PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

CONSTRAINT LOGIC PROGRAMMING SYSTEM
– CAL, GDCC AND THEIR CONSTRAINT SOLVERS –

Akira Aiba and Ryuzo Hasegawa
Fourth Research Laboratory
Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan
{aiba, hasegawa}@icot.or.jp

Abstract

This paper describes constraint logic programming languages, CAL (*Contrainte Avec Logique*) and GDCC (*Guarded Definite Clauses with Constraints*), developed at ICOT.

CAL is a sequential constraint logic programming language with algebraic, Boolean, set, and linear constraint solvers. GDCC is a parallel constraint logic programming language with algebraic, Boolean, linear, and integer parallel constraint solvers.

Since the algebraic constraint solver utilizes the Buchberger algorithm, the solver may return answer constraints including univariate nonlinear equations. The algebraic solvers of both CAL and GDCC have the functions to approximate the real roots of univariate equations to obtain all possible values of each variable. That is, this function gives us the situation in which a certain variable has more than one value. To deal with this situation, CAL has a multiple environment handler, and GDCC has a block structure.

We wrote several application programs in GDCC to show the feasibility of the constraint logic programming language.

1 Introduction

The Fifth Generation Computer System (FGCS) project is a Japanese national project that started in 1982. The aim of the project is to research and develop new computer technologies for knowledge and symbol processing parallel computers.

The FGCS prototype system has three layers: the prototype hardware system, the basic software system, and the knowledge programming environment. Parallel application software has been developed for these. The constraint logic programming system is one of the systems that form, together with the knowledge base construction and the programming environment, the knowledge programming environment. In this paper, we describe the overall research results of constraint logic programming systems in ICOT.

The programming paradigm of constraint logic programming (*CLP*) was proposed by A. Colmerauer [Colmerauer 1987] and J. Jaffar and J-L. Lassez [Jaffar and Lassez 1987] as an extension of logic programming by extending its computation domain. Jaffar and Lassez showed that CLP possesses logical, functional, and operational semantics which coincide with each other, in a way similar to logic programming [van Emden and Kowalski 1976].

In 1986, we began to research and develop high-level programming languages suitable for problem solving to achieve our final goal, that is, developing efficient and powerful parallel CLP languages on our parallel machine.

The descriptive power of a CLP language is strongly depend on its constraint solver, because a constraint solver determines the domain of problems which can be handled by the CLP language. Almost all existing CLP languages such as Prolog III [Colmerauer 1987] and CLP($\mathcal{R}$) [Jaffar and Lassez 1987] has a constraint solver for linear equations and linear inequalities.

Unlike the other CLP languages, we focused on nonlinear algebraic equation constraints to deal with problems which are described in terms of nonlinear equations such as handling robot problem. For the purpose, we selected the Buchberger algorithm for a constraint solver of our languages.

Besides of nonlinear algebraic equations, we were also interested in writing Boolean constraints, set constraints, linear constraints, and hierarchical constraints in our framework. For Boolean constraints, we modify the Buchberger algorithm to be able to handle Boolean constraints, and later, we developed the algorithm for Boolean constraints based on the Boolean unification. For set constraints, we expand the algorithm for Boolean constraints based on the Buchberger algorithm. We also implemented the simplex method to deal with linear equations and linear inequalities same as the other CLP languages. Furthermore, we tried to handle hierarchical constraints in our framework.

We developed two CLP language processors, first we implemented a language processor for sequential CLP

language named CAL (*Contrainte Avec Logique*) on sequential inference machine PSI, and later, we implemented a language processor for parallel CLP language named GDCC (*Guarded Definite Clauses with Constraints*), based on our experiments on extending CAL processor by introducing various functions.

In Section 2, we briefly review CLP, and in Section 3, we describe CAL. In Section 4, we describe GDCC, and in Section 5, we describe various constraint solvers and their parallelization. In Section 6, we introduce application programs written in our languages.

2 CLP and the role of the constraint solver

CAL and GDCC belong to the family of CLP languages. The concept of CLP stems from the common desire for easy programming. In fact, as claimed in the literature [Jaffar and Lassez 1987, Sakai and Aiba 1989], the CLP is a scheme of programming languages with the following outstanding features:

- Natural declarative semantics.
- Clear operational semantics that coincide with the declarative semantics.

Therefore, it gives the user a paradigm of declarative (and thus, hopefully easy) programming and gives the machine an effective mechanism for execution that coincide with the user's declaration.

For example, in Prolog (the most typical instance of CLP), we can read and write programs in declarative style like "... if ... and ...". The system execute these by a series of operations with unification as its basic mechanism.

Almost every CLP language has a similar programming style and a mechanism which plays the similar role to the unification mechanism in Prolog, and the execution of programs depends on the mechanism heavily. We call such a mechanism the constraint solver of the language.

Usually, a CLP language aims at a particular field of problems and its solver has special knowledge to solve the problems. In the case of Prolog, the problems are syntactic equalities between terms, that is, the unification. On the other hand, CAL and GDCC are tuned to deal with the following:

- algebraic equations
- Boolean equations
- set inclusion and membership
- linear inequalities

These relations are called constraints.

In the CLP paradigm, a problem is expressed as constraints on the objects in the problem. Therefore, an often cited benefit of CLP is that "One does not need to write an implementation but a specification." In other words, all that a programmer should write in CLP is constraints between the objects, but not how to find objects satisfying the relation. To be more precise, such constraints are described in the form of a logical combination of formulas each of which expresses a basic unit of the relation.

Though there are many others, the above benefit surely expresses an important feature of CLP. Building an equation is usually easier than solving it. Similarly, one may be able to write down the relation between the objects without knowing the method to find the appropriate values of objects which satisfy the relation.

An ideal CLP system should allow a programmer to write any combination of any well-formed formulas. The logic programming paradigm gives us a rich framework for handling logical combinations of constraints. However, we still need a powerful and flexible constraint solver to handle each constraint. To discuss the function of the constraint solver from a theoretical point of view, the declarative semantics of CLP [Sakai and Aiba 1989] gives us several criteria. Assume that constraints are given in the form of their conjunction. Then, the following are the criteria.

(1) Can the solver decide whether a given constraint is satisfiable?

(2) Given satisfiable constraints, is there any way for the solver to express all the solutions in simplified form?

Prolog's constraint solver, the unification algorithm, answers these criteria affirmatively and so do the solvers in CAL and GDCC. In fact, they satisfy the following stronger requirements almost perfectly:

(3) Given a set of constraints, can the solver compute the simplest form (called the canonical form of the constraints) in a certain sense?

However, these criteria may not be sufficient from an applicational point of view. For example, we may sometimes be asked the following:

(4) Given satisfiable constraints, can the solver find at least one concrete solution?

Finding a concrete solution is a question usually independent of the above and may be proved theoretically impossible to answer. Therefore, we may need an approximate solution to answer this partly. As discussed later, we incorporated many of the constraint solvers and functions into CAL and GDCC.

Another important feature of constraint solvers is their incrementality. An incremental solver can be given a constraint successively. It reduces each constraint as simple

as possible by the current set of constraints. Thus, an incremental solver finds the unsatisfiability of a set of constraints as early as possible and makes Prolog-type backtracking mechanism efficient. Fortunately, the solvers of CAL and GDCC are fully incremental like unification.

3 CAL - Sequential CLP Language

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

3.1 CAL language

The syntax of CAL is similar to that of Prolog, except for its constraints. A CAL program features two types of variables: logical variables denoted by a sequence of alphanumeric characters starting with an uppercase letter (as with Prolog variables), and constraint variables denoted by a sequence of alphanumeric characters starting with a lowercase letter. Constraint variables are global variables, while logical variables are local variables within the clauses in which they occur. This distinction is introduced to simplify incremental querying.

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

```
:- 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 `S` from the height `H` and the baseline length `L`. The second expresses 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.).

In the following query, `heron`, shows the name of the file in which the CAL source program is defined.

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

This query asks for the general relationship between the lengths of the three edges and the surface area.

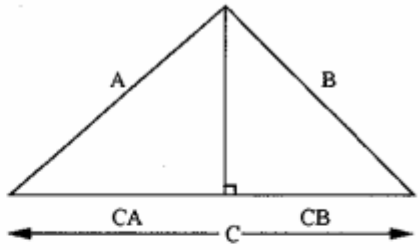

Figure 1: The third clause

The invocation of `alg:pre(s,10)` defines the precedence of the variable `s` to be 10. Since the algebraic constraint solver utilizes the Buchberger algorithm, ordering among monomials is essential for computation. This command changes the precedence of variables. Initially, the precedences of all variables are assigned to 0. Therefore, in this case, the precedence of variable `s` is raised.

To this query, the system responds with the following equation [1]:

```
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.

When we call the query

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

the CAL system returns the following answer:

```
s^2 = 36
```

If a variable has finitely many values in all its solutions, there is a way of obtaining a univariate equation with the variable in the Gröbner base. Therefore, if we can add a function that enables us to compute the approximate values of the solutions of univariate equations, we can approximate all possible value of the variable.

For this purpose, we implemented a method of approximating the real roots of univariate polynomials. In CAL, all real roots of univariate polynomials are isolated by obtaining a set of intervals, each of which contains one real root. Then, each isolated real root is approximated by the given precision.

For application programs, we wanted to use approximate values to simplify other constraints. The general method to do this is to input equations of variables and their approximate values as constraints. For this purpose, we had to modify the original algorithm to compute Gröbner bases to accept approximate values.

