Efficient Solution of Constraint Satisfaction Problems by Equivalent Transformation

Hiroshi Mabuchi[1]

[1] Faculty of Software and Information Science, Iwate Prefectural University, Takizawa, Iwate, Japan

ABSTRACT

In many conventional programming languages, programs are constructed from built-in data structures and built-in constraints. Due to the limitation of the expressive power, the computation may become very inefficient when they are used to solve large and complicated problems. In order to overcome the difficulty, we adopt a computing framework of "problem solving based on equivalent transformation (ET)." In this framework, a problem can be solved by simplifying declarative descriptions of the problem on the equivalent transformation basis, and the correctness of computation can be guaranteed. Furthermore, users can easily introduce new various data structures and constraint-solving rules necessary for the improving of the constraint-solving algorithms. In this paper, we show that "problem solving based on equivalent transformation" is useful to avoid the combinatorial explosion when solving large constraint satisfaction problems.

KEYWORDS: *Combinatorial explosion, Constraint satisfaction problem, Equivalent transformation, Transformation rule*

I. INTRODUCTION

In constraint logic programming languages [3,5,9,10,23], a system has constraint atoms and algorithms for solving them. The “built-in” approach of combining constraint atoms enables users to program. This approach offers an advantage of guaranteeing the correctness of a solving method, because a system provides a constraint-solving algorithm. If a system has constraint atoms which are efficient for problem solving, users can create efficient programs easily.Constraint logic programming languages, however, have also disadvantages. Due to dependency on a pre-supported constraint-solving algorithm, expressive power for programming is limited. If its limit is violated, the computation efficiency may drastically decrease. In such a case, the “built-in” approach can not improve the constraint-solving algorithm. It is desirable for users to be able easily to add to or improve data structures or rules which represent constraints. The "built-in" approach prevents free programming which efficiently enables users to solve problems.

As a result, it may cause crucial problems when large-scale and complicated problems are solved.This paper proposes to solve problems by the “expansive” approach, where without depending only on a constraint-solving algorithm possessed in a system, users can improve the algorithm and define new rules for constraint-solving.First, this paper provides an example of unsolvable problems: an attempt to solve a problem under a constraint-processing algorithm possessed in systems of constraint logic programming languages may cause combinatorial explosion. Next, this paper demonstrates that a better result can be obtained by improving constraint atoms and algorithms for solving them so as to carry out constraint processing suitable for the problem. The “expansive” approach enables users to define efficient constraint atoms and constraint-solving algorithms for a given problem, and consequently the approach may derive an efficient solving method.Constraint Handling Rules (CHR) [8] is a kind of expansion of Constraint Logic Programming (CLP). In CLP, rules to solve constraints are built in a system, but in CHR, users can define some rules to solve constraints, the correctness of which is assured based on logical inference. Making use of user-defined rules to solve constraints presents difficulties in assuring to the correctness of solving method. In the "expansive” approach, users must guarantee the correctness of a solving method; therefore, such theoretical foundation (computation model) that users can easily guarantee the correctness of the solving method is required.In this study, the equivalent transformation (ET) computation model [1,14] is used to solve constraint satisfaction problems (CSPs) of Number-Place Problems. In CHR, users can define some rules to solve constraints, but rules in CHR constitute only a part of rules in the ET computation model [2]. In the ET computation model, rules

which can not be described by formulas can be defined as well as rules dealt with in CHR. For example, basic transformation rules for equality constraint described in [13] can not be dealt with in CHR.Solutions to CSPs can be classified into systematic search algorithms, represented by search based on backtracking, and stochastic search algorithms, represented by hill-climbing method. Systematic search algorithms guarantees the completeness of the algorithm but is not suitable for solving large-scale CSPs [4,7,15,21,24]. On the other hand, in lieu of guaranteeing the completeness of the algorithm, stochastic search algorithms can deliver practical approximate solutions at high-speed. However, its problem is that it falls into local optima while searching for solutions [6,16,19,20].

