International Journal of Innovative Computing, Information and Control
Volume **11**, Number **2**, April **2015**
ICIC International ©2015 ISSN 1349-4198
pp. **523–539**

PROPOSAL FOR PARALLELISM BASED ON EQUIVALENT TRANSFORMATION MODEL AND ANALYSIS

Hiroshi Mabuchi
Software and Information Science
Iwate Prefectural University
Takizawa, Iwate 020-0693, Japan
mabu@iwate-pu.ac.jp

Received April 2014; revised August 2014

Abstract. *In imperative programming languages, the more complicated a problem is, the more difficult extracting parallelism becomes. On the other hand, it can be said that declarative programming languages are suited to parallelism extraction. The equivalent transformation (ET) programming language used in this study is a type of declarative programming language and is based on the ET computation model. The ET programming language is superior to other declarative programming languages in terms of guaranteeing independence, correctness and granularity of the rules that a program consists of as well as the correctness of its computing results (including parallel computing). To demonstrate the effectiveness of parallelism based on the ET computation model, it is essential to introduce the traditional concepts of And-parallelism and Or-parallelism in this study. Through the introduction of these concepts, this paper proposes And-parallelism, Or-parallelism and And & Or-parallelism based on the ET computation model. Then, using these parallel algorithms, a number-place problem, a type of constraint satisfaction problem (CSP), is solved and, by comparing the computation results, the characteristics of each algorithm are analyzed.*

Keywords: Parallelism, Parallel algorithm, Equivalent transformation programming language, Equivalent transformation model, Constraint satisfaction problem

1. Introduction.

Recently, parallel processing, which improves computation efficiency by executing many calculations simultaneously, is growing in significance [7, 21, 24, 25]. Parallel processing works on the principle that a large problem can be split into small problems. These small problems are then processed in parallel.

For many years parallelism was utilized mainly in high-performance computing systems; lately, however, interest has grown in the use of multithreaded applications on multi-core processors [13, 21].

In the field of parallel algorithm study, researchers have been developing efficient parallel algorithms for use in problems[1] which are solvable by sequential solutions [4, 7, 8]. Although some of these algorithms are effective only in a theoretical framework, there are many algorithms which are efficient in practice or provide important concepts for effective implementations.

Extracting parallelism implicitly is not easy. In imperative programming languages, the more complicated a problem is, the more difficult extracting parallelism becomes. On the other hand, it can be said that declarative programming languages are suited to parallelism extraction. This is because declarative programming languages are like logic programming languages in that they use a higher-level of abstraction so that there is no

[1]Many problems in AI and other areas of computer science can be viewed as special cases of constraint satisfaction problems (CSPs) [3, 14, 17, 23, 26, 27, 28].

need to see the detail of how a specific problem must be solved [2, 4, 15, 18, 19, 22]. The challenge faced by parallel computing systems is how to describe a problem and then how to split its functions [24]. Imperative programming languages, typified by C language or C++ language, use low-level abstraction and therefore the problem's functions are deeply intertwined. This makes it difficult to efficiently split the problem's functions. Because of this, it is generally difficult to detect parallelism with imperative programming languages. It is also difficult to guarantee the correctness of parallel computing results. By contrast, of the programming languages, a high level of abstraction is found in declarative programming languages because they express the problem through declarative descriptions. Therefore, the problem's functions can be split in an appropriate and efficient manner and through the use of declarative semantics, the correctness of parallel computing results can also be guaranteed. Based on this, it can be said that declarative programming languages are well suited for extracting parallelism.

The equivalent transformation (ET) programming language [9, 10, 11, 12] used in this study is a type of declarative programming language and is based on the ET computation model. The ET programming language is superior to other declarative programming languages in terms of guaranteeing independence, correctness and granularity of the rules that a program consists of as well as the correctness of its computing results (including parallel computing). Also, from high-level abstraction to low-level abstraction, ET rules, through a single language, express the problem in declarative descriptions and the problem's functions can be split in an appropriate and efficient manner through parallel processing [9, 11]. As a result, the execution efficiency of parallel programs can be improved.

To demonstrate the effectiveness of parallelism based on the ET computation model, it is essential to introduce the traditional concepts of And-parallelism and Or-parallelism [5, 6, 20] in this study. Through the introduction of these concepts, this paper proposes And-parallelism, Or-parallelism and And & Or-parallelism based on the ET computation model. Then, using these parallel algorithms, a number-place problem, a type of constraint satisfaction problem (CSP) [3, 14, 17, 23, 26, 27, 28], is solved and, by comparing the computation results, the characteristics of each algorithm are analyzed.

2. **Parallelism Based on ET Model.**

This section proposes three parallelism, which are And-parallelism, Or-parallelism, and And & Or-parallelism, based on the ET computation model.

2.1. **And-parallelism based on ET model.**

This section proposes a parallel model in which a CSP that has only a single solution solved by And-parallelism.

As shown in Figure 1, And-parallelism processes, in parallel, tasks which have And-relations. And-parallelism must simultaneously and logically satisfy the conditions of each task. The fundamental basis by which problems are solved is to eliminate in the early stages anything irrelevant from the problem domain. This will not result in contradiction as each function of the problem is processed individually in each task. Furthermore, each task solves a partial problem through the use of the same rule set. At this time, And-parallelism can be applied for the solution.

Problem solving by And-parallelism is explained as follows. The following `ans` clause may be taken as an example.

`ans` $\leftarrow$ `atom1, atom2,` $\cdots$, `atomn`.

`ans` is the solution obtained by the computations on the right side. For this `ans` clause, because the computation sequence on the right side of $\leftarrow$ does not depend on `atomj`'s sequence ($1 \leq j \leq n$), computation may be executed by And-parallelism.