When we call the query

[1] This equation represents the expression

$$s^2 = -\frac{1}{16}b^4 + \frac{1}{8}a^2b^2 - \frac{1}{16}a^4 + \frac{1}{8}c^2b^2 + \frac{1}{8}c^2a^2 - \frac{1}{16}c^4$$

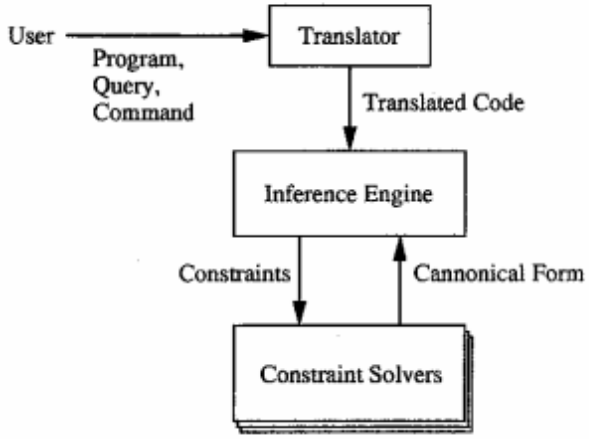

Figure 2: Overall construction of CAL language processor

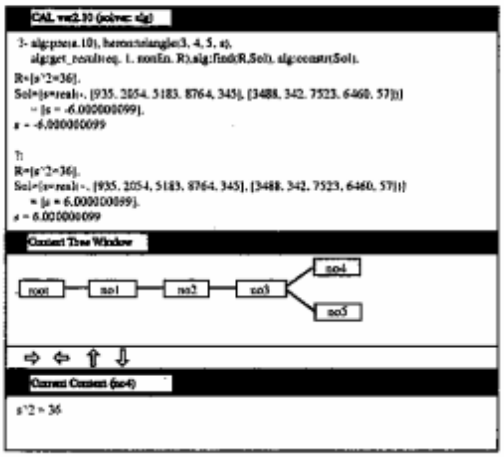

Figure 3: CAL system windows

```
?- 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).
```

we can obtain the answers `s = -6.000000099` and `s = 6.000000099`, successively by backtrack.

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.

The essence of the above query is invocations of `alg:get_result/4`, and `alg:find/2`. 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 the constraint to reduce other constraints in the Gröbner base.

3.2 Configuration of CAL system

In this section, we will introduce the overall structure of the CAL system.

The CAL language processor consists of a translator, a inference engine, and constraint solvers. These subsystems are combined as shown in Figure 2.

The translator receives input from a user, and translates it into ESP code. Thus, a CAL source program is translated into the corresponding ESP program by the translator, which is executed by the inference engine. An appropriate constraint solver is invoked everytime the inference engine finds a constraint during execution.

The constraint solver adds the newly obtained constraint to the set of current constraints, and computes the canonical form of the new set.

At present, CAL offers the five constraint solvers discussed in Section 1.

3.3 Context

To deal with a situation in which a variable has more than one value, as in the above example, we introduced context and context tree.

A context is a set of constraints. A new context is created whenever the set is changed. In CAL, contexts are represented as nodes of a context tree. The root of a context tree is called the root context. The user is supposed to be in a certain context called the current context.

A context tree is changed in the following cases:

1. Goal execution:
   A new context is created as a child-node of the current context in the context tree.
2. Creation of a new set of constraints by requiring other 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 all cases, the newly created node represents the new set of constraints and becomes the current context.

Several commands are provided to manipulate the context tree: These include a command to display the contents of a context, a command to set a context as the

current context, and a command to delete the sub-tree of contexts from the context tree.

Figure 3 shows an example of the CAL processor window.

4 GDCC - Parallel CLP Programming Language

There are two major levels to parallelizing CLP systems. One is the execution of the Inference Engines and the Constraint Solvers in parallel. The other is the execution of a Constraint Solvers in parallel. There are several works on the parallelization of CLP systems: a proposal of ALPS [Maher 1987] introducing constraints into committed-choice language, a report of some preliminary experiments on integrating constraints into the PEPSys parallel logic system [Van Hentenryck 1989], and a framework of concurrent constraint (cc) language for integrating constraint programming with concurrent logic programming languages [Saraswat 1989].

The cc programming language paradigm models computation as the interaction among multiple cooperating agents through the exchange of query and assertion messages into a central store as shown in Figure 4.

In Figure 4, query information to the central store is represented as *Ask* and assertion information is represented as *Tell*.

This paradigm is embedded in a guarded (conditional) reduction system, where the guards contain the queries and assertions. Control is achieved by requiring that the queries in a guard are true (entailed), and that the assertions are consistent (satisfiable), with respect to the current state of the store. Thus, this paradigm has high affinity with KL1 [Ueda and Chikayama 1990], our basic parallel language.

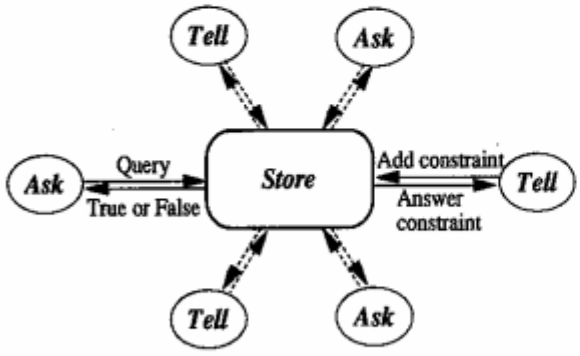

Figure 4: The cc language schema

GDCC (Guarded Definite Clauses with Constraints), which satisfies two level parallelism, is a parallel CLP language introducing the framework of cc. It is implemented in KL1 and is currently running on the Multi-PSI machine. GDCC includes most of KL1, since KL1 built-in predicates and unification can be regarded as a distinguished domain called HERBRAND [Saraswat 1989].

GDCC contains *Store*, a central database to save the canonical forms of constraints. Whenever the system meets an *Ask* or *Tell* constraint, the system sends it to the proper solver. *Ask* constraints are only allowed passive constraints which can be solved without changing the content of the *Store*. While in the *Tell* part, constraints which may change the *Store* can be written. In the GDCC program, only *Ask* constraints can be written in guards. This is similar to the KL1 guard in which active unification is inhibited.

GDCC supports multiple plug-in constraint solvers so that the user can easily specify a proper solver for a domain.

In this section, we briefly explain the language syntax of GDCC and its computation model. Then, the outline of the system is described. For further information about the implementation and the language specification, refer to [Terasaki *et al.* 1992].

4.1 GDCC language

A clause in GDCC has the following syntax:

Head :- *Ask* | *Tell*, *Goal*.

where, *Head* is a head part of a clause, "|" is a commit operator. *Goal* is a sequence of predicate invocations. *Ask* denotes Ask-constraints and invocations of KL1 built-in guard predicates, and *Tell* means Tell-constraints.

A clause is *entailed* if and only if *Ask* is reduced to *true*. Any clause with guards which cannot be reduced to either *true* or *false* is suspended. The body part, the right hand side of the commit operator, is evaluated if and only if *Ask* is *entailed*. Clauses whose guards are reduced true are called candidate clauses. A GDCC program fails when either all candidate clauses are rejected or there is a failure in evaluating *Tell* or *Goals*.

The next program is `pony_and_man` written in GDCC:

```
pony_and_man(Heads,Legs,Ponies,Men) :- true |
    alg# Heads= Ponies + Men,
    alg# Legs=  4*Ponies + 2*Men.
```

where, `true` is an *Ask* constraint which is always reduced as *true*. In the body, equations which begin with `alg#` are *Tell* constraints. `alg#` indicates that the constraints are solved by the algebraic solver. In a body part, not only *Tell* constraints but normal KL1 predicates can be written as well. Bi-directionality in evaluation of constraints, an important characteristic of CLP, is not spoiled by this limitation. For example, the query

```
?-  pony_and_man(5,14,Ponies,Men).
```

will return `Ponies=2`, and `Men=3`, and the query

```
?- pony_and_man(Heads,Legs,2,3).
```
will return `Heads=5`, and `Legs=14`, same as in CAL.

4.2 GDCC system

The GDCC system consists of the compiler, the shell, the interface and the constraint solvers. The compiler translates a GDCC source program into KL1 code. The shell translates queries and provides rudimentary debugging facilities. The debugging facilities comprise the standard KL1 trace and spy functions, together with solver-level event logging. The shell also provides limited support for incremental querying. The interface interacts with a GDCC program (object code), sends body constraints to a solver and checks guard constraints using results from a solver.

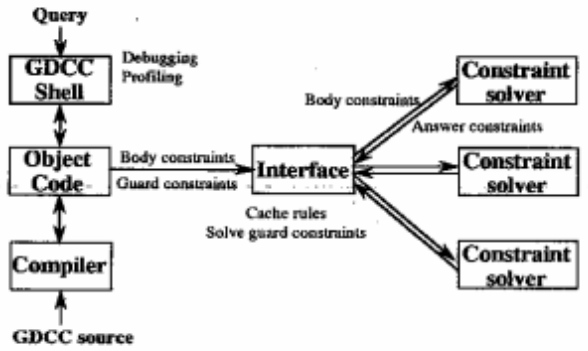

Figure 5: System Configuration of GDCC

The GDCC system is shown in Figure 5. The components are concurrent processes. Specifically, a GDCC program and the constraint solvers may execute in parallel, synchronizing only when, and to the extent, necessary at the program's guard constraints. That is, program execution proceeds by selecting a clause, and attempting to solve the guards of all its clauses in parallel. If one guard succeeds, the evaluation of the other guards is abandoned, and execution of the body can begin. In parallel with execution of the body goals by the inference engine, any constraints occurring in the body are passed to the constraint solver as they are being produced by the inference engine. This style of cooperation is very loosely synchronized and more declarative than sequential CLP.