In this paper, the computing framework of "Problem Solving based on ET," which effectively performs computations while preserving the algorithm's correctness In terms of problems in obtaining a solution set, the completeness of algorithms defined by conventional studies on CSP solving is that all solutions can be obtained. If an algorithm is complete and sound (obtained solutions are always correct), then that algorithm is correct1, is utilized to solve CSPs and the effectiveness of the proposed method is demonstrated [17,18]. In this computing framework, a problem can be solved by simplifying declarative descriptions of the problem using ET, and the correctness of computation can be guaranteed over a broader range than the framework of logic paradigm.In the computing framework for problem solving based on ET, a problem is successively simplified into different problems by selecting and applying an appropriate rule from many ET rules; by showing that each transformation rule causes the equivalent transformation, computation can be guaranteed without changing the meaning of the given problem. Furthermore, this computing framework allows an easy introduction of the new data structures and rules necessary for improving the constraint-solving algorithms.

II. COMPUTATION BY EQUIVALENT TRANSFORMATION

This study adopts, for problem solving, the computation model called "equivalent transformation", where computation is regarded as equivalent transformation of declarative descriptions. This section describes its outline.

2.1. Equivalent Ttransformation of Declarative Descriptions

In our approach, a problem is formalized by a declarative description, which is a set of extended definite clauses, where we can treat various data structures including multisets, strings, and constraints, as well as usual terms [12].A declarative description consists of the union of the definition part D and the query part Q. Given declarative description D∨Q of a problem, query part Q is said to be transformed correctly in one step into new query part Q' by an application of a rewriting rule, iff declarative descriptions D∨Q and D∨Q' are equivalent, i.e., they have the same meaning. A rewriting rule is considered to be correct, iff its application always results in correct transformation. A correct rewriting rule is referred to as an equivalent Transformation rule (ET rule).

2.2. Problem Solving Based on Equivalent Transformation

In problem solving based on equivalent transformation, a declarative description is successively simplified into different declarative descriptions by ET rules, and from the simplified declarative description the solution may be obtained. If ET rules are employed in all transformation steps, the answer is guaranteed to be correct.

Rules used for transformations should be only those which transform a declarative description correctly. The rules in which the conditions to apply are true can be repeatedly used in any order. The system selects a rule to apply depending on computational situations. Each rule should include

- conditions that decide the applicability of the rule.
- definitions that determine a new set of clauses.

2.3. Variety, Correctness and Confluence of Computation

ET approach has the following properties.

[Variety of computation]

This approach can use not only unfolding rules [22] but also other various ET rules as transformation rules. By nondeterministic selection of ET rules at each step of computation, a variety of computation becomes possible.

[Correctness of computation]

Strict correctness is guaranteed in problem solving based on equivalent transformation. The correctness of rules can be determined without considering interrelations with other rules. As long as (correct) ET rules are applied, no matter what the rules are and in what order they are applied, correctness of the result of computation can be assured.

[Confluence of computation]
Meaning (which is defined in [12] by the name of declarative semantics) of a declarative description at each step of computation is preserved by an ET rule applied. Therefore, the confluence of solutions of a problem is achieved. It is obvious that the confluence of declarative descriptions is unnecessary for correct computation.

2.4. Advantages of ET Approach

ET approach has the advantages of

- describing various expressive rules since it offers abundant data structures.
- controlling processing flexibly since the order of computation is not fixed.
- improving algorithms at a lower cost by the introduction and the deletion of rules.

Furthermore, this paper shows that efficient constraint processing is possible since users are allowed to define constraint atoms without depending only on the "built-in" constraint atoms.

III. PROBLEM AND ITS FORMALIZATION

This section defines constraint satisfaction problem, explains number-place problem as an example, and formalizes the problem in terms of declarative descriptions.

3.1. Constraint Satisfaction Problem and Its Example

Constraint satisfaction problem (CSP) [24,25] is generally defined by the following three sets: I, a set of n-variables (X1, X2,… , Xn); domain (D1, D2, … ,Dn), a set of the values which variables could take; and C : {Ci(… Xj …)}, a constraint which should be satisfied by the values of the variables. The objective of CSP is assignment of values to all variables in such a way that it satisfies all constraints in a given problem. In this study, number-place problems are used as examples of constraint satisfaction problems. Number-place is a puzzle in which numbers from 1 to 9 are placed in each small blank square (Fig. 1). There are two constraints:
(Constraint 1)
The numbers 1 through 9 will be placed into each small blank square.
(Constraint 2)
The same number can not be placed in any one column or row, nor within any one medium-sized box surrounded by a thicker border.

