ICOT Technical Report: TR-568

TR-568

An Impelentation of TMS in Concurrent Logic Programming Language; Preliminary Report

by
K. Satoh, N. Iwayama & E. Sugino

July, 1990

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

An Implementation of TMS in Concurrent Logic Programming Language: Preliminary Report *

Ken Satoh, Noboru Iwayama, Eiji Sugino
Institute for New Generation Computer Technology
1-4-28 Mita, Minato-ku, Tokyo 108, Japan
email: ksatoh@icot.jp, iwayama@icot.jp, sugino@icot.jp
Kurt Konolige
SRI International, Menlo Park, Stanford, CA, USA
email: konolige@ai.sri.com

April 30, 1990

Abstract

This paper presents an implementation of TMS (Truth Maintenance System) in KL1, a concurrent logic programming language developed in ICOT. A TMS has been intensively developed in AI community for nonmonotonic reasoning and hypothetical reasoning. In this paper, we firstly prove correctness of nondeterministic algorithms to compute TMS. Then, we provide a KL1 implementation of TMS and give a preliminary result.

Areas: Reasoning

Key Words: Truth Maintenance System, Concurrent Logic Language, Nonmonotonic Reasoning, Frame Problem

*This work was initiated by Kurt Konolige when he visited in ICOT from November 20 to December 8, 1989.

1 Introduction

Truth Maintenance System (TMS) [Doyle79] has been actively investigated in AI community for hypothetical reasoning and nonmonotonic reasoning. Recently, some researches reconsidered a semantics of TMS [Fujiwara89, Reinfrank89]. However, they have not proved correctness of any algorithms for TMS in their papers. This paper firstly shows correctness for such algorithms.

Then, we provide an implementation of TMS in KL1, a concurrent logic programming language developed in ICOT. In this paper, we implement TMS by translating each justification into processes and we use message passing to manipulate each justification. This implementation avoids the *frame problem* [Genesereth87] because each channel connected with process can be regarded as a connection between relevant information and so, through this channel, we can change relevant information without looking into irrelevant information.

Finally, we give a preliminary result for this implementation.

2 TMS

We follow the definition of TMS [Fujiwara89].

Definition 1 TMS:
A TMS *is a triple* $D = \langle N, J, C \rangle$ *such that*

1. N *is a finite set (The elements of* N *will be called* nodes*).*
2. J *is a subset of* $N \times 2^N \times 2^N$*, where* 2^N *denotes the power set of* N *(The elements of* J *will be called* justifications*).*
3. C *is a subset of* 2^N *(The elements of* C *will be called* nogoods*).*

Nodes represent a basic statement in agent's belief. Justifications represent a relation between those basic statements. Nogoods prohibit certain combinations of basic statements in agent's belief.

Let j be a justification of the form $\langle n, I, O \rangle$. Then, n is called the *consequent* node of j, I is called the *inlist* of j and O is called the *outlist* of j.

A justification means that if every node in its inlist is in agent's belief and if no node in its outlist is in her belief, then a consequent node is in her belief.

Let S be a subset of N. We say a justification $j = \langle n, I, O \rangle$ is *applicable* with respect to a state S if for every node $n^I \in I$, n^I belongs to S and there is no node $n^O \in O$ which belongs to S.

We would like to define a rational state of belief with respect to justifications and nogoods. In a rational state of belief, a node is believed (called *in*) if there must be some reason to believe the node, otherwise a node is not believed (called *out*). The following definition formalizes this intention.

Definition 2 Well-Founded Proof:
Let $D = \langle N, J, C \rangle$ be a TMS and S be a subset of N. A finite sequence of nodes $\langle n_1, ...n_k \rangle$ is called a well-founded proof *of n with respect to J in S if and only if the following conditions are satisfied.*

1. $n_k = n$.
2. *For every n_i in the sequence, n_i belongs to S.*
3. *For every n_i in the sequence, there is a justification $\langle n_i, I, O \rangle \in J$ such that for every node $n^I \in I$, n^I is one of $\langle n_1, ..., n_{i-1} \rangle$ and there is no node $n^O \in O$ which belongs to S.*

Note that well-founded proofs have next properties.