4.3 Block

In order to apply GDCC to problems such as handling robot design problem [Sato and Aiba 1991], there were two major issues: handling multiple environments and synchronizing the inference engine with the constraint solvers. For instance, when the solution $X^2 = 2$ is derived from the algebraic solver, it must be solved in more detail using a function to compute the approximate real roots in univariate equations. There are two constraint sets in this example, one includes $X = \sqrt{2}$ and the other includes $X = -\sqrt{2}$. In the CAL system, the system selects one constraint set from these two and solves it, then, the other is computed by backtracking (i. e. , a system forces a failure). In committed-choice language GDCC, however, we cannot use backtracking to handle multiple environments. A similar problem occurs when a meta operation to constraint sets is required such as when computing a maximum value with respect to a given objective function. Before executing a meta operation, all target constraints must be sent to the solver. In a sequential CLP, this can be controlled when this description is written in a program. While in GDCC, we need another kind of mechanism to specify a synchronization point, since the sequence of clauses in a program does not relate to the execution sequence.

Introducing local constraint sets, however, which are independent to the global ones, can eliminate these problems. Multiple environments are realized by considering each multiple local constraint as one context. An inference engine and constraint solvers can be synchronized after evaluating a local constraint set.

Therefore, we introduced a mechanism called *block* to describe the scope of a constraint set. We can solve a certain goal sequence with respect to a local constraint set in a block. To encapsulate failure in a block, the *shoen* mechanism of PIMOS [Chikayama *et al.* 1988] is used.

5 Constraint Solvers and Parallelization

In this section, constraint solvers for both CAL and GDCC are briefly described. First, we describe the algebraic constraint solver for both CAL and GDCC. Then, we describe two Boolean constraint solvers – one is a solver utilizing the modified Buchberger algorithm and the other is a solver utilizing the incremental Boolean elimination algorithm. The former is for both CAL and GDCC, while the later is for CAL alone. Third, an integer constraint solver for GDCC is described, and fourth, a hierarchical constraint solver for CAL and GDCC is described. In the next subsection,a set constraint solver for CAL is described. And in the last subsection, a preliminary consideration on efficiency improvement of the algebraic constraint solver by applying dependency analysis of constraints.

All constraint solvers for CAL are written in ESP, and those for GDCC are written in KL1.

5.1 Algebraic Constraint Solver

The constraint domain of the algebraic solver is multivariate (non-linear) algebraic equations. The Buchberger

algorithm [Buchberger 1985] is a method to solve nonlinear algebraic equations which have been widely used in computer algebra over the past years.

Recently, several attempts have been made to parallelize the Buchberger algorithm, with generally disappointing results in absolute performance [Ponder 1990, Senechaud 1990, Siegl 1990], except in shared-memory machines [Vidal 1990, Clarke *et al.* 1990]. We parallelize the Buchberger algorithm while laying emphasis on absolute performance and incrementality rather than on deceptive parallel speedup. We have implemented several versions and continue to improve the algorithm.

In this section, we outline both the sequential version and the parallel version of the Buchberger algorithm.

5.1.1 Gröbner base and Buchberger algorithm

Without loss of generality, we can assume that all polynomial equations are in the form of $p = 0$. Let $E = \{p_1 = 0, \ldots, p_n = 0\}$ be a system of polynomial equations. Buchberger introduced the notion of a Gröbner base and devised an algorithm to compute the basis of a given set of polynomials. A rough sketch of the algorithm is as follows (see [Buchberger 1985] for a precise definition).

Let a certain ordering among monomials and a system of polynomials be given. An equation can be considered a rewrite rule which rewrites the greatest monomial in the equation to the polynomial consisting of the remaining monomials. For example, if the ordering is $Z > X > B > A$, a polynomial equation, $Z - X + B = A$, can be considered to be the rewrite rule, $Z \rightarrow X - B + A$. A pair of rewrite rules $L_1 \rightarrow R_1$ and $L_2 \rightarrow R_2$, of which L_1 and L_2 are not mutually prime, is called a *critical pair*, since the least common multiple of their left-hand sides can be rewritten in two different ways. The S-polynomial of such a pair is defined as:

$$\text{S-poly}(L_1, L_2) = R_1 \frac{lcm(L_1, L_2)}{L_2} - R_2 \frac{lcm(L_1, L_2)}{L_1}$$

where $lcm(L_1, L_2)$ represents the least common multiplier of L_1 and L_2.

If further rewriting does not succeed in rewriting the S-polynomial of a critical pair to zero, the pair is said to be *divergent* and the S-polynomial is added to the system of equations. By repeating this procedure, we can eventually obtain a confluent rewriting system. The confluent rewriting system thus obtained is called a *Gröbner base* of the original system of equations.

If a Gröbner base does not have two rules, one of which rewrites the other, the Gröbner base is called *reduced*. The *reduced* Gröbner base can be considered a canonical form of the given constraint set since it is unique with respect to the given ordering of monomials. If all the solutions of a equation $f = 0$ are included in the solution set of E, then f is rewritten to zero by the Gröbner base of E. On the contrary, if a set of polynomials E has no solution, then the Gröbner base of E includes "1". Therefore, this algorithm has good properties for deciding the satisfiability of a given constraint set.

5.1.2 Parallel Algorithm

The coarse-grained parallelism in the Buchberger algorithm, suitable for the distributed memory machine, is the parallel rewriting of a set of polynomials. However, since the convergence rate of the Buchberger algorithm is very sensitive to the order in which polynomials are converted into rules, implementation must carefully select small polynomials at an early stage. We have implemented solvers in three different architectures: namely, a pipeline, a distributed architecture, and a master-slave architecture. We briefly mention here the master-slave architecture since this solver has comparatively good performance.

Figure 6 shows the architecture.

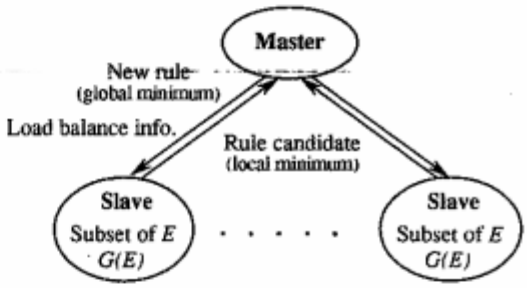

Figure 6: Architecture of master-slave type solver

The set of polynomials E is physically partitioned with each slave taking a different part. The initial rule set of $G(E)$ is duplicated so that all slaves use the same rule set. New polynomials are distributed to the slaves by the master. The outline of the reduction cycle is as follows.

Each slave rewrites its own polynomials by the $G(E)$, selects the local minimum polynomial from them, and sends its leading power product to the master. The master processor waits for reports from all the slaves, and selects the global minimum power products. The minimum polynomial can be decided only after all slaves finish reporting to the master. A polynomial, however, which is not the minimum can be decided quickly. Thus, the *not-minimum* message is sent to slaves as soon as possible, and the processors that receive the *not-minimum* message reduce polynomials by the old rule set while waiting for a new rule. While the slave is receiving the *minimum* message, the slave converts the polynomial into a new rule and sends it to the master. The master sends the new rule to all slaves except the owner. If more than one candidate have equal power products, then all of these

candidates are converted to rules by slaves and they go to final selection at the master.

Table 1 shows the results of the benchmark problems. The problems are adopted from [Boege *et al.* 1986, Backelin and Fröberg 1991]. Refer to [Terasaki *et al.* 1992] for further details.

Table 1: Timing and speedup of the master-slave arch.(unit:sec)

Problems	Processors				
	1	2	4	8	16
Katsura-4	8.90	7.00	5.83	6.53	9.26
	1	1.27	1.53	1.36	0.96
Katsura-5	86.74	57.81	39.88	31.89	36.00
	1	1.50	2.18	2.72	2.41
Cyc.5-roots	27.58	21.08	19.27	19.16	25.20
	1	1.31	1.43	1.44	1.10
Cyc.6-roots	1430.18	863.62	433.73	333.25	323.38
	1	1.66	3.30	4.29	4.42

5.2 Boolean Constraint Solver

There are several algorithms that solve Boolean constraints, but we do not know so many that we can get the canonical form of constraints, one that can calculate solutions incrementally and that uses no parameter variables. These criteria are important for using the algorithm as a constraint solver, as we described in Section 2. First, we implemented the Boolean Buchberger algorithm [Sato and Sakai 1988] for the CAL system, then we tried to parallelize it for the GDCC system. This algorithm satisfies all of these criteria. Moreover, we developed another sequential algorithm named Incremental Boolean elimination, that also satisfies all these criteria, and we implemented it for the CAL system.

5.2.1 Constraint Solver by Buchberger Algorithm

We first developed a Boolean constraint solver based on the modified Buchberger algorithm called the Boolean Buchberger algorithm [Sato and Sakai 1988, Aiba *et al.* 1988]. Unlike the Buchberger algorithm, it works on the Boolean ring instead of on the field of complex numbers. It calculates the canonical form of Boolean constraints called the Boolean Gröbner base. The constraint solver first transforms formulas including some Boolean operators such as *inclusive-or* ($\vee$) and/or *not* ($\neg$) to expressions on the Boolean ring before applying the algorithm.