	6		2		4		5	
4	7			6			8	3
		5		7		1		
9			1		3			2
	1	2						9
6			7		9			8
		6		8		7		
1	4			9			2	5
	8		3		5		9	

Fig. 1 Number-place problem

3.2. Declarative Descriptions Representing Problems

This section formalizes problems in terms of declarative descriptions. Constraints to be satisfied when solving a numberplace problem are to satisfy a given assignment of the problem (given assignment predicate) and to adhere to the constraints of the problem (NP constraints predicate). These are represented with the following clause. The syntax of declarative descriptions is expressed by S-expressions. Symbols starting with "*" represent variables.

(answer *numberplace) ←
(given_assignment *numberplace),
(NP_constraints *numberplace).

"given assignment" predicate is defined as follows. The "?" marks represent anonymous variables, each of which is different from all others.

(given_assignment *numberplace)←

```
(= *numberplace ( (? 6 ? 2 ? 4 ? 5 ?)
(4 7 ? ? 6 ? ? 8 ?)
(? ? 5 ? 7 ? 1 ? ?)
(9 ? ? 1 ? 3 ? ? 2)
(? 1 2 ? ? ? ? ? 9)
(6 ? ? 7 ? 9 ? ? 8)
(? ? 6 ? 8 ? 7 ? ?)
(1 4 ? ? 9 ? ? ? 2 5)
(? 8 ? 3 ? 5 ? 9 ?))).
```

A variable, *numberplace, is equal to the list which represents the given assignment of a problem. "NP constraints" predicate is expressed in accordance with (Constraint 1) and (Constraint 2) in Section 3.1. Fig. 2 shows the predicates which define the "NP constraints" predicate.

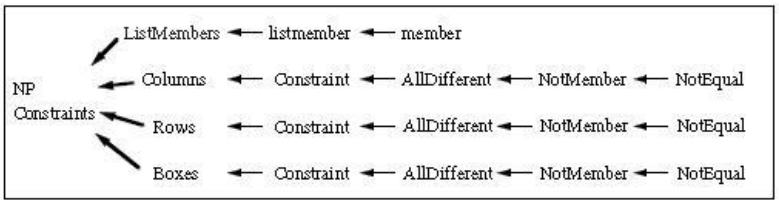

Fig. 2 Predicates which define "NP constraints" predicate

Member atom representing Constraint 1, providing that a variable is any of the numbers 1 through 9, is represented as follows:
(member *a (1 2 3 4 5 6 7 8 9))
AllDifferent atom representing Constraint 2, providing that elements of the list are different from each other, is represented as follows:
(AllDifferent (*a 6 *b 2 *c 4 *d 5 *e))
AllDifferent predicate is defined with Notmember predicate providing that an element does not belong to a given list. Notmember predicate is defined with NotEqual predicate providing that two arguments are not equal.

IV. RULES FOR SIMPLE CONSTRAINT ATOMS

In the problem solving based on equivalent transformation, various rules that are defined based on the declarative descriptions are applied. The problem can be successfully solved when all atoms in a clause are eliminated.In the problem shown in Section 3.2, first the answer clause is transformed and through various transformations, the NP constraints atom is transformed only into member atoms and NotEqual atoms. This section defines rules for these atoms.

4.1. Rules for Member Atoms

In the previous section, it was shown that the NP constraints atom is transformed into only member atoms and NotEqual atoms. It is hard to transform the clause further without increasing the number of clauses if we pay attention only to one atom. We think the following example using two atoms.

```
(member *a (1 2 3 4 5 6 7 8 9))
(NotEqual *a 6)
```

In this case, since NotEqual atom shows that the variable *a is not 6, 6 can be eliminated from the list (the second argument) of the member atom.However, the cost of such transformation is rather high because a computational cost of the order in the square of the number of atoms in order to find the two atoms, which are to be transformed, is required; therefore, it would be very efficient if the information of one atom could be spread to other atoms.