1. A prefix of a sequence of a well-founded proof is also a well-founded proof of the last element of the prefix.
2. A concatenation of well-founded proofs in S is also a well-founded proof in S.
3. For every well-founded proof p of n in S, there is a well-founded proof of n in S such that every node n in the proof occurs once at most. If a proof p is $\langle n_1, ..., n_i, ..., n_{j-1}, n_j, n_{j+1}, ..., n_k \rangle$ and n_i and n_j are same nodes, a proof $\langle n_1, ..., n_i, ..., n_{j-1}, n_{j+1}, ..., n_k \rangle$ is also a well-founded proof of n in S. We call this proof *normalized* proof of proof p.

4. For a well-founded proof $\langle n_1, ..., n_i, n_{i+1}, ..., n_{j-1}, n_j, n_{j+1}, ..., n_k \rangle$ in S, there is a justification j_j for n_j such that every node in its inlist is one of $\langle n_1, ..., n_{j-1} \rangle$ and no node in its outlist belongs to S. If every node in j_j's inlist is one of $\langle n_1, ..., n_i \rangle$, a sequence $\langle n_1, ..., n_i, n_j, n_{i+1}, ..., n_{j-1}, n_{j+1}, ..., n_k \rangle$ is also a well-founded proof in S.

The above definition prohibits believing a node by circular arguments such as "if p is in agent's belief, then she believes p."

Definition 3 Well-Founded Admissible State:
Let $D = \langle N, J, C \rangle$ be a TMS and S be a subset of N. Then S is a well-founded admissible state *if the following conditions are satisfied.*

1. *If there is an applicable justification $j = \langle n, I, O \rangle \in J$ w.r.t. S, $n \in S$.*
2. *For any nogood $c \in C$, c is not a subset of S.*
3. *Every node $n \in S$ has a well-founded proof with respect to J in S. (And, every node n has a normalized well-founded proof w.r.t. J in S.)*

The third condition above is somewhat different from the original definition in [Fujiwara89] and similar to the definition of finite groundedness in [Doyle83], but it is actually equivalent. The above condition is more direct to our intention because it states that if a node is believed, then there is a set of supporting justifications to the node without circular arguments.

Although this definition provides a condition to check if a given set S is a well-founded admissible state, it does not directly provide a non-deterministic algorithm to compute it. In this paper, we give such a procedure as follows.

A Non-deterministic Algorithm to Compute a Well-Founded Admissible State
Let $D = \langle N, J, C \rangle$ be a TMS and S_0 be $\emptyset$ (an empty set) and J_0 be $\emptyset$ and $i := 0$.

Step 1: If there is an applicable justification w.r.t. S_i which is not in J_i, go to Step 2, otherwise output S_i.

Step 2: Take one applicable justification $j_i = \langle n, I, O \rangle \notin J_i$ w.r.t. S_i.
$J_{i+1} := J_i \cup \{j_i\}$

$S_{i+1} := S_i \cup \{n\}$
Check if there is no nogood $c \in C$ such that c is a subset of S_{i+1} and every justification in J_{i+1} is applicable w.r.t. S_{i+1}. If so, $i := i + 1$ and go to Step 1, otherwise fail.

We say that a sequence of justifications $\langle j_1, ..., j_n \rangle$ *outputs* S in the above procedure if a sequence of justifications is $\langle j_1, ..., j_n \rangle$ when the above procedure outputs S.

Note that step 2 in the procedure involves a non-deterministic choice of applicable justification. Since a set of justifications is finite, we can try this procedure exhaustively with respect to choices of applicable justification. However, since in each loop, the number of justifications which can be checked is decreased by 1, we need an exponential time for an exhaustive try. If there is no output in the exhaustive search, then there is no well-founded admissible state.

Example 1 *(Computing a Well-Founded Admissible State)*

Let $D = \langle \{p, q\}, \{j_1, j_2\}, \emptyset \rangle$ where $j_1 = \langle p, \{p\}, \emptyset \rangle$ and $j_2 = \langle q, \emptyset, \{p\} \rangle$. Then there is only one well-founded admissible state $\{q\}$. In the above procedure, only the sequence $\langle j_2 \rangle$ outputs $\{q\}$. □

The following Theorem states soundness (the procedure outputs only well-founded admissible states) and completeness (if there is a well-founded admissible state, the procedure outputs by appropriate choices of justifications) of the procedure.