We parallelized the Boolean Buchberger algorithm in KL1. First we analyzed the execution of the Boolean Buchberger algorithm on CAL for some examples, then we found the large parts that may be worth parallelizing, rewriting formulas by applying rules. We also tried to find parts in the algorithm which can be parallelized by analyzing the algorithm itself. Then, we decided to adopt a master-slave parallel execution model.

In a master-slave model, one master processor plays the role of the controller and the other slave processors become the reducers. The controller manages Boolean equations, updates the temporary Gröbner bases (**GB**) stored in all slaves, makes S-polynomials and self-critical pair polynomials, and distributes equations to the reducers. Each reducer has a copy of **GB** and reduces equations which come from the controller by **GB**, and returns non-zero reduced equations to the controller. When the controller becomes idle after distributing equations, the controller plays the role of a reducer during the process of reduction.

For the 6-queens problem, the speedup ratio of 16 processors to a single processor is 2.96. Because the parallel execution part of the problem is 77.7% of whole execution, the maximum speedup ratio is 4.48 in our model. The difference is due to the task distribution overhead, the update of **GB** in each reducer, and the imbalance of distributed tasks.

Then, we improved our implementation so as not to make redundant critical pairs. This improvement causes the ratio of parallel executable parts to decrease, so the improved version becomes faster than the original version, but the speedup ratio of 16 processors to a single processor drop to 2.28.

For more details on the parallel algorithm and results, refer to [Terasaki *et al.* 1992].

5.2.2 Constraint Solver by Incremental Boolean Elimination Algorithm

Boolean unification and SL-resolution are well known as Boolean constraint solving algorithms other than the Boolean Buchberger algorithm. Boolean unification is used in CHIP [Dincbas *et al.* 1988] and SL-resolution is used in Prolog III [Colmerauer 1987]. Boolean unification itself is an efficient method. It becomes even more efficient using the binary decision diagrams (BDD) as data structures to represent Boolean formulas. Because the solutions by Boolean unification include extra variables introduced during execution, it cannot calculate any canonical form of the given constraints if we execute it incrementally. For this reason, we developed a new algorithm, *Incremental Boolean elimination*. As with the Boolean unification, this algorithm is based on Boole's elimination, but it introduces no extra variables, and it can calculate a canonical form of the given Boolean constraints.

We denote Boolean variables by $x, y, z, \ldots$ and Boolean polynomials by $A, B, C, \ldots$. We represent all Boolean formulas only by logical connectives *and* ($\times$) and *exclusive-or* ($+$). For example, we can represent Boolean formulas $F \wedge G$, $F \vee G$ and $\neg F$ by $F \times G$, $F \times G + F + G$ and $F + 1$. We use the expression $F_{x=G}$ to represent the formula obtained by substituting all occurrences of variable x in formula F with formula G. We omit $\times$ symbols

as usual when there is no confusion. We assume that there is a total order over variables.

We define the *normal* Boolean polynomials recursively as follows.

1. The two constants 0, and 1 are *normal*.
2. If two normal Boolean polynomials A and B consist of only variables smaller than x, then $Ax + B$ is *normal*, and we denote it by $Ax \dot{+} B$. We call A the *coefficient of* x.

If variable x is at a maximum in formula F, then we can transform F to the normal formula $(F_{x=0}+F_{x=1})x \dot{+} F_{x=0}$. Hence we assume that all polynomials are normal.

Boole's elimination says that if a Boolean formula F is 0, then $F_{x=0} \times F_{x=1}$ $(= G)$ is also 0. Because G does not include x, if F includes x, then G includes fewer variables than F. Similarly we can get polynomials with fewer variables gradually by Boole's eliminations.

Boolean unification unifies x with $(F_{x=0}+F_{x=1}+1)u+F_{x=0}$ after eliminating variable x from formula F, where u is a free extra variable. This unification means the substitution x with $(F_{x=0}+F_{x=1}+1)u+F_{x=0}$, when a new Boolean constraint with variable x is given, the result of the substitution contains u instead of x. Therefore, Boolean unification unifies u with a formula with another extra variable.

Incremental Boolean elimination applies the following reduction to every formula instead of transforming $F = 0$ to $x = (F_{x=0} + F_{x=1} + 1)u + F_{x=0}$ and unifying x with $(F_{x=0} + F_{x=1} + 1)u + F_{x=0}$. That is why the Incremental Boolean elimination needs no extra variables.

Reduction A formula Cx $(C \not\equiv 1)$ is reduced by the formula $Ax \oplus B = 0$ shown below. This reduction tries to reduce the coefficient of x to 1 if possible, otherwise it tries to reduce it to the smallest formula possible.

$$\begin{array}{ll} Cx \to x + BC + B & (AC + A + C \equiv 1) \\ Cx \to (A+1)Cx + BC & (\text{otherwise}) \end{array}$$

When a new Boolean constraint is given, the following operation is executed, since Incremental Boolean elimination does not execute unification.

Merge Operation Let $Cx \dot{+} D = 0$ be a new constraint, and suppose that we have a constraint $Ax \dot{+} B = 0$. Then we make the merged constraint $(AC+A+C)x \dot{+} (BD+B+D) = 0$ the new solution. If the normal form of $ACD + BC + CD + D$ is not 0, we successively apply the merge operation to it.

This operation is an expansion of Boole's elimination. That is, if we have no constraint yet, we can consider A and B as 0. In this case, the merge operation is the same as Boole's elimination.

Example Consider the following constraints. Exactly one of five variables a, b, c, d, e $(a < b < c < d < e)$ is 1.

$$\begin{array}{l} a \wedge b = 0. \quad a \wedge c = 0. \quad a \wedge d = 0. \quad a \wedge e = 0. \quad b \wedge c = 0. \\ b \wedge d = 0. \quad b \wedge e = 0. \quad c \wedge d = 0. \quad c \wedge e = 0. \quad d \wedge e = 0. \\ a \vee b \vee c \vee d \vee e = 1 \end{array}$$

By Incremental Boolean elimination, we can obtain the following canonical solution.

$$\begin{array}{rcl} e & = & d + c + b + a + 1 \\ (c + b + a) \times d & = & 0 \\ (b + a) \times c & = & 0 \\ a \times b & = & 0 \end{array}$$

The solution can be interpreted as follows. Because the solution does not have an equation of the form $A \times a = B$, variable a is free. Because $a \times b = 0$, if $a = 1$ then the variable b is 0. Otherwise b is free. The discussion continues and, finally, because $e = d + c + b + a + 1$, if a, b, c, d are all 0, then variable e is 1. Otherwise e is 0.

By assignment of 0 or 1 to all variables in increasing order of $<$ under a solution by Boolean Incremental elimination, we can easily obtain any assignments that satisfy the given constraints. Thus, by introducing an adequate order to variables, we can obtain a favorite enumeration of assignments satisfy the given constraints.

5.3 Integer Linear Constraint Solver

The constraint solver for the integer linear domain checks the consistency of the given equalities and inequalities of the rational coefficients, and, furthermore, gives the maximum or minimum values of the objective linear function under these constraint conditions. The purpose of this constraint solver is to provide an efficient constraint solver for the integer optimization domain by achieving a computation speedup incorporating parallel execution into the search process.

The integer linear solver utilizes the rational linear solver (parallel linear constraint solver) for the optimization procedure to obtain an evaluation of relaxed linear problems created in the course of its solution. A rational linear solver is realized by the simplex algorithm. We implemented the integer linear constraint solver for GDCC.

5.3.1 Integer Linear Programming and Branch and Bound Method

In the following, we discuss a parallel search method employed in this integer linear constraint solver. The problem we are addressing is a mixed integer programming problem, namely, to find the maximum or minimum value of a given linear function under the integer linear constraints.

The problem can be defined as follows: The problem is to minimize the following objective function on variables

x_j which run on real numbers, and variables y_j which run on integers:

$$z = \sum_{i=1}^{n} p_i x_i + \sum_{i=1}^{m} q_i y_i$$

under the linear constraint conditions:

$$\sum_{i=1}^{n} a_{ij} x_i + \sum_{i=1}^{m} b_{ij} y_i \geq e_j, \text{ for } j = 1, \ldots, l.$$

$$\sum_{i=1}^{n} c_{ij} x_i + \sum_{i=1}^{m} d_{ij} y_i = f_j, \text{ for } j = 1, \ldots, k.$$

where

$$x_i \in \boldsymbol{R}, \text{ and } x_i \geq 0, \text{ for } i = 1, \ldots, n$$

$$y_i \in \boldsymbol{Z}, \text{ where } l_i \leq y_i \leq u_i,$$

$$l_j, u_j \in \boldsymbol{Z}, \text{ for } i = 1, \ldots, m$$

and

$$a_{ij},\ b_{ij},\ c_{ij},\ d_{ij},\ e_i,\ f_i \text{ are real constants.}$$

The method we use is the Branch-and-Bound algorithm. Our algorithm checks in the first place the solution of the original problem without requiring variables y_i in the above to take integer value. We call this problem a continuously relaxed problem. If the continuously relaxed problem does not have an integer solution, then we proceed by dividing the original problem into two sub-problems successively, producing a tree structured search space.