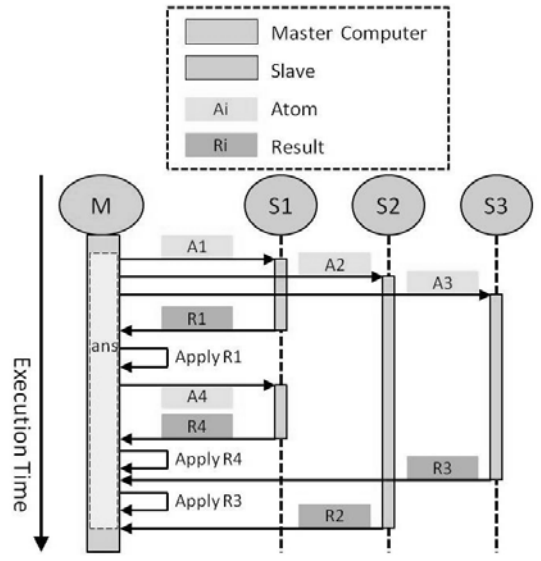

FIGURE 1. Parallel solution in And-parallelism

In Figure 1, the And-parallelism solution is explained. In the problem's initial state, if there are `n` constraints to the problem, the number of slaves is set to `n`. Once computations are started, the master computer sends body atoms to each slave and each slave simultaneously performs specialization (see Section 4.2) [1, 11]. In Figure 1, the number of slaves is 4 (`n` = 4). Of the specialization results from each slave, the first-obtained result (R1) is sent to the master computer and it is then applied by the master computer (Apply R1). Through this application, the problem's constraints are updated and a new atom (A4) is sent to the slave (S1) on which calculations have not yet been performed. And then, the three slaves are compared and the first result (R4) obtained is sent to the master computer. By repeating a full sequence of these operations, parallel computing is realized.

2.2. **Or-parallel based on ET model.** A program for solving CSP is constructed with "non-splitting" rules and "splitting" rules in Or-parallel model. A "non-splitting" rule is one in which the problem state may not be split while a "splitting" rule is one in which the problem state can be split. In this model, we use "splitting" rules to realize the mechanism of Or-parallel.

In the case of Or-parallel computations, there will be two or more splits. And there is no interrelation between those split computations. Therefore, it is a model in which each computation may be carried out independently and simultaneously.

Problem solving by Or-parallelism is explained as follows. The following three `ans` clauses may be taken as an example.

`ans1` $\leftarrow$ `atom11, atom12`, $\cdots$, `atom1n`.
`ans2` $\leftarrow$ `atom21, atom22`, $\cdots$, `atom2n`.
`ans3` $\leftarrow$ `atom31, atom32`, $\cdots$, `atom3n`.

In this example, the range for i, j of `atomij` is $1 \leq i \leq 3$, $1 \leq j \leq n$.

These three `ans` clauses are independent of each other. The rules are applied when processing (computing) each `atomij`, and solution `ans` is obtained when all atoms in a single `ans` clause are processed. The order of priority for the application of the rules must be defined at the beginning of the program, so the priority of "non-splitting" rules will be higher than that of "splitting" rules. That is, "non-splitting" rules should be given priority in application. This is because computation efficiency decreases when "splitting"

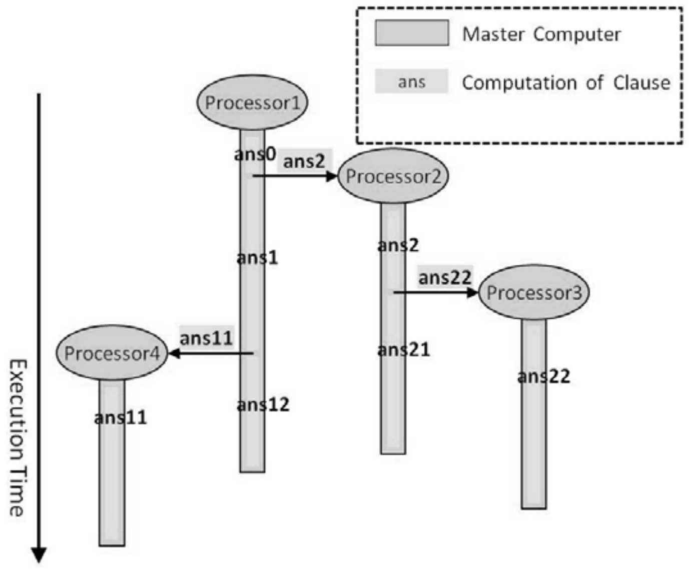

FIGURE 2. Parallel solution in Or-parallelism

rules are used. Only when the problem cannot be solved with "non-splitting" rules should "splitting" rules be used.

Figure 2 shows parallel solution in Or-parallelism. After the master computer starts processing (Process1), the process is executed through application of "non-splitting" rules (`ans0`) as long as the rules are applicable. Only when the rules are no longer applicable should "splitting" rules be then applied. As a result, `ans0` is split into `ans1` and `ans2`. Similarly, Process1's computations are carried out using "non-splitting" rules (`ans1`) until the "non-splitting" rules are no longer applicable and the use of "splitting" rules becomes absolutely necessary. As a result, `ans1` is split into `ans11` and `ans12`. The problem may be solved by repeating the application of these processes. As the number of problem state splits increases, so too does the number of processors, thus increasing the computation cost.

2.3. **And & Or-parallelism based on ET model.** And & Or-parallelism is parallelism in which And-parallelism and Or-parallelism are mixed. Figure 3 shows parallel solution in And & Or-parallelism.

In this model, And-parallelism computations are performed as long as "non-splitting" rules can be applied. When computation becomes impossible through the application of "non-splitting" rules, it will switch to Or-parallelism processing and apply "splitting" rules.

3. Comparison between Parallelism Based on ET Model and Conventional Parallelism.

In this section, we will demonstrate the advantages of parallelism based on the ET model through a detailed comparison of it to conventional parallelism.

3.1. **Characteristics and problem areas of parallelism in imperative programming languages.** This section presents C++ language, an efficient programming language.

[Characteristics]

In principle, because the language is capable of directly controlling hardware, it can make maximal use of computation resources to describe a program that has higher processing efficiency.

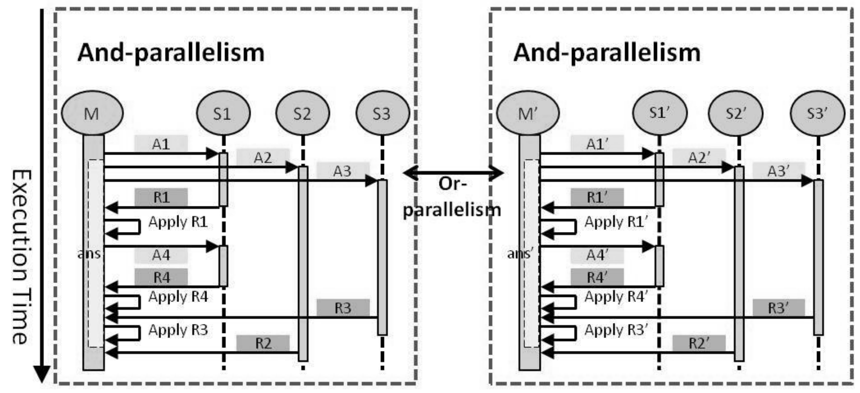

FIGURE 3. Parallel solution in And & Or-parallelism

[Problem areas]

As program complexity increases, difficulty in its manual optimization also increases. Especially, global optimization of a program by manpower is difficult. Even if an attempt is made to automate global optimization, in theory it is difficult to achieve because exact correctness does not exist in the specification and program.