This study introduces a new data structure called an i-var [11]. An i-var is defined as a variable which has been given information. An i-var has the form in which a variable is followed by a symbol, "˜", and ends with S-expressions such as "apple" or "(1 2 3)", as with ? ˜apple or *x˜(1 2 3). Then S-expressions are called informations and the whole variable is called an i-var. A variable followed by nothing is called a pure variable. Users can freely get, replace, or eliminate information from i-var. Also, users can freely define the meaning of the information which the variable has.In the two atoms shown above, the member atom is transformed and the

variable *a is changed to an i-var in which the variable *a must be one of the numbers from 1 to 9. The change from a pure variable to an i-var instantly spreads over the entire clause. As a result, the NotEqual atom mentioned above is also changed into

(NotEqual *a˜(1 2 3 4 5 6 7 8 9) 6).

All member atoms are transformed and all variables are changed to i-vars. Therefore, only NotEqual atoms remain in the body of the clause.

4.2. Rules for NotEqual Atoms

NotEqual atom is a constraint such that two arguments are not equal. The candidate elimination means to eliminate elements (candidates) that do not obviously satisfy a constraint from elements (candidates) in the information of an i-var.The method for eliminating unnecessary numbers from candidates consists of the following procedure: when one argument in an atom is already decided as a number, the other variable in the atom can not be the same number, so the number can be eliminated from the candidates. When numbers in the information of a variable are reduced to only one number, the variable is unified with the number. Some examples of the rules are shown below.

- "candidate elimination" rule

When one argument is a number and the other argument is an i-var, the number is eliminated from the candidates in the i-var.

(NotEqual *a˜(1 2 3 4) 4)
In such an atom, since the variable *a can not be equal to the number 4, 4 is eliminated from the candidates, and this atom can be eliminated. The change in the information instantly spreads over the entire clause, and the atoms that have the variable *a change.
*a˜(1 2 3 4) → *a˜(1 2 3)

- "unification" rule

When either i-var reduces numbers in the information to one number, the variable is unified with the number, and the number is eliminated from the candidates in the other i-vars.
(NotEqual *a˜(1 2 3 4) *b˜(4))
In this case, since the numbers of the variable *b are reduced to only one number, *b is unified with the number, 4. As a result, since it is clear that the variable *a is not 4, 4 is eliminated from the candidates in the variable *a, and this atom can be eliminated.
*b → 4, *a˜(1 2 3 4) → *a˜(1 2 3)

Another rule is the "number check" rule, such that when both arguments in a NotEqual atom are numbers, checks are made to determine whether the numbers are different and then, when possible, the atom is eliminated. This rule also eliminates the entire clause when the elements consist of the same number.

4.3. "Splitting" Rule

There may remain atoms in which candidates can not be reduced only using the rules described in the preceding section. This is because since there are nine possible values for one variable, both variables in the atom may be variables whose candidates are two or more. An example of such a NotEqual atom is shown as follows.
(NotEqual *a˜(3 5 7 9) *b˜(5 9))

Since the candidates of the variable *b is reduced to two numbers, the "splitting" rule is employed, according to which a clause is separated into two clauses. When the "splitting" rule is applied, the clause is branched into two clauses: one clause of the variable *b is unified with 5 and the other clause of the variable *b is unified with 9. Computation proceeds in each clause and the clause that causes any contradiction is eliminated. However, computation efficiency usually decreases when the branching of a clause occurs. Therefore, less priority should be given to the "splitting" rule, in order to avoid the branching of a clause when possible.

V. EXPLOSION OF PROCESSING TIME

The results obtained by solving problems by using the rules described in the previous section are shown as follows. The problem is number-place problem of size 25 × 25, i.e., 25 rows and 25 columns. The number of blank squares is the number of variables. The more the number of variables is, the greater amount of the computation is required.The horizontal axis in the graph in Fig. 3 shows the number of blank squares, i.e.,

the number of variables. 0 indicates that values are assigned to all variables. This graph shows the amount of processing time. The vertical axis shows the processing time required to obtain answers. From Fig. 3, the processing time does not change up to about 200 variables but the processing time drastically increases when the number of variables surpasses that extent: about 5 seconds for 225 variables, about 3.5 minutes for 250 variables, about 17 minutes for 275 variables, and about 100 minutes for 286 variables, and the problem with 300 or more variables can not be solved. This is because a clause has been separated into many clauses and a combinatorial explosion has occurred.