Continuously relaxed problems can be solved by the simplex algorithm, and if the original integer variables have exact integer values, then it yields the solution to the integer problem. Otherwise, we select an integer variable y_s which takes a non-integer value $\bar{y}_s$ for the solution of continuously relaxed problems, and imposes two different interval constraints derived from neighboring integers of the value $\bar{y}_s$, $l_s \leq y_s \leq [\bar{y}_s]$ and $[\bar{y}_s] + 1 \leq y_s \leq u_s$ to the already existing constraints, and obtains two child problems (See Figure 7). Continuing this procedure, which is called branching, we go on dividing the search space to produce more constrained sub-problems. Eventually this process leads to a sub-problem with the continuous solution which is also the integer solution of the problem. We can select the best integer solution from among those found in the process.

While the above branching process only enumerates integer solutions, if we have a measure to guarantee that a sub-problem cannot have a better solution compared to the already obtained integer solution in terms of the optimum value of the objective function, then we can skip that sub-problem and only need to search the rest of the nodes. Continuously relaxed problems give a measure for this, since these relaxed problems always have better optimum values for the objective function than the original integer problems. Sub-problems whose continuously relaxed problems have no better optimum than the integer

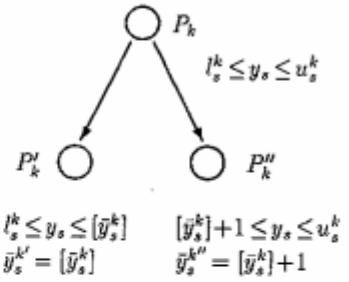

Figure 7: Branching of Nodes

solution obtained already cannot give a better optimum value, which means it is unnecessary to search further (bounding procedure).

We call these sub-problems obtained through the branching process search nodes.

The following two important factors decide the order in which the sequential search process goes through nodes in the search space:

1. The priorities of sub-problems(nodes) in deciding the next node on which the branching process works.

2. Selection of a variable out of the integer variables with which the search space is divided.

It is preferable that the above selections are done in such a way that the actual nodes searched in the process of finding the optimal form as small a part of the total search space as possible. We adopted one of the best heuristics of this type from operations research as a basis of our parallel algorithm([Benichou *et al.* 1971]).

5.3.2 Parallelization of Branch-and-Bound Method

As a parallelization of the Branch-and-Bound algorithm, we distribute search nodes created through the branching process to different processors, and let these processors work on their own sub-problems following a sequential search algorithm. Each sequential search process communicates with other processes to transmit information on the most recently found solutions and on pruning sub-nodes, thus making the search proceed over a network of processors. We adopted one of the best search heuristics used in sequential algorithms. Heuristics are used for controlling the schedule of the order of sub-nodes to be searched, in order to reduce the number of nodes needed to get to the final result. Therefore, it is important in designing parallel versions of search algorithms to balance the distributed load among processors, and to communicate information for pruning as fast as possible between these processors.

We considered a parallel algorithm design derived from the above sequential algorithm to be implemented on the distributed memory parallel machine Multi-PSI.

Our parallel algorithm exploits the independence of many sub-processes created through the branching procedure in the sequential algorithm and distributes these processes to different processors (see Figure 8). Scheduling of sub-problems is done by the use of the priority control facility provided from the KL1 language (See[Oki *et al.* 1989]). The incumbent solutions are transferred between processors as global data to be shared so that each processor can update the current incumbent solution as soon as possible.

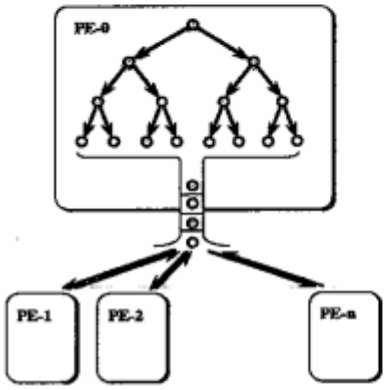

Figure 8: Generation of Parallel Processes

5.3.3 Experimental Results

We implemented the above parallel algorithm in the KL1 language and experimented with the job-shop scheduling problem as an example of mixed-integer problems. Below are the results of computation speedups for a "4 job 3 machine" problem and the total number of searched nodes to get to the solution.

Table 2: Speedup of the Integer Linear Constraint Solver

processors	1	2	4	8
speedup	1.0	1.5	1.9	2.3
number of nodes	242	248	395	490

The above table shows the increase of the number of searched nodes as the number of processors grows. This is for one reason because of the speculative computation inherent in this type of parallel algorithm. Another reason is that the communication latency produces unnecessary computation which could have been avoided if incumbent solutions are communicated instantaneously from the other processor and the unnecessary nodes are pruned.

It is in this way that we get the problem in parallel programming of how to reduce the growth in size of the total search space when multi-processors are used compared with that traversed on one processor using sequential algorithms.

5.4 Hierarchical Constraint Solver

5.4.1 Soft Constraints and Constraint Hierarchies

We have proposed a logical foundation of soft constraints in [Satoh 1990] by using a meta-language which expresses interpretation ordering. The idea of formalizing soft constraints is as follows. Let hard constraints be represented in first-order formulas. Then an interpretation which satisfies all of these first-order formulas can be regarded as a possible solution and soft constraints can be regarded as an order over those interpretations because soft constraints represent criteria applying to possible solutions for choosing the most preferred solutions. We use a meta-language which represents a preference order directly. This meta-language can be translated into a second-order formula to provide a syntactical definition of the most preferred solutions.

Although this framework is rigorous and declarative, it is not computable in general because it is defined by a second-order formula. Therefore, we have to restrict the class of constraints so that these constraints are computable.

Therefore, we introduce the following restriction to make the framework computable.

1. We fix the considered domain so that interpretations of domain-dependent relations are fixed.
2. Soft and hard constraints consist of domain-dependent relations only.

If we accept this restriction, the soft constraints can be expressed in a first-order formula. Moreover, there is a relationship between the above restricted class of soft constraints and hierarchical CLP languages (HCLP languages) [Borning *et al.* 1989, Satoh and Aiba 1990b], as shown in [Satoh and Aiba 1990a].

HCLP language is a language augmenting CLP language with labeled constraints. An HCLP program consists of rules of the form:

$$h \text{ :- } b_1,\ldots,b_n$$

where h is a predicate, and $b_1,\ldots,b_n$ are predicate invocations or constraints or labeled constraints. Labeled constraints are of the form:

`label` C

where C is a constraint in which only domain-dependent functional symbols can be functional symbols and `label` is a label which expresses the strength of the constraint C.

As shown in [Satoh and Aiba 1990a], we can calculate the most preferable solutions by constraint hierarchies

in the HCLP language. Based on this correspondence, we have implemented an algorithm for solving constraint hierarchy on the PSI machine with the following features.

1. There are no redundant calls of the constraint solver for the same combination of constraints since it calculates reduced constraints in a bottom-up manner.
2. If an inconsistent combination of constraints is found by calling the constraint solver, it is registered as a nogood and is used for detecting further contradiction. Any extension of the combination will not be processed so as to avoid unnecessary combinations.
3. Inconsistency is detected without a call of the constraint solver if a processed combination subsumes a registered nogood.

In [Borning *et al.* 1989], Borning et al. give an algorithm for the solving constraint hierarchy. However, it uses backtracking to get an alternative solution and so may redundantly call the constraint solver for the same combination of constraints.

Our implemented language is called CHAL (Contrainte Hierarchiques avec Logique) [Satoh and Aiba 1990b], and is an extension of CAL.

5.4.2 Parallel Solver for Constraint Hierarchies

The algorithms we have implemented on the PSI machine have the following parallelism.

1. Since we construct a consistent constraint set in a bottom-up manner, the check for consistency for each independent constraint set can be done in parallel.
2. We can check if a constraint set is included in nogoods in parallel for each independent constraint set.
3. There is parallelism inside a domain-dependent constraint solver.
4. We can check for answer redundancy in parallel.

Among these parallelisms, the first one is the most coarse and the most suitable for implementation on the Multi-PSI machine. So, we exploit the first parallelism. Then, features of the parallel algorithm become the following.

1. Each processor constructs a maximal consistent constraint set from a given constraint set in a bottom-up manner in parallel. However, once a constraint set is given, there is no distribution of tasks. So, we make idle processors require some task from busy processors and if a busy processor can divide its task, then it sends the task to the idle processor.
2. By pre-evaluation of a parallel algorithm, we found that the nogood subsumption check and the redundancy check have very large overheads. So, we do not check nogood subsumptions and we check redundancy only at the last stage of execution.

Table 3: Performance of Parallel Hierarchical Constraint Solver(unit: sec)

	Processors				
problems	1	2	4	8	16
Tele4	43	32	32	32	29
	1	1.34	1.34	1.34	1.48
5queen	69	39	26	21	19
	1	1.77	2.65	3.29	3.63
6queen	517	264	136	77	50
	1	1.96	3.80	6.71	10.34

Table 3 shows the speedup ration for three examples. Tele4 is to solve ambiguity in natural language phrases. 5queen and 6queen are to solve the 5 queens and 6 queens problem. We represent these problems in Boolean constraints and use the Boolean Buchberger algorithm [Sato and Sakai 1988, Sakai and Aiba 1989] to solve the constraints.