3.2. **Characteristics and problem areas of parallelism in declarative programming languages.** This section presents declarative programming languages [2, 4, 15, 18, 19, 22].

[Characteristics]

Ideally, it lets problems be automatically solved through its description of the problem with declarative descriptions.

[Problem areas]

If the computation's correctness must be strictly guaranteed, either the problem description's expressive power or its execution efficiency becomes insufficient and, in many cases, operations with no theoretical grounding are employed in the name of execution efficiency.

3.3. **Importance and advantages of parallelism based on ET model.** This section presents the importance and advantages of parallelism based on ET model.

[Importance]

In parallelism based on ET model, the program's correctness is strictly defined as a mathematical relationship between specification and program by completely separating the problem description (formal specification), which uses definite clause sets and representation sets of first-order predicate logic in its declarative description, from the program (procedure). Instead of directly executing the formal specification, a correct program specifically for the formal specification is created.

Under this general framework for correct parallel processing, application of parallel processing may be made to a wider range of general problems compared to those that can be handled by conventional parallel processing (see Sections 3.1 and 3.2) and it is possible for problems to be efficiently solved while guaranteeing the computation's correctness.

[Comparison with imperative programming languages]

In ET model-based parallel processing, the correctness of the executable program, obtained through a theory that guarantees the correctness of program generation and parallel processing, is fully guaranteed. Also, at the time of rule creation, global optimization and guaranteeing correctness for the program can take place at the same time. Because the resultant program that is generated is an executable procedure that is independent of implementation, it can be converted into an existing imperative programming language. At this point, the existing optimization technique can be utilized.

[Comparison with declarative programming languages]

In the ET model, executable programs can describe more than other declarative programming language frameworks because any type of description is acceptable as long as the requirements for the program's correctness are satisfied. Therefore, the existence of efficient programs in the ET model is highly likely and, as a result, efficient computation can be performed compared to declarative programming languages.

In program processing, many declarative programming languages have more constraints on their computation principles than imperative programming languages, but there are fewer constraints with the ET theory than with declarative programming languages.

4. **Formalization of Problem and Rules for Problem Solving.**

In this section, a number-place problem, a typical CSP, is formalized based on the ET computation model [9, 11, 12] and the rules for solving this problem are explained.

4.1. **Formalization of problem.**

This section formalizes number-place problems (henceforth NP represents a single problem and NPs represents more than one problem) by declarative descriptions [16]. Constraints to be satisfied when solving a number-place problem are to satisfy an initial arrangement of the problem and to adhere to the constraints of the problem. These are represented with the following clause. Symbols starting with "*" represent variables.

```
(answer *NP) ←
    (initial-arrangement *NP),
    (rules-of-NP *NP).
```

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

```
(initial-arrangement *NP) ←
    (= *NP ( (? 4 ? 3 1 ? 6 ? 7)
             (6 ? ? ? ? 7 ? 3 ?)
             (? ? ? ? 8 ? ? ? 4)
             (7 ? ? ? ? ? 5 ? ?)
             (? ? 3 6 ? ? 4 ? ?)
             (1 ? ? 5 ? ? ? 8 ?)
             (4 5 ? 8 ? ? ? ? 9)
             (? 1 ? ? ? 3 ? ? ?)
             (? ? 2 ? 5 ? 7 ? ?))).
```

A variable, `*NP`, is equal to the list which represents the given assignment of a problem. "rules-of-NP" predicate is expressed in accordance with the following two constraints:

(Constraint 1): The numbers 1 through 9 will be placed into each small blank square.

(Constraint 2): The same number cannot be placed in any one column or row, nor within any one sub-grid surrounded by a thicker border.

4.2. **Specialization in number-place problem.**

allDifferent atom representing (Constraint 2) (see Section 4.1), providing that elements of the list are different from each other. As seen in `(allDifferent (3 *a~(2 3 4) 5 *b~(2 3 4 5)))`, an atom includes the predicate name, allDifferent, and the list which is surrounded at both ends by `(` and `)`. The allDifferent predicate controls the constants, variables and i-vars, which are elements of its list, to eliminate numbers based on the relationship between elements, to substitute numbers for variables or to eliminate an i-var's information. 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 "(1 2 3)," as with `*x~(1 2 3)`. Specialization of each variable is executed based on the following six rules.

1. Rule allDifferent1 (see (1) in Appendix): In the constraint list `*list`, the constants and variables are first separated and only the constants are eliminated from the list.
   ex.)
   ```
   (allDifferent (3 *a~(2 3 4) 5 *b~(2 3 4 5)))
                   ↓
   (allDifferent (*a~(2 4) *b~(2 4)))
   ```
2. Rule allDifferent2 (see (2) in Appendix): If the constraint list is empty, it means that specialization of the list has been performed and the list can be eliminated from the specialization routine.
3. Rule allDifferent3 (see (3) in Appendix): When i-var information contains only one number, its variable can be specialized to that number and the number can be eliminated from the list.
   ex.)
   ```
   (allDifferent (*a~(2 3 4) *b~(2 3 4) *c~(3)))
                   ↓
   (allDifferent (*a~(2 3 4) *b~(2 3 4) 3))
                   ↓
   (allDifferent (*a~(2 4) *b~(2 4))
   ```
4. Rule split2-1 (see (4) in Appendix): If the information of two variables in the constraint list `*list` consists of the same two numbers, these two numbers can be eliminated from the other variables in `*list`.
   ```
   (allDifferent (*a~(2 3) *b~(1 2 3 4) *c~(2 3) *d~(2 3 4)))
                   ↓
   (allDifferent (*a~(2 3) *b~(1 4) *c~(2 3) *d~(4)))
   ```
5. Rule splitN-1 (see (5) in Appendix): If the information of two variables in the constraint list `*list` consists of the same n numbers, these n numbers can be eliminated from the other variables in `*list`. This rule follows similar processing to Rule split2-1.
6. Rule allDifferent4 (see (6) in Appendix): This rule splits the problem state into two states. Using `*a~(1 2)` as an example, if the i-var information contains two elements, the problem state is split into two cases where the first element is 1 and the second element is 2 and the computations then continue for each. As a result, there will now be two "Body". Because this rule results in high computation costs, it should be given a lower priority.