The next section introduces a new constraint processing to solve such a difficulty.

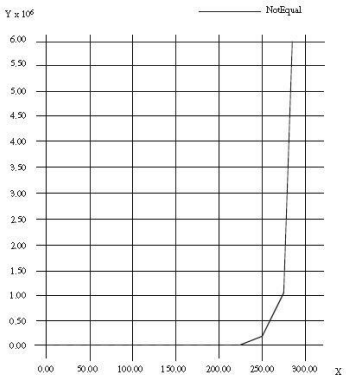

Fig. 3 Results of 25 × 25 puzzles

VI. INTRODUCTION OF NEW RULES FOR ALLDIFFERENT ATOMS

This section improves the constraint-solving algorithms by introducing new constraints atoms to solve problems more efficiently.

6.1. Changing of Constraint Atoms

As described in Section 3.2, an AllDifferent atom is transformed into NotMember atoms. A NotMember atom is transformed into NotEqual atoms. That is, the candidate elimination is performed using constraints of NotEqual atom which is the simplest in the atoms. This is because constraints of NotEqual atom are the simplest and easiest to deal with. We do not transform AllDifferent atom into NotMember atoms, but instead we regard AllDifferent atom as the smallest unit and try to perform the candidate elimination, i.e., we deal with more complicated atoms. The reason is that since information in AllDifferent atom is more than that in NotMember atom, the candidate elimination can be performed by good use of those information (Fig. 4).

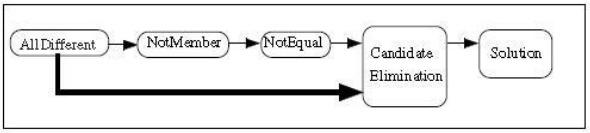

Fig. 4 Removing NotMember and NotEqual atoms

The next section defines the rules to perform constraint processing using constraints of AllDifferent atom where all elements in the list are different.

6.2. Rules for AllDifferent Atoms

This section defines the candidate elimination rules for AllDifferent atoms.

- "candidate elimination" rule

Elements that are identical to the numbers are all eliminated from the candidates of other i-vars.

(AllDifferent (3 *a˜(2 3 4) 5 *b˜(2 3 4 5)))

↓

(AllDifferent (*a˜(2 4) *b˜(2 4)))

- "unification" rule

When an i-var reduces candidates to one number, the variable is unified with the number.

(AllDifferent (*a˜(2 3 4) *b˜(2 3 4) *c˜(3)))

↓

(AllDifferent (*a˜(2 3 4) *b˜(2 3 4) 3))

- "number decision" rule

(AllDifferent (*a˜(5 7 9) *b˜(5 9) *c˜(4 5 9) *d˜(4 5))

In this example, variable *a can be determined as 7. The reason is as follows. The candidates of *b, *c or *d are contained either in any two or in all three of 4,5,9. The number of candidates is three of 4,5,9 and the number of variables is also three of *b, *c, *d. Then, the relation between the variables and the candidates (numbers) should be one-to-one. Therefore, three numbers of 4,5,9 are used as the values of *b, *c, *d. *a can be determined as 7 by removing 4,5,9 from the candidates of *a.

VII. COMPARISON AND CONSIDERATION

This section offers a comparison between results obtained through constraint processing using constraints of NotEqual atoms and those using constraints of AllDifferent atoms.

7.1. A Comparison in The Case of The 9 × 9 Puzzle

Table 1 shows a comparison between the results obtained by applying the candidate elimination rule to NotEqual atoms and those obtained by applying the candidate elimination rule to AllDifferent atoms for the same problem. In the processing time and the number of rule applications, the respective values of AllDifferent atoms when each value of NotEqual atoms equals 1 are shown.