According to Table 3, we obtain 1.34 speedup for Tele4, 3.63 speedup for 5queen, and 10.34 speedup for 6queen. Although 6queen is a large problem for the Boolean Buchberger algorithm and gives us the largest speedup, the speedup saturates at around 16 processors. This expresses that the load is not well-distributed and we have to look for a better load-balancing method in the future.

5.5 Set Constraint Solver

The set constraint solver handles any kind of constraint presented in the following conjunction of predicates.

$$\begin{array}{c} F_1(\bar{x}, \bar{X}) \\ \vdots \\ F_n(\bar{x}, \bar{X}) \end{array}$$

where each predicate $F_i(\bar{x}, \bar{X})$ is a predicate constructed from predicate symbols $\in$, $\subseteq$, $\neq$ and $=$, function symbols $\cap$, $\cup$, and $\cdot^{\sim}$, element variables $\bar{x}$, and set variables $\bar{X}$, and some symbols of constant elements.

For the above constraints, the solver gives the answer of the form:

$$\begin{array}{c} f_1(\bar{x}, \bar{X}) = 0 \\ \vdots \\ f_l(\bar{x}, \bar{X}) = 0 \\ h_1(\bar{x}) = 0 \\ \vdots \\ h_m(\bar{x}) = 0 \end{array}$$

where $h_1(\bar{x}) = 0, \ldots, h_m(\bar{x}) = 0$ give the necessary and sufficient conditions for satisfying the constraints. Moreover, for each solution for the element variables, the system of whole equations instantiated by the solution puts the original constraints into a normal form (i.e. a solution).

For more detailed information on the constraint solver, refer to [Sato *et al.* 1991].

Let us first consider the following example.

$$\begin{aligned} A^{\sim} \cap C^{\sim} \cap E^{\sim} &= \emptyset \\ C \cup E &\supseteq B \\ C \cup E &\supseteq D \\ D \cap B^{\sim} &\supseteq A \\ A^{\sim} \cap B &\subseteq D \\ A \cup B &\supseteq D \end{aligned}$$

where the notation $A^{\sim}$ denotes the complement of A.

Since a class of sets forms a Boolean algebra, this constraint can be considered a Boolean constraint. Hence we can solve this by computing its Boolean Gröbner base:

$$\begin{aligned} D &= A + B \\ E * C &= E + C + 1 \\ A * B &= 0 \end{aligned}$$

We should note that there is neither an element variable nor a constant on elements in the above constraints. Hence they can be expressed as Boolean equations with variables A, B, C, D and E. This, however, does not necessarily hold in every constraint of sets.

Consider the following constraints with an additional three predicates including elements.

$$\begin{aligned} A^{\sim} \cap C^{\sim} \cap E^{\sim} &= \emptyset \\ C \cup E &\supseteq B \\ C \cup E &\supseteq D \\ D \cap B^{\sim} &\supseteq A \\ A^{\sim} \cap B &\subseteq D \\ A \cup B &\supseteq D \\ (C \cap \{x\}) \cup (E \cap \{p\}) &= D \cap \{x, p\} \\ x &\notin A \\ p &\notin B \end{aligned}$$

where x is an element variable and p is a constant symbol of an element.

This can no longer be represented with the Boolean equations as above. For example the last formula is expressed as $\{p\} * B = 0$, where $\{p\}$ is considered a coefficient. In order to handle such general Boolean equations, we extended the notion of Boolean Gröbner bases [Sato *et al.* 1991], which enabled us to implement the set constraint solver.

For the above constraint, the solver gives the following answer:

$$\begin{aligned} D &= A + B \\ E * C &= E + C + 1 \\ A * B &= 0 \\ \{x\} * E * B &= \{x\} * E + \{x\} * B + \{x\} \\ \{p\} * C * A &= \{p\} * C + \{p\} * A + \{p\} \\ \{p\} * E &= \{p\} * A \\ \{x\} * C &= \{x\} * B \\ \{x\} * A &= 0 \\ \{p\} * B &= 0 \\ \{p\} * \{x\} &= 0 \end{aligned}$$

In this example, $\{p\} * \{x\} = 0$ is the satisfiability condition. This holds if and only if $x \neq p$. In this case, there are always A, B, C and D that satisfy the original constraints. The normal form is:

$$\begin{aligned} D &= A + B \\ E * C &= E + C + 1 \\ A * B &= 0 \\ \{x\} * E * B &= \{x\} * E + \{x\} * B + \{x\} \\ \{p\} * C * A &= \{p\} * C + \{p\} * A + \{p\} \\ \{p\} * E &= \{p\} * A \\ \{x\} * C &= \{x\} * B \\ \{x\} * A &= 0 \\ \{p\} * B &= 0 \end{aligned}$$

5.6 Dependency Analysis of Constraint Set

From several experiments on writing application programs, we can conclude that the powerful expressiveness of these languages is a great aid to programming, since all users have to do to describe a program is to define the essential properties of the problem itself. That is, there is no need to describe a method to solve the problem.

On the other hand, sometimes the generality and power of constraint solvers turn out to be a drawback for these languages. That is, in some cases, especially for very powerful constraint solvers like the algebraic constraint solver in CAL or GDCC, it is difficult to implement them efficiently because of their generalities, in spite of great efforts.

As a subsystem of language processors, efficiency in constraint solving is, of course, one of the major issues in the implementation of those language processors [Marriott and Sondergaard 1990, Cohen 1990].

In general, for a certain constraint set, the efficiency of constraint solving is strongly dependent on the order in which constraints are input to a constraint solver. However, in sequential CLP languages like CAL, this order is determined by the position of constraints in a program, because a constraint solver solves constraints accumulated by the inference engine that follows SLD-resolution.

In parallel CLP languages like GDCC, the order of constraints input to a constraint solver is more important than in sequential languages. Since an inference engine and constraint solvers can run in parallel, the order of constraints is not determined by their position in a program. Therefore, the execution time may vary according to the order of constraints input to the constraint solver.

In CAL and GDCC, the computation of a Gröbner base is time-consuming and it is well known that the Buchberger algorithm is doubly exponential in worst-case complexity [Hofmann 1989]. Therefore, it is worthwhile to rearrange the order of constraints to make the constraint solver efficient.

We actually started research into the order of constraints based on dependency analysis [Nagai 1991, Nagai and Hasegawa 1991]. This analysis consisted of dataflow analysis, constraint set collection, dependency analysis on constraint sets, and determination of the ordering of goals and the preference of variables.

To analyze dataflow, we use top-down analysis based on SLD-refutation. For a given goal and a program, the invocation of predicates starts from the goal without invoking a constraint solver, and variable bindings and constraints are collected.

In this analysis, constraints are described in terms of graphical (bipartite graph) representation. An algebraic structure of s set of constraints is extracted using DM decomposition [Dulmage and Mendelsohn 1963], which computes a block upper triangular matrix by canonical reordering a matrix corresponding to the set of constraints.

As a result of analysis, a set of constraints can be partitioned into relatively independent subsets of constraints. These partitions are obtained so that the number of variables shared among different blocks is as small as possible. Besides this partition, shared variables among partitions and shared variables among constraints inside of a block are also obtained. Based on these results, the order of goals and the precedence of variables are determined.

We show the results of this method for two geometric theorem proving problems [Kapur and Mundy 1988, Kutzler 1988]: one is the theorem that three perpendicular bisectors of three edges of a triangle intersect at a point, and the other is the, so-called, nine points circle theorem. The former theorem can be represented by 5 constraints with 8 variables and gives about 3.2 times improvement. The latter theorem can be represented by 9 constraints with 12 variables and gives about 276 times improvement.

6 CAL and GDCC Application Systems

To show the feasibility of CAL and GDCC, we implemented several application systems. In this section, two of these, the handling robot design support system and the Voronoi diagram construction program, are described.

6.1 Handling Robot Design Support System

The design process of a handling robot consists of a fundamental structure design and a internal structure design [Takano 1986]. The fundamental structure design determines the framework of the robot, such as the degree of freedom, number of joints, and arm length. The internal structure design determines the internal details of the robot, such as the mortar torque of each joint. The handling robot design support system mainly supports the fundamental structure design.

Currently, the method to design a handling robot is as follows:

1. First, the type of the robot, such as cartesian manipulator, cylindlical manipulator, or articulated manipulator has to be decided according to the requirements for the robot.
2. Then, a system of equations representing the relation between the end effector and joints is deduced. Then the system of equations is transformed to obtain the desired form of equations.
3. Next, a program to analyze the robot being designed is coded by using an imperative programming language, such as Fortran or C.
4. By executing the program, the design is evaluated. If the result is satisfactory, then the design process terminates, otherwise, the whole process should be repeated until the result satisfies the requirements.

By adopting the CLP paradigm to the design process of a handling robot, through coding a CLP program representing the relation obtained in 2 in the above, the transformation can be done by executing the program. Thus, processes 2 and 3 can be supported by a computer.

6.1.1 Kinematics and Statics Program by Constraint Programming

Robot kinematics represents the relation between a position and the orientation of the end effector, the length of each arm, and the rotation angle of each joint. We call a position and an orientation of the end effector, hand parameters, and we call the rest, joint parameters. Robot statics represent the relation between joint parameters: force working on the end-effector, and torque working on each joint [Tohyama 1989]. These relations are essential for analyzing and evaluating the structure of a handling robot.

To make a program that handles handling robot structures, we have to describe a program independent of its fundamental structure. That is, kinematics and statics programs are constructed to handle any structure of robot by simply changing a query.