Theorem 1 *There is a sequence of justifications which outputs S in the procedure if and only if S is a well-founded admissible state.*

Proof:
(Soundness) Suppose there is a sequence of applicable justifications $\langle j_1, ..., j_i \rangle$ which outputs S in the procedure. We check S satisfies three conditions of well-founded admissible state. Let J_i be a set of (used) justifications when the procedure outputs S.

1. Suppose there is an applicable justification j w.r.t. S in J. This justification must be in J_i because there must be no applicable justification

w.r.t. S in $J - J_i$ when the procedure outputs S. From the construction of S, the consequent node of j is in S.

2. From the construction of S, no nogood $c \in C$ is a subset of S.

3. Let $\langle n_1, ..., n_i \rangle$ be a sequence of nodes each of which is a consequent node of $j_1, ..., j_i$ respectively. Then, this sequence of nodes is a well-founded proof of n_i. Since this sequence has every node in S and any prefix of the sequence is a well-founded proof of the last element, every node in S has a well-founded proof.

(Completeness) Suppose there is a well-founded admissible state S. Then every node n in S has a well-founded proof. We construct a normalized well-founded proof from well-founded proofs in the following algorithm.

Let $N_0 := \langle\rangle$ (empty sequence) and $i := 0$ and $S_0 := \emptyset$.
Repeat
Take a node $n_i \notin S_i$ from S.
Let a well-founded proof of n_i be $\langle n_i^1, ..., n_i^m \rangle$.
$N_{i+1} := N_i \cdot \langle n_i^1, ..., n_i^m \rangle$ ($\cdot$ is a concatenation operator.)
$S_{i+1} := S_i \cup \{n_i^1, ..., n_i^m\}$
$i := i + 1$
until $S_i = S$
Each N_i is a well-founded proof in S, let $\langle n^1, ..., n^m \rangle$ be a normalized proof of an output proof N_i of the above iteration. There is a justification j^k for each n^k such that every node in its inlist is one of $\langle n^1, ..., n^{k-1} \rangle$ and no node in its outlist belongs to S. Let J be a concatenation of a sequence of justifications $j^1, ..., j^m$ and any arbitrary sequence of (unused) applicable justifications w.r.t S which are not in $j^1, ..., j^m$.

Then we can easily check that J outputs S. □

Although the above procedure is correct, it has some redundancies. For example, if we decide n to be in, then we do not need to check any other justification whose consequent node is n. The following algorithm is due to Kurt Konolige. It reduces those redundancies.

An Enhanced Algorithm to Compute a Well-Founded Admissible State
Let $D = (N, J, C)$ be a TMS and $S_0 := \emptyset$, $T_0 := \emptyset$ and $i := 0$.

Step 1: Find an applicable justification $j = \langle n, I, O \rangle$ w.r.t. S_i such that n is not in S_i. If it is not found, output S_i else go to Step 2.

Step 2:

$S_i^0 := S_i \cup \{n\}$
$T_i^0 := T_i \cup O$
$k := 0$

Repeat

$$S_i^{k+1} := S_i^k \cup \{n | n \notin S_i^k \text{ and}$$
$$\exists j_i^k = \langle n, I, O \rangle \text{ s.t.}$$
$$(\forall n^I \in I \supset n^I \in S_i^k) \wedge (\forall n^O \in O \supset n^O \in T_i^k)\} \quad (1)$$

$$T_i^{k+1} := T_i^k \cup \{n | n \notin T_i^k \text{ and}$$
$$\exists c \in C \text{ s.t. } \forall n^c \in c(n^c \neq n \equiv n^c \in S_i^k)\} \quad (2)$$

If $S_i^{k+1} \cap T_i^{k+1} \neq \emptyset$, then fail. (3)

If $\exists c \subset C$ s.t. $c \subseteq S_i^{k+1}$, then fail.

$k := k + 1$

until ($S_i^k = S_i^{k-1}$ and $T_i^k - T_i^{k-1}$)

$S_{i+1} := S_i^k$
$T_{i+1} := T_i^k$
$i := i + 1$
Go to Step 1.

In the above procedure, T_i^k expresses a set of nodes which are decided to be out. Therefore, Operation (1) expresses an addition of nodes which are newly founded to be in. Operation (2) expresses an addition of nodes which are newly founded to be out. The node must be out by operation (2) because all nodes except the node in nogood c is in. Operation (3) checks if every used justification is still applicable.

