cos3^2   =  1-sin3^2
px       =  -5*cos2*sin3*sin1+z3*sin2*sin1+5*cos3*cos1+x2*cos1
py       =  5*cos3*sin1+x2*sin1+5*cos1*cos2*sin3-z3*cos1*sin2
pz       =  5*sin3*sin2+z1+z3*cos2
ax       =  -1*cos2*sin3*sin1+cos3*cos1
ay       =  cos3*sin1+cos1*cos2*sin3
az       =  sin3*sin2
cx       =  -1*cos1*sin3-cos3*cos2*sin1
cy       =  -1*sin3*sin1+cos3*cos1*cos2
cz       =  cos3*sin2
```

In this case, the parameters of the target position and the target orientation are expressed as functions of the length of each arm and the rotation angle of each joint.

Next, we can calculate the values of the length of each arm and the rotation angle of each joint when concrete values are assigned to all target position and target orientation parameters by using the same program. The query in this case is as follows.

```
?- alg:set_out_mode(float),
   alg:set_error1(1/1000000),
   alg:set_error2(1/100000000),
   robot:robot([[cos3, sin3, 0,  0, z3, 0, 0, 1],
                [cos2, sin2, x2, 0, 0,  1, 0, 0],
                [cos1, sin1, 0,  0, z1, 0, 0, 1]],
               5, 0, 0, 1, 0, 0, 0, 1, 0,
               40, -30, 20, -1/3, 2/3, -2/3, 2/3, 2/3, 1/3).
   alg:get_result(eq,1,nonlin,Res),
   alg:find(Res,Sol),
   alg:constr(Sol).
```

In this query, the target position is (40, -30, 20), and the target orientation is represented by (-1/3, 2/3, -2/3), and (2/3, 2/3, 1/3). For this query, the following two answers are possible according to the value of cos1. Obtaining these two answers was not possible in the original version of CAL. This can be done by using the functions of current CAL: the approximation of real roots, and the multiple context.

```
sin1 = 8.944272995e-1
sin2 = 7.453560829165e-1
sin3 = -8.9442729951e-1
cos1 = -4.4721364975e-1
cos2 = 6.66666666667e-1
cos3 = 4.4721364975e-1
z3   = 2.6e1
x2   = -4.69574332237e1
z1   = 6.0
```

and,

```
sin1 = -8.944272995e-1
sin2 = -7.453560829165e-1
sin3 = 8.9442729951e-1
cos1 = 4.4721364975e-1
cos2 = 6.66666666667e-1
cos3 = -4.4721364975e-1
z3   = 2.6e1
x2   = 4.69574332237e1
z1   = 6.0
```

These two solutions represent two different shapes of handling robot for which the position and the orientation of the end effector are identical.

4.3.2 Statics Program

We can ubapply the statics program to the results obtained with the kinematics program.

In the statics program, the head of the top level predicate is as follows.

```
robot2(Qlist,Mlist,Tlist,Gx,Gy,Gz,Pxm,Pym,Pzm,Fxm,Fym,Fzm)
```

where `Qlist` is the list of position parameters of each joint `[[qx1,qy1,qz1], [qx2,qy2,qz2], ..., [qxm,qym,qzm]]`. `Mlist` is same as for the kinematics program. `Tlist` is the list of torque parameter `[t1, t2, ... , tm]` working on each joint. `(Gx,Gy,Gz)` is same as that for the kinematics program. `(Pxm,Pym,Pzm)` is the position of the end-effector, and `(Fxm,Fym,Fzm)` is the force acting on the end-effector.

The program calculates the values of the parameters in `Qlist` and `Tlist` from the values of the parameters in `Mlist`. In the same way as in the kinematics program, the statics program is also structure-free, and able to describe any type of handling robot by changing the contents of the query. We will consider the same robot as that shown in Figure-5. The parameters needed to execute the statics program are obtained by the kinematics program. Thus, the query and answer in the same case as above are as follows.