Actually, these programs receive a matrix which represents the structure of a handling robot being designed in terms of a list of lists. By manipulating the structure of this argument, any type of handling robot can be handled by the one program.

For example, the following query asks the kinematics of a handling robot with three joints and three arms.

```
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).
```

where the first argument represents the structure of the handling robot, **px**, **py**, and pz represents a position, **ax**, **ay**, **az**, **cx**, cy, and cz represents an orientation by defining two unit vectors which are perpendicular to each other. **sin**'s and cos's represent the rotation angle of each joint, and z3, **x2**, and z1 represent the length of each arm. For this query, the program returns the following answer.

```
cos1^2 = 1-sin1^2
cos2^2 = 1-sin2^2
cos3^2 = 1-sin3^2
    px = -5*cos2*sin3*sin1+z_3*sin2*sin1
         +5*cos3*cos1+x_2*cos1
    py = 5*cos3*sin1+x_2*sin1
         +5*cos1*cos2*sin3-z_3*cos1*sin2
    pz = 5*sin3*sin2+z_1+z_3*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
```

That is, the parameters of the position and the orientation are expressed in terms of the length of each arm and the rotation angle of each joint.

Note that this kinematics program has the full features of the CLP program. The problem of calculating hand parameters from joint parameters is called forward kinematics, and the converse is called inverse kinematics. We can deal with both of them with the same program.

This program can be seen as a generator of programs dealing with any handling robot which has a user designed fundamental structure.

Statics has the same features as the kinematics program described. That is, the program can deal with any type of handling robot by simply changing its query.

6.1.2 Construction of Design Support System

The handling robot design support system should have the following functions

1. to generate the constraint representing kinematics and statics for any type of robot,
2. to solve forward and inverse kinematics,
3. to calculate the torque which works on each joint, and
4. to evaluate manipulability.

The handling robot design support system consists of the following three GDCC programs in order to realize these functions,

Kinematics a kinematics program

Statics a statics program

Determinant a program to calculate the determinant of a matrix

Kinematics and **Statics** are the programs we described above. A matrix to evaluate the manipulability of a handling robot, called a Jacobian matrix, is obtained from the **Statics** program. **Determinant** is used to calculate the determinant of a Jacobian matrix. This determinant is called the manipulability measure and it expresses the manipulability of the robot quantitatively [Yoshikawa 1984].

To obtain concrete answers, of course, the system should utilize the GDCC ability to approximate the real roots of univariate equations.

6.2 Constructing the Voronoi Diagram

We developed an application program which constructs *Voronoi Diagram* written in GDCC.

By using the constraint paradigm, we can make a program without describing a complicated algorithm. A Voronoi diagram can be constructed by using constraints which describe only the properties or the definition of the Voronoi diagram. This program can compute the Voronoi polygon of each point in parallel.

6.2.1 Definition of the Voronoi Diagram

For a given finite set of points S in a plane, a Voronoi diagram is a partition of the plane so that each region of the partition is a set of points in the plane closer to a point in S in the region than to any other points in S [Preparata and Shamos 1985].

In the simplest case, the distance between two points is defined as the Euclidian distance. In this case, a Voronoi diagram is defined as follows.

Given a set S of N points in the plane, for each point P_i in S, the *Voronoi polygon* denoted as $V(P_i)$ is defined by the following formula.

$$V(P_i) = \{P \mid d(P, P_i) < d(P, P_j), \forall j \neq i\}$$

where $d(P, P_i)$ is a Euclidian distance between P and P_i.

The Voronoi diagram is a partition so that each region is the Voronoi polygon of each point (see Figure 9). The vertices of the diagram are *Voronoi vertices* and its line segments are *Voronoi edges*.

Voronoi diagrams are widely used in various application areas, such as physics, ecology and urbanology.

6.2.2 Detailed Design

The methods of constructing Voronoi diagrams are classified into the following two categories:

1. The incremental method([Green and Sibson 1978]), and

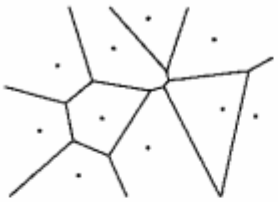

Figure 9: A Voronoi Diagram

2. The divide-and-conquer method([Shamos and Hoey 1975]).

However, the simplest approach to constructing a Voronoi diagram is, of course, constructing its polygons one at a time.

Given two points, P_i and P_j, a set of points closer to P_i than to P_j is just a half-plane containing P_i that is divided by the perpendicular bisector of $\overline{P_iP_j}$. We name this line $H(P_i, P_j)$.

The Voronoi polygon of P_i can be obtained by the following formula.

$$V(P_i) = \cap_{j \neq i} H(P_i, P_j) \ . \ \text{p}$$

By using the linear constraint solver for GDCC, the Voronoi polygon can be constructed by the following algorithm which utilizes the above method to obtain the polygon directly.

```
E_s ← {x ≥ 0, y ≥ 0, x ≤ x_Max, y ≤ y_Max}
{loop a}
for i = 1 to n
    CF_0 ← linear_constraint_solver(E_s)
    for j = 1 to n
        if(j ≠ i) then
          E_j ← y ≤ (P_jx − P_ix)/(P_jy − P_iy) · x
                  +(P_jy^2 + P_ix^2 − P_iy^2 − P_jx^2)/2 · (P_jy − P_iy)^2
          CF_j ← linear_constraint_solver(E_j ∪ CF_{j−1})
          Let {eq_1, eq_2, ..., eq_k}(0 ≤ k ≤ n) be
          a set of equations obtained by changing
          inequality symbols in CF_j to equation symbols.
    {loop b}
    for l = 1 to k
        vertices := {}
        m := 1
        while (m =< k &
                number of elements of vertices ≠ 2)
                pp ← intersection(eq_l, eq_m)
                if pp satisfies the constraint set CF_i
                  then vertices := {pp} ∪ vertices
                m := m + 1
        add the line segment between vertices
                to Voronoi edges.
end.
```

In this algorithm, the first half computes the Voronoi polygon for each point's P_i by obtaining all perpendicular bisectors of segments between P_i and other points and eliminating redundant ones. The second half computes the Voronoi edges.

[2]This inequality represents a half plane divided by a perpendicular bisector of $\overline{(P_i, P_j)}$

Table 4: Runtime and reductions

Points	Processors 1	2	4	8	15	Reductions (×1000)
10	130	67	33	17	16	5804
	1	1.936	3.944	7.377	7.844	
20	890	447	241	123	88	42460
	1	1.990	3.685	7.218	10.077	
50	4391	2187	1102	566	336	210490
	1	2.007	3.981	7.749	13.065	
100	17287	8578	4305	2191	1263	830500
	1	2.015	4.014	7.887	13.679	
200	52360	26095	13028	6506	3500	2458420
	1	2.006	4.018	8.047	14.959	
400	220794	110208	54543	27316	14819	10161530
	1	2.003	4.048	8.082	14.899	

To realize the above algorithm on parallel processors, each procedure for each i in loop **a** in the above is assigned to a group of processes. That is, there are n process groups. Each procedure for each l in the loop **b** is assigned to a process in the same process group. This means that each process group contain k processes. These $n \times k$ processes are mapped onto multi-processor machines.

6.2.3 Results

Table 4 shows the execution time and speedup for 10 to 400 points with 1 to 15 processors.

According to the results, we can conclude that, when the number of points is large enough, we can obtain efficiency which is almost in proportion to the number of processors.

By using this algorithm, we can handle the problem of constructing a Voronoi diagram in a very straight forward manner. Actually, comparing the size of the programs, this algorithm can be described in almost one third of the size of the program that is used by the incremental method.

7 Conclusion

In the FGCS project, we developed two CLP languages: CAL, and GDCC to establish the knowledge programming environment, and to write application programs. The aim of our research is to construct a powerful high-level programming language which is suitable for knowledge processing. It is well known that constraints play an important role in both knowledge representation and knowledge processing. That is, CLP is a promising candidate as a high level programming language in this field.

Compared with other CLP languages such as CLP($\mathcal{R}$), Prolog III, and CHIP, we can summarize the features of CAL and GDCC as follows:

- CAL and GDCC can deal with nonlinear algebraic constraints.

- In the algebraic constraint solver, the approximate values of all possible real solutions can be computed, if there are only finite number of solutions.
- CAL and GDCC have a multiple environment handler. Thus, even if there is more than one answer constraints, users can manipulate them flexibly.
- Users can use multiple constraint solvers, and furthermore, users can define and implement their own constraint solvers.

CAL and GDCC enable us to write possibly nonlinear polynomial equations on complex numbers, relations on truth values, relations on sets and their elements, and linear equations and linear inequalities on real numbers.

Since starting to write application programs for the algebraic constraint solver in the field of handling robot, we have wanted to compute the real roots of univariate nonlinear polynomials. We made this possible with CAL by adding a function to approximate the real roots, and we modified the Buchberger algorithm able to handle approximation values.

Then, we faced the problem that a variable may have more than one value. To handle this situation in the framework of logic programming, we introduced a context tree in CAL. In GDCC, we introduced blocks into the language specification. The block in GDCC not only handle multiple values, but also localize the failure of constraint solvers.