5. **Serial and Parallel Solutions for Number-Place Problems.** In this section, using each parallelism proposed in Section 2, the solutions to number-place problems, the computation processes and the execution times are explained.

5.1. **Serial solution for number-place problems.** Figure 4 depicts a serial solution for number-place problems. As demonstrated in this figure, to solve a given problem, the serial solution removes a sub-constraint from the given problem. Specialization is performed to the removed sub-constraint and through the transmission of information

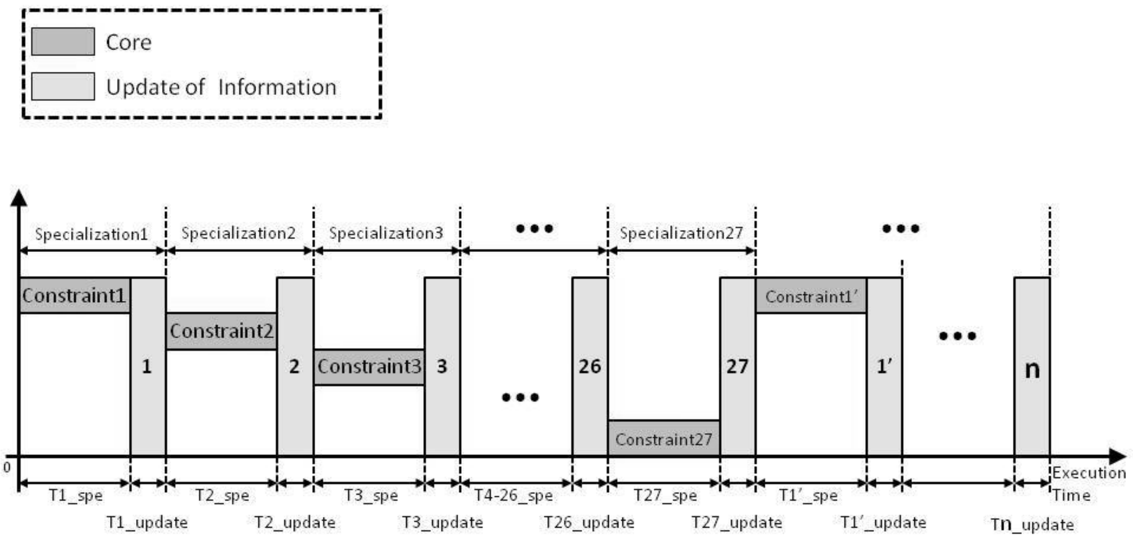

FIGURE 4. Serial solution for number-place problems

resulting from the specialization, the information of the variable in the sub-constraint is updated. Next, another sub-constraint is removed and the same process is repeated. This process will be repeated until all variables in the given problem have been eliminated. When all variables in the given problem are transformed into constants, that will be the solution to the problem.

It is easy to predict the execution time that will be required for this solution to solve the problem. The serial solution's execution time is the sum of the sum of specialization times and the time required to update each variable's information.

5.2. **Solution for number-place problems in And-parallelism.** Figure 5 depicts the solution process in And-parallelism. Since there are 27 constraints (9 rows, 9 columns, 9 sub-grids) for the And-parallelism solution, 27 cores need to be set. Each of the 27 cores performs a constraint specialization sent from the master computer. Let the time required to complete all constraint specializations – in the case of Process1 in Figure 5, and this is the time to complete Constraint2's specialization as it took the most time – be $T1\_spe$. Then, the information of all variables in all constraints in Process1 is updated. Let the time required for this be $T1\_update$. This process will be repeated n times until all variables are eliminated and from this, the given problem's solution is obtained.

The execution time required to solve this problem is the sum of the sum of $T1\_spe$ to Tn_spe and the sum of $T1\_update$ to Tn_update.

5.3. **Solution for number-place problems in Or-parallelism.** With "splitting" rules, Or-parallelism can be easily realized. A "splitting" rule is a rule that splits a single problem state into multiple different states. The master computer performs successive specializations of each constraint until the split point is reached. For example, the variable `*a` in the i-var `*a~(1 2)` can have two numbers, 1 and 2, placed into it. Therefore, `*a` is split into case 1 and case 2.

Figure 6 depicts Or-parallelism for number-place problems. Although the problem state is always split into two different states in this example, it may be split into three or more states.

The numbers ① to ⑥ indicate the number of body atoms (see Appendix) to be split, and in this example, a split into six body atoms is shown. Each of the six body atoms is individually processed in the flow as depicted in ① to ⑥. The horizontal axis shows