```
?- alg:set_out_mode(float),
   alg:set_error1(1/1000000),
   alg:set_error2(1/100000000),
   robot:robot([[cos3, sin3, 0,  0, z3, 0, 0, 1],
                [cos2, sin2, x2, 0, 0,  1, 0, 0],
                [cos1, sin1, 0,  0, z1, 0, 0, 1]],
               5, 0, 0, 1, 0, 0, 0, 1, 0,
               40, -30, 20, -1/3, 2/3, -2/3, 2/3, 2/3, 1/3),
   alg:get_result(eq,1,nonlin,Res),
   alg:find(Res,Sol),
   alg:constr(Sol),
   robot:robot2([[qx1, qy1, qz1],
                 [qx2, qy2, qz2],
                 [qx3, qy3, qz3]],
                [[cos1, sin1, 0,  0, z1, 0, 0, 1],
                 [cos2, sin2, x2, 0, 0,  1, 0, 0],
                 [cos3, sin3, 0,  0, z3, 0, 0, 1]],
                    [t1, t2, t3],
                    5, 0, 0, pxm, pym, pzm, 0, 0, fzm).
```

Then, there are two answers to this query. The following are fragments of answers including values for `Qlist`, and `Tlist`.

The first answer is:

```
qx1 = 0.0                       t1 = 0.0
qy1 = 0.0                       t2 = 4.91934937379e1*fzm
qz1 = 7.0e1                     t3 = 1.66666658684*fzm
qx2 = -4.99999988026
qy2 = -9.99999976052
qz2 = 7.0e1
qx3 = 4.16666656688e1
qy3 = -3.33333325351e1
qz3 = 2.33333333333e1
```

and the second answer is:

```
qx1 = 0.0                       t1 = 0.0
qy1 = 0.0                       t2 = -4.91934937379e1*fzm
qz1 = 7.0e1                     t3 = 1.66666658684*fzm
qx2 = -4.99999988026
qy2 = -9.99999976052
qz2 = 7.0e1
qx3 = 4.16666656688e1
qy3 = -3.33333325351e1
qz3 = 2.33333333333e1
```

In this way, torque parameters are calculated as functions of **fzm**, which is the weight of the object held by the end-effector.

5 Other Constraint Solvers

Besides the algebraic constraint solver described in the previous sections, CAL version 2.1 features several constraint solvers for different domains.

1. Boolean Solver employing *Boolean Buchberger algorithm* [9, 1, 7],

2. Boolean Solver employing *incremental Boolean elimination algorithm* [6]

3. Linear Solver employing *Simplex method*, and

4. Set Solver employing *generalized Boolean Buchberger algorithm* [10]

Boolean constraints can be represented in terms of a polynomial ring of Boolean algebra {**0**, **1**}. The "*Boolean Buchberger algorithm*" is a variant of the Buchberger algorithm [9] which calculates Gröbner bases of this polynomial ring.

Another constraint solver for the truth values employs a new algorithm called the "*incremental Boolean elimination algorithm*" [6]. This algorithm is based on a concept similar to that of Boolean unification, but introduces no extra variables. This algorithm can obtain different cannonical forms from Boolean Gröbner bases.

For linear equations and linear inequalities, we have the "*linear solver*" that employs the Simplex method, in much the same way as other CLP languages.

The "*Generalized Boolean Buchberger algorithm*" [10] for the set solver is also an extension of the Boolean Buchberger algorithm, itself a variant of the Buchberger Algorithm. Its domain is a Boolean Algebra of *finite sets* and *co-finite sets*. The solver deals with any kind of constraints expressed in terms of equations of Boolean algebra. It can handle variables both for sets and elements.

6 Conclusion

We have described the current status of the constraint logic programming language CAL developed at ICOT and its application to handling robots.

Basically, CAL can be regarded as a scheme for constraint logic programming, because a user can attach his/her own constraint solver that follows our requirements of constraint solvers [7] for a CAL language processor.

CAL has several standard constraint solvers such as the algebraic solver that employs the Buchberger algorithm, the Boolean solver that employs the Boolean Buchberger algorithm, the incremental Boolean elimination method, a linear solver for linear equations and linear inequalities, and a set solver which employs a generalized Boolean Buchberger algorithm.

In the algebraic solver, we have incorporated the functionalities that load and save constraint sets, approximate real roots of univariate equations, and handle contexts, with several meta-predicates for constraint sets.

At ICOT, the overall direction of research on CLP language processor is now shifting from sequential processing towards parallel processing. We are now implementing the first trial of a parallel constraint logic programming language named GDCC (*Guarded Definite Clause with Constraint*), with parallel implementation of the Buchberger algorithm as the parallel algebraic solver and the parallel Boolean solver on top of our multi-processor machine, named Multi-PSI, using KL1 language.

Handling robot applications will also be shifted from CAL to GDCC. We are now aiming to develop a handling robot design support system based on the two CAL programs presented in this paper.

As described in the section 4, the design process of structure for handling robots can be decomposed into the following sequence:

1. Determining the configuration of the handling robot.
2. Deriving the relationship among arm lengths, joint rotation angles, position of the end effector, and its orientation.
3. Obtaining the Jacobian matrix of position parameters and orientation parameters with respect to joint rotation angles.
4. Obtaining singular points where the determinant of the Jacobian matrix is equal to 0.
5. Evaluating the configuration.

The two CAL programs presented in this paper can help the second stage of the support process presented in section 4. The Jacobian matrix and its determinant can also obtained using CAL. Therefore, the whole design process except the analysis of the Jacobian matrix can be supported in the context of the constraint logic programming. This analysis can be performed by introducing factorization.

Through this research work, we are aiming at a practical, efficient constraint logic programming language within the framework of parallel processing based on our experience gained through the research and development of CAL.

Finally, we would like to note that the CAL Version 2.1 system is now implementing on ESP language that runs on UNIX environment called Common ESP (CESP).

Acknowledgements

The authors are indebted to many people who have contributed this work over several years. We especially thank Dr. Kazuhiro Fuchi who gave us the opportunity of working on Constraint Logic Programming at ICOT. Dr. Ryuzo Hasegawa who gave us contenious encouragement to carry out our research activities. We also thank all other members in the fourth research laboratory of ICOT who took charge of various research themes on Constraint Logic Programming.

References

[1] A. Aiba, K. Sakai, Y. Sato, D. J. Hawley, and R. Hasegawa. Constraint Logic Programming Language CAL. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, November 1988.

[2] Takashi Chikayama. Unique features of ESP. In *Proceedings of FGCS'84*, pages 292-298, 1984.

[3] A. Colmerauer. Opening the Prolog III Universe: A new generation of Prolog promises some powerful capabilities. *BYTE*, pages 177-182, August 1987.

[4] Institute for New Generation Computer Technology. *Contrante Avec Logique version 2.12 User's manual*. in preparation.

[5] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *4th IEEE Symposium on Logic Programming*, 1987.

[6] S. Menju, K. Sakai, Y. Satoh, and A. Aiba. A Study on Boolean Constraint Solvers. Technical Report TM 1008, Institute for New Generation Computer Technology, February 1991.

[7] K. Sakai and A. Aiba. CAL: A Theoretical Background of Constraint Logic Programming and its Application. *Journal of Symbolic Computation*, 8(6):589-603, December 1989.

[8] S. Sato and A. Aiba. An Application of CAL to Robotics. Technical Report TM 1032, Institute for New Generation Computer Technology, February 1991.

[9] Y. Sato and K. Sakai. Boolean Gröbner Base. February 1988. LA-Symposium in winter, RIMS, Kyoto University.

[10] Y. Sato, K. Sakai, and S. Menju. Solving constraints over sets by Boolean Gröbner bases (In japanese). In *Proceedings of The Logic Programming Conference '91*, September 1991.

[11] M. H. van Emden and R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23(4), October 1976.