Table 1 Results of the 9 × 9 Puzzle (Fig.1)

Atom	NotEqual	AllDifferent
Processing Time	**1**	0.37
Number of Rule Applications	**1**	0.01

The comparison above demonstrates that applying the candidate elimination rule to AllDifferent atoms enables the user to obtain better results on the processing time and the number of rule applications.

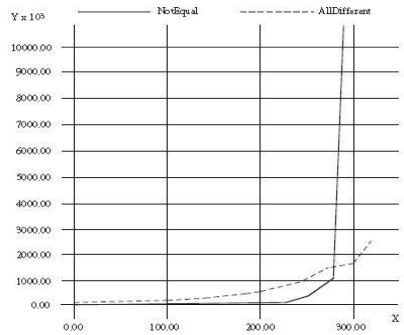

Fig. 5 Comparison of results of 25 × 25 puzzles

7.2. A Comparison in The Case of 25 × 25 Puzzles

Fig. 5 shows the results obtained by solving 25 × 25 puzzles (problems), which are discussed in Section V, through applying the candidate elimination rule to AllDifferent atoms. When the number of variables is 275 or less, the processing time is shortened by transforming the NP constraints atom to NotEqual atoms for constraint processing. When the number of variables is higher than that, the processing time required explodes, and consequently it is impossible to obtain a solution for this problem with 325 variables. However, in the case of constraint processing by using constraints of AllDifferent atom, even if the number of variables increases, we are able to continue performing computation. As shown in Fig. 5, we can obtain the solutions of a problem without an explosion of processing time. Thus the improvement in constraint processing enables us to solve problems which have previously been unsolvable.

7.3. Characteristics of Constraint Processing

The use of constraints of NotEqual atom is explained below. An AllDifferent atom is transformed into NotMember atoms. A NotMember atom is transformed into NotEqual atoms. The number of AllDifferent atoms is three times the number of squares in a set of squares. The number of NotEqual atoms is a value obtained by multiplying the number of AllDifferent atoms and the number of possible combinations for any two elements selected from the set of squares; therefore, the number of NotEqual atoms is enormously larger than that of AllDifferent atoms. For example, in the case of a 9 × 9 puzzle, the number of AllDifferent atoms is 27, while that of NotEqual atoms is 972. In the case of a 25 × 25 puzzle, the former is 75, while the latter 22,500. All atoms obtained in the cases above should be eliminated by applying transformation rules. Therefore, the larger the number of atoms is, the higher the number of rule applications becomes.Reducing the number of rule applications, however, may not lead to the shortening of processing time. The application of the "candidate elimination" rule to AllDifferent atoms has the following advantage. As described in Section 6.2, if there are two or more numbers in a body atom, the elimination of two or more numbers can be performed at a time regarding a variable (or two or more variables). Compared to the candidate elimination using constraints of NotEqual atom, the number of transformations is one, but a burden of one transformation becomes heavier. Because NotEqual atom is the most simplified, its constraint processing is also simple. The simpler the processing is, the shorter the processing time becomes, though the number of rule applications is larger.

Since the information in an AllDifferent atom is more than that in a NotEqual atom, we can define various rules for constraint processing by good use of those information. The unique rule, which uses constraints of AllDifferent atom, is called the "number decision" rule (See Section 6.2). When the number of variables to which a value can be assigned is only one, this rule enables assignment of the value to the variable. When this rule is applied to a NotEqual atom, a variable can not be unified with a number unless the number of candidates of a variable is one.However, only when this rule is applied to an AllDifferent atom, a variable can be unified with a number by using other information even when the number of candidates of a variable is not one. This is because elements of the list in the AllDifferent atom do not break the relations among the 9 elements which are different each other in specified squares.