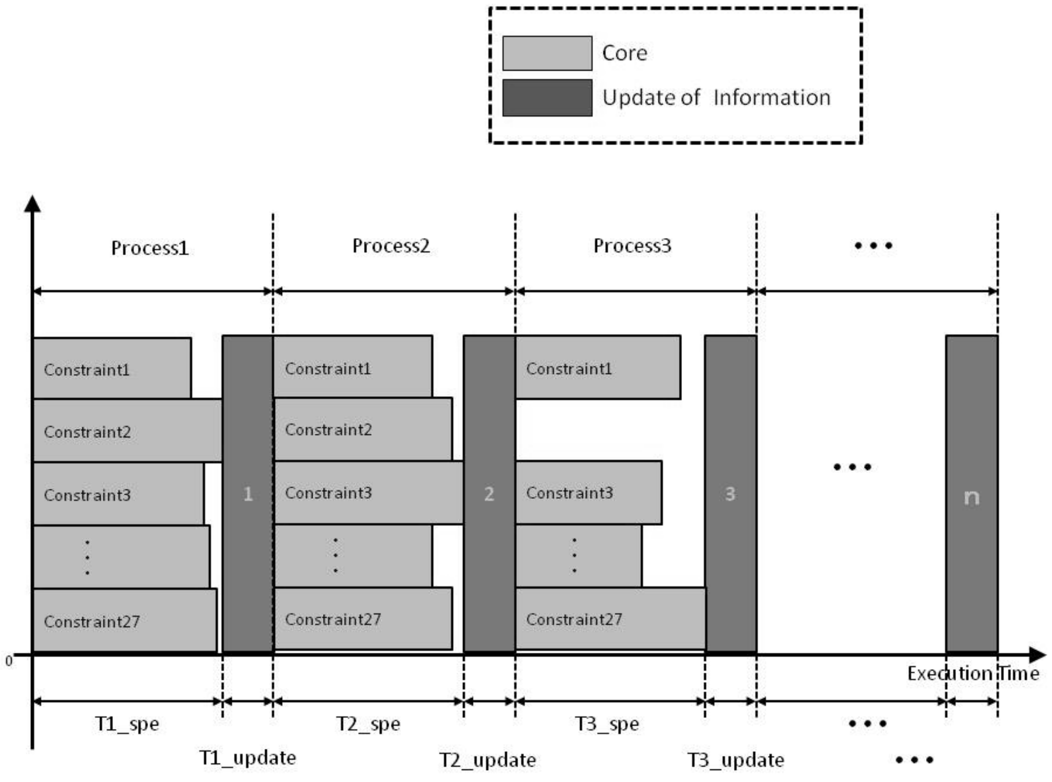

FIGURE 5. Solution for number-place problems in And-parallelism

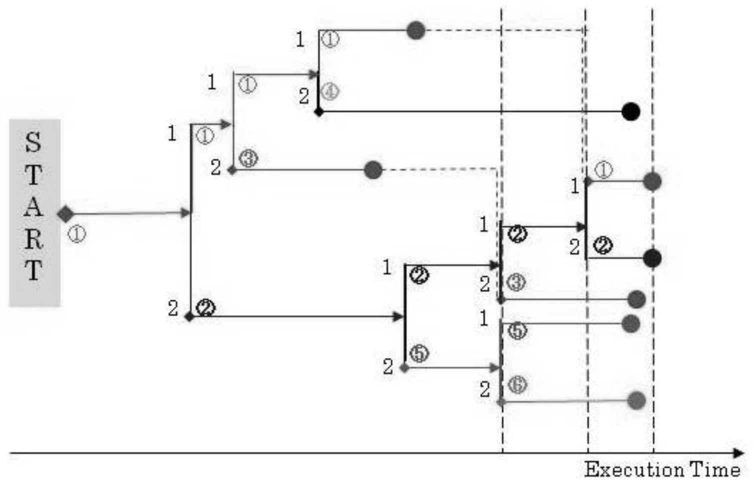

FIGURE 6. Solution for number-place problems in Or-parallelism

the execution time; the reason why the lines are different lengths following a split, shown on the horizontal axis, is because different computations are performed. These splits are performed under a "splitting" rule.

The state of the end atom (leaf-node) is the result of multiple splits. The execution time required to achieve each leaf-node state is individually measured and the longest of these execution times will be the computation time for this solution. In Figure 6 example, ① and ② took the longest time to obtain the leaf-node state, so their time becomes the computation time.

5.4. **Solution for number-place problems in And & Or-parallelism.** In this model, computations are performed in And-parallelism as long as "splitting" rules are applicable. And, Or-parallelism is implemented by "splitting" rules. Therefore, the total execution time is the sum of the And-parallelism execution time and the Or-parallelism execution time without including the time that was taken to perform the splitting (see Figure 7).

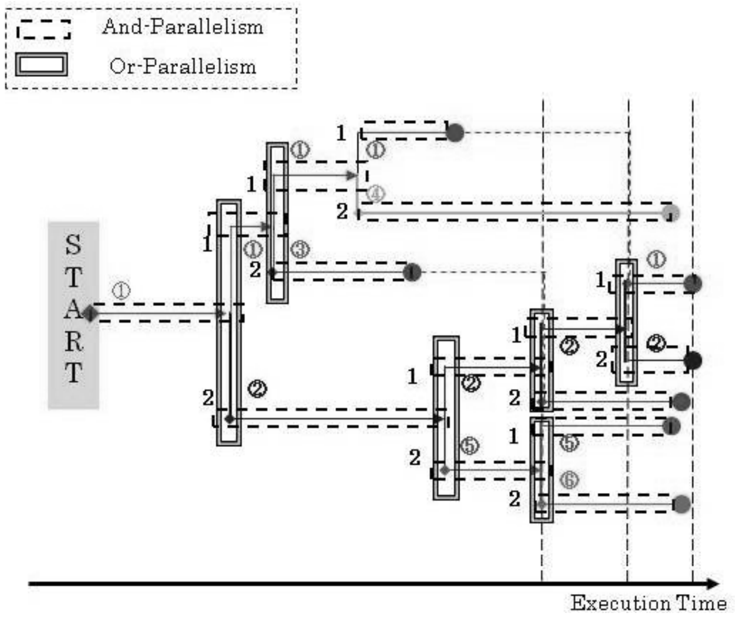

FIGURE 7. Solution for number-place problems in And & Or-parallelism

Also, the execution time for this parallel solution depends on the number of working cores. If m cores perform processing as child-nodes, the And-parallelism execution time at the first node is assumed to be $t0/m$. ti is the execution time at each Or-parallelism node i shown in Figure 6.

The total computation time can be assumed to be the sum of the execution time at each node (including each leaf) and the split time at nodes other than leaves. When compared to serial computation, if the average magnification of And-parallelism is m, the computation time at each node or leaf is assumed to be $\frac{1}{m}$ of serial computation. Here, average magnification is the average of the number of cores used in a computation.

Using Figure 8 as an example, the process of obtaining execution time is explained.

Let

Actual serial time: T_e,

Theoretical serial time: T_s,

Parallel computation time: $T_p = \Sigma\ t_i$,

Number of splits: n,

And-parallelism magnification: m,

Average split time: $t_s = (T_e - T_s)/n$.

Using these, And & Or-parallelism execution time can be calculated with the following equation.

$$\begin{aligned} T_{And\text{-}Or} = {} & [(t0\ -\ ts0)/m\ +\ ts0] \\ & + [(t1\ -\ ts1)/m\ +\ ts1] \\ & + [(t2\ -\ ts2)/m\ +\ ts2] \\ & + [(t11\ -\ ts11)/m\ +\ ts11] \end{aligned}$$

$$
\begin{aligned}
&+ [(t21 - ts21)/m + ts21] \\
&+ t111/m + t112/m + t12/m + t211/m \\
&+ t212/m + t22/m \\
&= T_p/m + n \times \Sigma\, tsj \times [(m-1)/m] \\
&= T_p/m + n \times t_s \times [(m-1)/m]
\end{aligned}
$$

Here, $[(t0 - ts0)/m + ts0]$ to $[(t21 - ts21)/m + ts21]$ are nodes from which leaves have been removed, and tsj indicates the split time at each node j.