As for CAL, the following issues are still to be considered:

1. Meta facilities:
   Users cannot deal with a context tree from a program, that is, meta facilities in CAL are insufficient to allow users to do all possible handling of answer constraints themselves.
2. Partial evaluation of CAL programs:
   Although we try to analyze constraint sets by adopting dependency analysis, that work will be more effective when combined with partial evaluation technology or abstract interpretation.
3. More application programs:
   We still have a few application programs in CAL. By writing many application programs in various application field, we will have ideas to realize a more powerful CLP language. For this purpose, we are now implementing CAL in a dialect of ESP, called Common ESP, which can run on the UNIX operating system to be able to use CAL in various machines.

As for GDCC, the following issues are still to be considered:

1. Handling multiple contexts:
   Although current GDCC has functionalities to handle multiple contexts, users have to express everything explicitly. Therefore, we can design high-level tools to handle multiple contexts in GDCC's language specification.
2. More efficient constraint solvers:
   We need to improve both the absolute performance and the parallel speedup of the constraint solvers.
3. More application programs:
   Since parallel CLP language is quite new language, writing application programs may help us to make it powerful and efficient.

Considering our experiences of using CAL and GDCC and the above issues, we will refine the specification and the implementation of GDCC.

These refinements and experiments on various application programs clarified the need for a sufficiently efficient constraint logic programming system with high functionalities in the language facilities.

Acknowledgment

The research on the constraint logic programming system was carried out by researchers in the fourth research laboratory in ICOT in tight cooperation with cooperating manufactures and members of the CLP working group. Our gratitude is first due to those who have given continuous encouragement and helpful comments. Above all, we particularly thank Dr. K. Fuchi, the director of the ICOT research center, Dr. K. Furukawa, a vice director of the ICOT research center, and Dr. M. Amamiya of Kyushu University, the chairperson of the CLP working group.

We would also like to thank a number of researchers we contacted outside of ICOT, in particular, members of the working group for their fruitful and enlightening discussions and comments.

Special thanks go to all researchers in the fourth research laboratory: Dr. K. Sakai, Mr. T. Kawagishi, Mr. K. Satoh, Mr. S. Sato, Dr. Y. Sato, Mr. N. Iwayama, Mr. D. J. Hawley, who is now working at Compuflex Japan, Mr. H. Sawada, Mr. S. Terasaki, Mr. S. Menju, and the many researchers in the cooperating manufacturers.

References

[Aiba *et al.* 1988] 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.

[Backelin and Fröberg 1991] J. Backelin and R. Fröberg. How we proved that there are exactly 924 cyclic 7-roots. In S. M. Watt, editor. *Proceedings of ISSAC'91*. ACM, July 1991.

[Benichou *et al.* 1971] M. Benichou. L. M. Gauthier. P. Girodet. G. Hentges. G. Ribiere. and O. Vincent. Experiments in mixed-integer linear programming. *Mathematical Programming*. (1). 1971.

[Boege *et al.* 1986] W. Boege, R. Gebauer. and H. Kredel. Some Examples for Solving Systems of Algebraic Equations by Calculating Gröbner Bases. *Journal of Symbolic Computation*, 2(1):83–98, 1986.

[Borning *et al.* 1989] A. Borning, M. Maher, A. Martindale, and M. Wilson. Constraint Hierarchies and Logic Programming. In *Proceedings of the International Conference on Logic Programming*, 1989.

[Buchberger 1985] B. Buchberger. Gröbner bases: An Algorithmic Method in Polynomial Ideal Theory. In N. Bose, editor, *Multidimensional Systems Theory*, pages 184–232. D. Reidel Publ. Comp., Dordrecht. 1985.

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

[Chikayama 1984] T. Chikayama. Unique Features of ESP. In *Proceedings of FGCS'84*, pages 292–298. 1984.

[Chikayama *et al.* 1988] T. Chikayama, H. Sato, and T. Miyazaki. Overview of Parallel Inference Machine Operationg System (PIMOS). In *International Conference on Fifth Generation Computer Systems*, pages 230–251, 1988.

[Clarke *et al.* 1990] E. M. Clarke, D. E. Long, S. Michaylov, S. A. Schwab, J. P. Vidal, and S. Kimura. Parallel Symbolic Computation Algorithms. Technical Report CMU-CS-90-182, Computer Science Department, Carnegie Mellon University, October 1990.

[Cohen 1990] J. Cohen. Constraint logic programming languages. *Communications of the ACM*, 33(7), July 1990.

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

[Dincbas *et al.* 1988] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, November 1988.

[Dulmage and Mendelsohn 1963] A. L. Dulmage and N. S. Mendelsohn. Two algorithms for bipartite graphs. *Journal of SIAM*, 11(1), March 1963.

[Green and Sibson 1978] P. J. Green and R. Sibson. Computing Dirichlet Tessellation in the Plane. *The Computer Journal*. 21. 1978.

[Hofmann 1989] C. M. Hoffmann. *Gröbner Bases Techniques*, chapter 7. Morgan Kaufmann Publishers, Inc., 1989.

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

[Kapur and Mundy 1988] K. Kapur and J. L. Mundy. Special volume on geometric reasoning. *Artificial Intelligence*, 37(1-3), December 1988.

[Kutzler 1988] B. Kutzler. *Algebraic Approaches to Automated Geometry Theorem Proving.* PhD thesis, Research Institute for Symbolic Computation , Johannes Kepler University, 1988.

[Lloyd 1984] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[Maher 1987] M. J. Maher. Logic Semantics for a Class of Committed-choice Programs. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 858–876, Melbourne, May 1987.

[Marriott and Sondergaard 1990] K. Marriott and H. Sondergaard. Analysis of constraint logic programs. In *Proc. of NACLP '90*, 1990.

[Menju *et al.* 1991] 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.

[Nagai 1991] Y. Nagai. Improvement of geometric theorem proving using dependency analysis of algebraic constraint (in Japanese). In *Proceedings of the 42nd Annual Conference of Information Processing Society of Japan*, 1991.

[Nagai and Hasegawa 1991] Y. Nagai and R. Hasegawa. Structural analysis of the set of constraints for constraint logic programs. Technical report TR-701, ICOT, Tokyo, Japan, 1991.

[Oki *et al.* 1989] H. Oki, K. Taki, S. Sei, and S. Furuichi. Implementation and evaluation of parallel Tsumego program on the Multi-PSI (in Japanese). In *Proceedings of the Joint Parallel Processing Symposium (JSPP'89)*, 1989.

[Ponder 1990] C. G. Ponder. Evaluation of 'Performance Enhancements' in algebraic manipulation systems. In J. Della Dora and J. Fitch, editors, *Computer Algebra and Parallelism*, pages 51–74. Academic Press, 1990.

[Preparata and Shamos 1985] F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag, 1985.

[Sakai and Aiba 1989] K. Sakai and A. Aiba. CAL: A Theoretical Background of Constraint Logic Programming and its Application. *Journal of Symbolic Computation*, 8:589–603, 1989.

[Saraswat 1989] V. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, Computer Science Department. January 1989.

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

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

[Sato *et al.* 1991] 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.

[Satoh 1990] K. Satoh. Formalizing Soft Constraints by Interpretation Ordering. In *Proceedings of 9th European Conference on Artificial Intelligence*, pages 585–590, 1990.

[Satoh and Aiba 1990a] K. Satoh and A. Aiba. Computing Soft Constraints by Hierarchical Constraint Logic Programming. Technical Report TR-610. ICOT, Tokyo, Japan, 1990.

[Satoh and Aiba 1990b] K. Satoh and A. Aiba. Hierarchical Constraint Logic Language: CHAL. Technical Report TR-592, ICOT, Tokyo, Japan, 1990.

[Senechaud 1990] P. Senechaud. Implementation of a parallel algorithm to compute a Gröbner basis on Boolean polynomials. In J. Della Dora and J. Fitch, editors, *Computer Algebra and Parallelism*, pages 159–166. Academic Press, 1990.

[Shamos and Hoey 1975] M. I. Shamos and D. Hoey. Closest-point problems. In *Sixteenth Annual IEEE Symposium on Foundations of Computer Science*, 1975.

[Siegl 1990] K. Siegl. Gröbner Bases Computation in STRAND: A Case Study for Concurrent Symbolic Computation in Logic Programming Languages. Master's thesis, CAMP-LINZ, November 1990.

[Takano 1986] M. Takano. Design of robot structure (in Japanese). *Journal of Japan Robot Society*, 14(4), 1986.

[Terasaki *et al.* 1992] S. Terasaki, D. J. Hawley, H. Sawada, K. Satoh, S. Menju, T. Kawagishi, N. Iwayama, and A. Aiba. Parallel Constraint Logic Programming Language GDCC and its Parallel Constarint Solvers. In *International Conference on Fifth Generation Computer Systems*, 1992.

[Tohyama 1989] S. Tohyama. *Robotics for Machine Engineer (in Japanese)*. Sougou Denshi Publishing Corporation, 1989.

[Ueda and Chikayama 1990] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, 33(6), December 1990.

[Vidal 1990] J. P. Vidal. The Computation of Gröbner bases on a shared memory multi-processor. Technical Report CMU-CS-90-163, Computer Science Department, Carnegie Mellon University, August 1990.

[van Emden and Kowalski 1976] 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.

[Van Hentenryck 1989] P. Van Hentenryck. Parallel constraint satisfaction in logic programming: Preliminary results of chip with pepsys. In *6th International Conference on Logic Programming*, pages 165–180, 1989.

[Yoshikawa 1984] T. Yoshikawa. Measure of manipulatability of robot arm (in Japanese). *Journal of Japan Robot Society*, 12(1), 1984.