7.4. Effect of Global Processing

As shown in Fig. 5, when the number of variables is 275 or less, constraint processing by using constraints of NotEqual atom shortens the processing time, and when it is over 275, the processing time explodes. This is because when a problem is a small-scale one, the smaller the cost required for single processing, the better the results obtained are; on the other hand, as the size of a problem becomes larger, the clause is branches explosively. Branching of the clause reduces the efficiency of computation, as in the case of backtracking. An explosive branch makes computation impossible. Since constraint processing by using constraints of NotEqual atom can perform only local processing – comparison of two elements in the atom, it is difficult to determine the values of variables; therefore, branches of clauses cause.Constraint processing by using constraints of AllDifferent atom enables the user to get solutions without causing an explosion in processing time. This is because the effective rules suppress the clause's branching, though the cost required for single processing is large. It is clear that the larger the size of a problem is, the more effective such a global processing becomes. Such a global processing can be obtained by the user's improving constraint-solving algorithms. From the observations above, it is clear that a large-scale and complicated problem should be solved not only through local processing but also through global processing. Therefore, we need a computing framework which defines a processing scheme suitable for a problem and improves the system.

VIII. EFFECT OF USER-DEFINED CONSTRAINT PROCESSING

8.1. Definition of Flexible Constraint Processing

In various constraint logic programming languages, including CHIP [5] and Prolog III [3], the domains of constraints have been fixed, the constraint solvers of which have been provided by system designers. In other programming languages, the user also solves problems making use of subroutines and libraries. Therefore, the user is not allowed to change or improve rules and data structures which constitute algorithms, resulting in the incapability of the user when a problem goes beyond the range of the provided system.A computing framework, where a problem is solved by equivalent transformations, enables the user to share the rules as a library. Furthermore, as shown in Section VI, the user is allowed to define rules and data structures which carry out constraint processing suitable for a problem. If the builtin constraint-solving algorithms are suitable for a problem solving, the user easily obtains effective solutions. If not, the computing framework in the present study allows the user to compensate new data structures and constraint-solving rules to obtain effective solutions. Therefore, whenever the user solves complicated knowledge processing problems, the user becomes capable of extending the system.

8.2. Data Structures and ET Rules

In the computing framework proposed by this study, data structures, rules, and control descriptions of computation can be extended freely irrespective of the restrictions of the fixed framework. Section 4.1 introduces a new data structure called an i-var. Such data structures are transformed by equivalent transformation rules. Various constraints are also transformed by equivalent transformation rules. These rules can be easily defined by the definition of a predicate. Therefore efficient data structures, efficient rules, and efficient control can be easily and simply obtained from the specification of a problem.

8.3. Ensuring The Correctness of Solving Method

When the system provides constraint-solving algorithms, it ensures the correctness of constraints to be treated. When a user improves and defines newly constraint atoms, the user should ensure the correctness of solving method; therefore, such theoretical foundation that the user can easily guarantee the correctness of the algorithm is required. The framework of "problem solving based on equivalent transformation," on the other hand, strictly ensures the correctness of a solution. In this computing framework, since each rule transforms equivalently and correctly a declarative description at each computation step, the application of those rules in any order as long as they are correct ensures the correctness of a solution. Thus, rules transform mutually independently and correctly, a requirement which is called the "independence of rules." Therefore, the extension of data structures, changes in rules and the addition of rules due to a change in constraint atoms (as long as the new rules are correct) present no barrier to improving algorithms with the correctness of solving methods preserved.

IX. CONCLUSIONS

This paper has solved number-place problems using a computing framework of "problem solving based on equivalent transformation," and showed the effectiveness of the method. Since this computing framework allows for easy introduction of new data structures or rules while preserving the algorithm's correctness, adding better data structures and rules to the system makes it possible for computations to be performed in a more efficient manner.

REFERENCES

[1] Akama,K. and Nantajeewarawat,E., "Formalization of the Equivalent Transformation Computation Model," Journal of Advanced Computational Intelligence and Intelligent Informatics, vol.10, no.3, pp.245–259, 2006.

[2] Chippimolchai,P., Akama,K., Ishikawa,T., and Wuwongse,V., "Correct Computation with Multi-Head Rules in the Equivalent Transformation Framework," Proc. of the 4th International Conference on Intelligent Technologies, pp.531–538, 2003.

[3] Colmerauer,A., "An introduction to Prolog III," Communications of the ACM, vol.33, no.7, pp.69–90, 1990.

[4] Dechter,R., "Constraint Processing," Morgan Kaufmann Publishers, 2003.

[5] Dincbas,M., et al., "The Constraint Logic Programming Language CHIP," Fifth Generation Computer Systems, Tokyo, Japan, 1988.