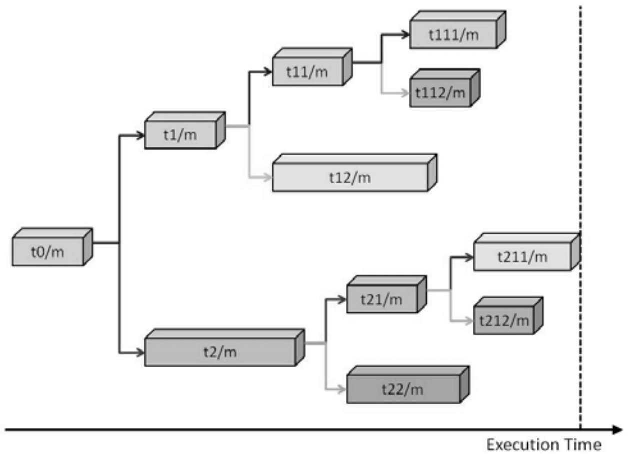

FIGURE 8. Execution time in And & Or-parallelism

6. **Results and Analysis of Evaluation Experiment.** The efficiency of the solution model described in Section 5 is estimated. First, 100 number-place problems to be solved under And-parallelism (A-group) and 100 number-place problems to be solved under Or-parallelism (B-group) are selected. Then, these problems are executed and the characteristics of these two solutions are analyzed. Lastly, we will demonstrate that And & Or-parallelism is more efficient than Or-parallelism.

6.1. **Characteristics of A-group's problems.** When solving each of the problems, if a regular pattern is not detectable, it is difficult to correctly analyze each problem's execution results. In discussing the characteristics of the problems, all the problems are executed. Then, the execution time, the number of numbers assigned to the number-place problem's grids and the specialization time are examined. These three sets of data have a significant effect on the difficulty of each problem.

When these experiment data are analyzed, the following conclusions are drawn.

- In solving a number-place problem, the longer it requires for execution time, the more difficult that problem is.
- The higher the number of numbers assigned to a number-place problem's grids is, the quicker that problem can be solved.
- If less time is required for specialization to solve a problem, it can be thought that the problem is easy.

Table 1. Experimental results

Relationship between times of specialization and magnification		
Problem No.	Times of specialization	Magnification
A001	9	12.9972384
A002	5	11.498833
A003	8	12.2501142
A004	6	12.2575068
A005	4	10.4339508
A006	5	12.8154586
A007	7	12.6742232
A008	8	14.164478
A009	9	12.5454958
A010	6	11.9636074
A011	9	14.5604178
A012	6	14.3381648
A013	10	13.5176254
A014	7	12.5706168
A015	6	13.6237528
A016	10	11.9367016

6.2. **Execution efficiency of A-group problems in And-parallelism.** In this section, the average magnification of the 100 problems in A-group in And-parallelism will be found. To achieve this, an experiment was conducted to estimate the serial solution's execution time the And-parallelism execution time [8].

In this experiment, the serial solution program was executed first. From its actual execution time, the And-parallelism execution time was estimated, the magnifications of the time estimated by Section 5.2's method and the actual time were computed, and the number of times of specialization was examined.

From Table 1, the average magnification of And-parallelism can be assumed to be 12.

6.3. **Execution efficiency of B-group problems in Or-parallelism.** The solution described in Section 5.3 was used to estimate parallel execution time. Then, the serial execution time and the number of splits were examined, and the magnification between parallel execution and serial execution was computed.

Experiment procedure:

1. Execute all the problems in B-group.
2. Compare the parallel execution time (paraTime) with the serial execution time (seriTime).
3. Examine the relationship between the number of splits and execution time.

Results:

1. In parallel computation, there is no significant variation in each problem's execution time.
2. In serial computation, there is an explosion in execution time when problems require more splitting.
3. When problems require more splitting, their execution time becomes longer.

6.4. **Comparison between And-parallelism and Or-parallelism.** In order to compare And-parallelism of A-group with Or-parallelism, of the problems for which the correct solution was obtained by serial computation, problems that were both easy and difficult

Table 2. Experimental results

Problem No.	paraTime (msec)	seriTime (msec)	times of split	magnification
B001	2437	2515	1	1.032
B012	1796	2139	2	1.191
B013	2078	2863	3	1.378
B025	2000	2610	4	1.305
B029	1594	2362	5	1.482
B088	2563	3690	6	1.440
B030	2485	4198	7	1.705
B047	3047	5828	8	1.913
B009	3904	9155	9	2.345
B023	4140	13052	16	2.015
B045	4359	8785	18	3.152
B033	4328	15131	22	3.496
B015	4530	16324	31	3.604
B081	3609	18844	38	5.221
B068	4292	24808	47	5.780
B017	4609	32454	49	7.041
B063	4531	27662	50	6.105
B039	4578	34051	52	7.438

Table 3. Serial execution time

Problem No.	Serial execution time (msec)
A075	1122
A031	3748

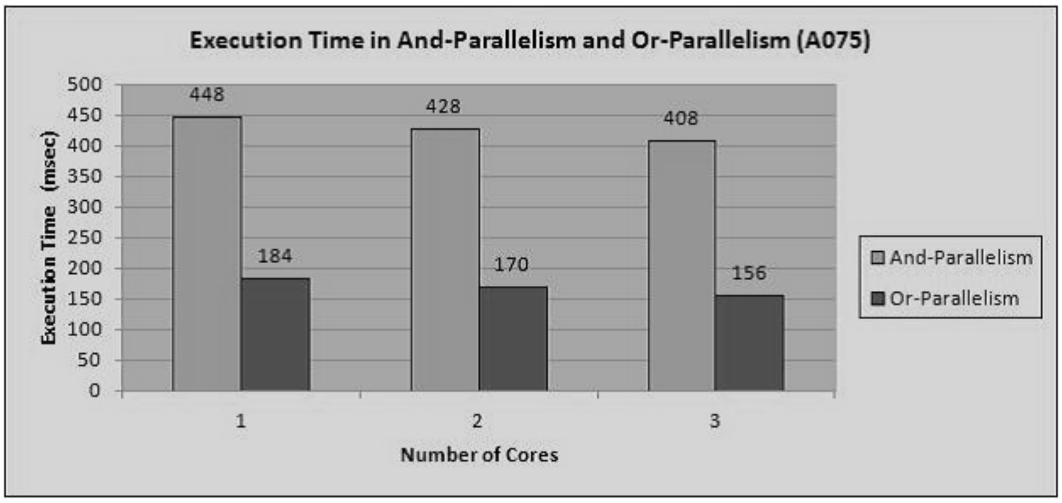

Figure 9. Execution time in And-parallelism and Or-parallelism (A075)

were chosen. Then, the execution efficiency of And-parallelism and Or-parallelism was compared.

As demonstrated in Figure 9, the execution efficiency of Or-parallelism is higher than that of And-parallelism.

6.5. **Execution efficiency of B-group problems in And & Or-parallelism.** Here, the execution efficiency of And & Or-parallelism is compared with that of Or-parallelism. For the purpose of comparison, take B030, a large and complex problem in B-group, as an example.

The following data are obtained from the experiment.
Actual serial time: $T_e = 5983$ (msec),
Theoretical serial time: $T_s = 4198$ (msec),
Parallel computation time: $T_p = \Sigma\ t_i = 2484$ (msec),
Number of splits: $n = 7$,
And-parallelism magnification: $m = 12$,
Average split time: $t_s = (T_e - T_s)/n = 255$.

Therefore, based on computation method in Section 5.4, the execution time obtained was 1843 (msec), and And & Or-parallelism execution time was approximately 3.25 times faster than that of Or-parallelism.

7. **Application of the Proposed Method.**
In general, it can be considered that the proposed method is very effective for problems, such as NP-hard problems, that are difficult to solve.

For example, we are currently addressing container pre-marshalling problems [31]. A container pre-marshalling problem is one in which prioritized containers that are randomly stacked two dimensionally are reshuffled so that higher priority containers will not be brought down while minimizing the total number of moves required for the reshuffle.

Also, the proposed method is effective for puzzle problems such as Pic-a-Pix puzzles, number area problems and crossword puzzles in addition to number-place problems used in this paper.

On the other hand, when it comes to practical applications, the proposed method can be applied to scheduling problems and the proposed method would prove to be effective in solving this type of problem [30]. Examples of this type of problem include class scheduling problems, travel scheduling problems and nurse scheduling problems.

8. **Conclusions.**
This paper proposed And-parallelism, Or-parallelism and And & Or-parallelism based on the ET computation model. And using these algorithms, we solved number-place problems – a type of CSP – and analyzed the characteristics of each algorithm by comparing their computation methods and execution times.

It is hoped that the outcomes reached in this study can be applied to other CSPs, possibly with similar results.

REFERENCES

[1] A. Kiyoshi, Y. Tadayuki and M. Eiichi, Programming based on equivalent transformation, *Information Processing Society of Japan*, vol.39, no.82, 1998.

[2] D. C. Gras and M. Hermenegildo, Non-strict independence-based program parallelization using sharing and freeness information, *Theoretical Computer Science*, vol.410, pp.4704-4723, 2009.

[3] R. Dechter, *Constraint Processing*, Morgan Kaufmann Publishers, 2003.

[4] G. Gupta, E. Pontelli, K. Ali, M. Carlsson and M. Hermenegildo, Parallel execution of Prolog programs: A survey, *ACM Trans. Programming Languages and Systems*, vol.23, pp.472-602, 2001.

[5] G. Gupta, M. Hermenegildo and V. S. Costa, And-or parallel prolog: A recomputation based approach, *New Generation Computing*, vol.1, no.3-4, pp.297-322, 1993.

[6] G. Gupta, M. Hermenegildo, E. Pontelli and V. S. Costa, ACE: And/Or-parallel copying-based execution of logic programs, *Proc. of ICLP*, pp.93-109, 1994.

[7] G. E. Blelloch, Programming parallel algorithms, *Magazine Communications of the ACM CACM Homepage Archive*, vol.39, no.3, 1996.

[8] H. Mabuchi, Effect estimation method of parallel computing based on dynamic generation of equivalent transformation rules, *International Journal of Modern Engineering Research*, vol.3, no.5, pp.3181-3187, 2013.

[9] H. Mabuchi, K. Akama and T. Wakatsuki, Equivalent transformation rules as components of programs, *International Journal of Innovative Computing, Information and Control*, vol.3, no.3, pp.685-696, 2007.

[10] H. Ogasawara, K. Akama, H. Koike, H. Mabuchi and Y. Saito, Parallel processing method based on equivalent transformation, *Proc. of IEEE International Conference Intelligent Engineering Systems*, pp.111-116, 2005.

[11] K. Akama, E. Nantajeewarawat and H. Ogasawara, A processing model based on equivalent transformation and a sufficient condition for program correctness, *International Journal of Foundations of Computer Science*, vol.3, 2010.

[12] K. Miura, K. Akama, H. Mabuchi and H. Koike, Theoretical basis for making equivalent transformation rules from logical equivalences for program synthesis, *International Journal of Innovative Computing, Information and Control*, vol.9, no.6, pp.2635-2650, 2013.

[13] K. Minami, Performance improvement of application on the "K computer", *IEICE Transactions*, vol.95, no.2, pp.125-130, 2012.

[14] K. Uchino, S. Kubota, H. Konoh and S. Nishihara, Classifying CSPs on the basis of parallelism, *Information Processing Society of Japan*, vol.35, pp.2676-2684, 1994.

[15] J. W. Lloyd, *Foundations of Logic Programming*, 2nd Edition, Springer-Verlag, 1987.

[16] H. Mabuchi, Efficient solution of constraint satisfaction problems by equivalent transformation, *International Journal of Computational Engineering Research*, vol.3, no.11, pp.61-70, 2013.

[17] A. Mackworth, Constraint satisfaction, *Encyclopedia of Artificial Intelligence*, vol.1, pp.205-221, 1987.

[18] M. Hermenegildo, Towards efficient parallel implementation of concurrent constraint logic programming, *Proc. of ICOT/NFS Workshop on Parallel Logic Programming and Its Programming Environments*, Oregon, 1994.

[19] M. Nakaizumi, H. Shen, H. Kobayashi and T. Nakamura, Implementing functional programs based on the SPMD Modek, *Technical Report of Information Processing Society of Japan*, vol.97, no.61, pp.25-30, 1997.

[20] M. V. Hermenegildo and K. J. Greene, The &-Prolog system: Exploiting independent And-parallelism, *New Generation Computing*, vol.9, no.3-4, pp.233-257, 1991.