[6] Frank,J., Cheeseman,P., and Stutz,J., "When Gravity Fails: Local Search Topology," Journal of Artificial Intelligence Research, vol.7, pp.249–281, 1997.

[7] Freuder,C.E., et al., "Systematic Versus Stochastic Constraint Satisfaction," IJCAI-95, pp.2027–2032, 1995.

[8] Fr¨uhwirth,T., "Theory and Practice of Constraint Handling Rules," Journal of Logic Programming, Special Issue on Constraint Logic Programming, vol.37, nos1-3, pp.95–138, 1998.

[9] Jaffar,J. and Lassez,J.-L., "Constraint Logic Programming," Proc. 14th Ann. ACM Symp. Principles of Programming Languages, pp.111–119, 1987.

[10] Jaffar,J. and Maher,M., "Constraint Logic Programming, A Survey," J. of Logic Programming, vol.19/20, pp.503–581, 1994.

[11] Koike,H., Akama,K., and Mabuchi,H., "Dynamic Interaction of Syntactic and Semantic Analyses Based on the Equivalent Transformation Computation Model," Journal of Advanced Computational Intelligence and Intelligent Informatics, vol.10, no.3, pp.302–311, 2006.

[12] Lloyd,J.W., "Foundations of Logic Programming," Second Edition, Springer-Verlag, 1987.

[13] Mabuchi,H., Akama,K., Miura,K., and Ishikawa,T., "Constraint Solving Specializations for Equality on an Interval-Variable Domain," Journal of Advanced Computational Intelligence and Intelligent Informatics, vol.11, no.2, pp.210–219, 2007.

[14] Mabuchi,H., Akama,K., and Wakatsuki,T., "Equivalent Transformation Rules as Components of Programs," International Journal of Innovative Computing, Information and Control, vol.3, no.3, pp.685–696, 2007.

[15] Mackworth,A., "Constraint Satisfaction," In Encyclopedia of Artifical Intelligence ((ed.) Shapiro, S.C.), vol.1, pp.205–221, JohnWiley & Sons, Inc., 1987.

[16] Minton,S., Johnston,M.D., Philips,A.B., and Laird,P., "Minimizing Conflicts: A Heuristic Method for Constraint Satisfaction and Scheduling Problems," Artificial Intelligence, vol.58, pp.161–205, 1992.

[17] Miyajima,S., Akama,K., Mabuchi,H., and Wakamatsu,Y., "Automatic Detection of Incorrect Rules in Equivalent Transformation Programs," International Journal of Innovative Computing, Information and Control, vol.5, no.8, pp.2203–2218, 2009.

[18] Miyajima,S., Akama,K., and Mabuchi,H., "Algorithmic Debugging of Equivalent Transformation Programs using Oracle Rules," International Journal of Innovative Computing, Information and Control, vol.7, no.8, pp.4703–4716, 2011.

[19] Mizuno,K., Kanoh H., and Nishihara,S., "Solving Constraint Satisfaction Problems by an Adaptive Stochastic Search Method," Journal of Information Processing Society of Japan, vol.39, no.8, pp.2413–2420, 1998.

[20] Morris,P., "The Breakout Method for Escaping from Lacal Minima," AAAI-93, pp.40–45, 1993.

[21] Nishihara,S., "Fundamentals and Perspectives of Constraint Satisfaction Problems," Journal of Japanese Society for Artificial Intelligence, vol.12, no.3, pp.351–358, 1997.

[22] Pettorossi, K. and Proietti, M. , "Transformation of Logic Programs: Foundations and Techniques," Journal of Logic Programming, vol.19/20, pp.261–320, 1994.

[23] Stuart Russell, Peter Norvig, "Artifical Intelligence, A Modern Approach," Second Edition, Prentice Hall Series in Artifical Intelligence, Pearson Education, 2003.
[24] Tsang,E., "Foundations of Constraint Satisfaction," Computation in Cognitive Science, Academic Press, 1993.
[25] van Hentenryck,P., Simonis,H., and Dincbas,M., "Constraint satisfaction using constraint logic programming," Artificial Intelligence, vol.58, pp.113–159, 1992.