[21] N. Nishimura, Grid computing and parallel computing, *Journal of the Japan Society for Computational Engineering and Science*, vol.10, no.3, pp.1198-1202, 2005.

[22] K. Pettorossi and M. Proietti, Transformation of logic programs: Foundations and techniques, *Journal of Logic Programming*, vol.19-20, pp.261-320, 1994.

[23] S. Nishihara, Fundamentals and perspectives of constraint satisfaction problems, *Japanese Society for Artificial Intelligence*, vol.12, no.3, pp.351-358, 1997.

[24] T. Abe, T. Hiraishi, Y. Miyake and T. Iwashita, Job-level parallel executions for satisfying distributed constraints, *Information Processing Society of Japan*, vol.59, pp.1-8, 2011.

[25] W. D. Hillis and G. L. Steele Jr., Data parallel algorithms, *Magazine Communications of the ACM – Special Issue on Parallelism CACM Homepage Archive*, vol.29, no.12, 1986.

[26] Y. Makoto, E. H. Durfee, T. Ishida and K. Kuwabara, The distributed constraint satisfaction problem: Formalization and algorithms, *IEEE Trans. Knowledge and Data Engineering*, vol.10, pp.673-685, 1998.

[27] Y. Makoto and K. Hirayama, Distributed breakout: Iterative improvement algorithm for solving distributed constraint satisfaction problem (special issue on parallel processing), *Information Processing Society of Japan*, vol.39, no.6, 1998.

[28] Y. Saito, M. Munetomo and K. Akama, Generation of correct parallel programs for solving constraint satisfaction problems, *IPSJ SIG Technical Report*, vol.20, pp.103-108, 2006.

[29] *http://ext-web.edu.sgu.ac.jp/koike/eti/documents/eti_builtin/index.htm.*

[30] A. Allahverdi, C. T. Ng, T. C. E. Cheng and M. Y. Kovalyov, A survey of scheduling problems with setup times or costs, *European Journal of Operational Research*, vol.187, no.3, pp.985-1032, 2008.

[31] C. Expósito-Izquierdo, B. Melián-Batista and M. Moreno-Vega, Pre-Marshalling problem: Heuristic solution method and instances generator, *Expert Systems with Applications*, vol.39, no.9, pp.8337-8349, 2012.

Appendix.

$$\begin{aligned}
&(\text{Rule allDifferent1}\\
&(\text{Head } (\text{allDifferent } \ast\text{list}))\\
&(\text{Cond } (\text{existNumber } \ast\text{list } ?))\\
&(\text{Body } (\text{exec } (\text{search } \ast\text{list } \ast\text{N } \ast\text{I})\\
&(\text{diffnumbers } \ast\text{N})\\
&(\text{allDel } \ast\text{I } \ast\text{N}))\\
&(\text{allDifferent } \ast\text{I})))
\end{aligned} \tag{1}$$

"allDifferent1" is the name of the rule and "Head" stands for the proposition before replacement. "Cond" is the condition under which this rule can be applied. In this example, the rule can be applied in cases where there are constants in `*list`. "Body" stands for the proposition obtained by replacing the proposition in the Head. "exec" in "Body" means that there is execution of search atom, diffnumbers atom and allDel atom. When this execution succeeds, (allDifferent `*list`) is replaced by (allDifferent `*I`).

An explanation is omitted for the following rules because similar procedures are performed.

$$\begin{aligned}
&(\text{Rule allDifferent2}\\
&(\text{Head } (\text{allDifferent } ()))\\
&(\text{Body}))
\end{aligned} \tag{2}$$

$$\begin{aligned}
&(\text{Rule allDifferent3}\\
&(\text{Head } (\text{allDifferent } \ast\text{list}))\\
&(\text{Cond } (\text{ivars } \ast\text{list})\\
&(\text{choice } \ast\text{list } \ast\text{x } \ast\text{rest}))\\
&(\text{Body } (\text{exec } (\text{getInfo } \ast\text{x } (\text{or } \ast\text{one}))\\
&(\text{rmInfo } \ast\text{x})\\
&(= \ \ast\text{x } \ast\text{one})\\
&(\text{allDel } \ast\text{rest } (\ast\text{ one})))\\
&(\text{allDifferent } \ast\text{rest})))
\end{aligned} \tag{3}$$

$$\begin{aligned}
&(\text{Rule split2} - 1\\
&(\text{Head } (\text{allDifferent } \ast\text{list}))\\
&(\text{Cond } (\text{ivars } \ast\text{list})\\
&(\text{split2 } \ast\text{list } \ast\text{x } \ast\text{y } \ast\text{m } \ast\text{n } \ast\text{rest})\\
&(\text{notNil } \ast\text{rest}))\\
&(\text{Body } (\text{exec } (\text{remove } \ast\text{rest } \ast\text{m } \ast\text{n } \ast\text{newrest}))\\
&(\text{allDifferent } \ast\text{newrest})\\
&(\text{allDifferent } (\ast\text{x } \ast\text{y}))))
\end{aligned} \tag{4}$$

$$
\begin{aligned}
&(\text{Rule splitN}-1\\
&(\text{Head (allDifferent } *\text{list}))\\
&(\text{Cond (ivars } *\text{list})\\
&(\text{splitN}-1\ *\text{list } *\text{x } *\text{rest } *\text{union } *\text{kazu } *\text{val}))\\
&(\text{Body (exec (update } *\text{x } *\text{union } *\text{kazu } *\text{val}))\\
&(\text{allDifferent } *\text{rest})))
\end{aligned} \tag{5}
$$

$$
\begin{aligned}
&(\text{Rule allDifferent4}\\
&(\text{Head (allDifferent } *\text{list}))\\
&(\text{Cond (ivars } *\text{list})\\
&(\text{choice}-\text{w } *\text{list } *\text{double})\\
&(\text{ivar } *\text{double})\\
&(\text{getInfo } *\text{double (or } *\text{one } *\text{two})))\\
&(\text{Body (exec (rmInfo } *\text{double})\\
&(=\ *\text{double } *\text{one}))\\
&(\text{allDifferent } *\text{list}))\\
&\quad(\text{Body (exec (rmInfo } *\text{double})\\
&\quad(=\ *\text{double } *\text{two}))\\
&\quad(\text{allDifferent } *\text{list})))
\end{aligned} \tag{6}
$$

For further information on built-in atoms, etc., please refer to [29] in References.