Theorem 2 *The above procedure outputs S by appropriate choices of justifications if and only if S is a well-founded admissible state.*

Proof:

(Soundness) Same as soundness part of theorem 1.

(Completeness) Suppose there is a well-founded admissible state S. We construct a normalized proof $\langle n^1, ..., n^m \rangle$ in the same way of comleteness part in theorem 1. There is a justification j^k for each n^k such that every node in its inlist is one of $\langle n^1, ..., n^{k-1} \rangle$ and no node in its outlist belongs to S. By following way we can rearrange the proof, and, at the same time, the sequence of justifications. We move all nodes n with each used justification $j = \langle n, I, O \rangle$ after $\langle n^1, ..., n^i \rangle$ such that $\langle n^1, ..., n^i \rangle$ is the shortest sequence which satisfies next two conditions.

1. Every node in I is one of $\langle n^1, ..., n^i \rangle$.
2. Every node in O belongs to any outlist which is one of $\langle j^1, ..., j^i \rangle$, or the node belongs to any nogood such that each node in the nogood except the node is one of $\langle n^1, ..., n^i \rangle$.

Then we can check the rearranged sequence of justifications outputs S in the previous algorithm. □

Konolige gives further refinement for his algorithm. He regards a nogood c as a justification whose consequent node is $\perp$ (special node which expresses contradiction) and whose inlist is c and whose outlist is $\emptyset$. So, it is sufficient to consider only a set of justifications by this transformation. Also, in his refined algorithm, a set of justifications is changed every time when a state of a node is decided so that we can check if justification is applicable or not without checking states of nodes which has already been decided.

A Refined Algorithm to Compute a Well-Founded Admissible State
Let $D = \langle N, J, C \rangle$ be a TMS and $S_0 := \emptyset$, $T_0 := \{\perp\}$, $J_0 := J$ and $i := 0$.

Step 1: Find a justification $j = \langle n, \emptyset, O \rangle$ in J_i. If it is not found, output S_i else go to Step 2.

Step 2:
$S_i^0 := S_i \cup \{n\}$
$T_i^0 := T_i \cup O$
$J_i^0 := J_i$
$IN_i^0 := \{n\}$
$OUT_i^0 := O$
$k := 0$

Repeat

$IN_i^{k+1}, OUT_i^{k+1}, J_i^{k+1} := propagate(IN_i^k, OUT_i^k, J_i^k)$

$S_i^{k+1} := IN_i^k \cup S_i^k$

$T_i^{k+1} := OUT_i^k \cup T_i^k$

If $IN_i^{k+1} \cap T_i^{k+1} \neq \emptyset$ then fail.

$k := k + 1$

until ($IN_i^k = \emptyset$ and $OUT_i^k = \emptyset$)

$S_{i+1} := S_i^k$
$T_{i+1} := T_i^k$
$J_{i+1} := J_i^k$
$i := i + 1$
Go to Step 1.

$propagate(IN, OUT, J)$
begin
$NewIN := \emptyset$
$NewOUT := \emptyset$
$J' := \emptyset$
Repeat
Take a node n in IN.
For each $j = \langle n', I, O \rangle \in J$ **do**
If $n' = n$ or $n \in O$ then do nothing
else if $I = \{n\}$ and $O = \emptyset$ then add n' into $NewIN$
else if $n' = \perp$ and $I = \{n, n''\}$ then add n'' into $NewOUT$
else if $n \in I$ then add $j' = \langle n', I - \{n\}, O \rangle$ into J'
else add j into J'.
until (we check every node in IN)
$NewJ := \emptyset$
Repeat
Take a node n in OUT.
For each $j = \langle n', I, O \rangle \in J'$ **do**
If $n \in I$ then do nothing
else if $I = \emptyset$ and $O = \{n\}$ then add n' into $NewIN$
else if $n \in O$ then add $j' = \langle n', I, O - \{n\} \rangle$ into $NewJ$
else add j into $NewJ$.

until (we check every node in OUT)
Return $NewIN$, $NewOUT$, $NewJ$
end

This algorithm gives same output as the enhanced algorithm because each S_i^k and T_i^k is the same as the enhanced algorithm.

3 Implementing TMS in KL1

3.1 KL1

In this subsection, we briefly explain KL1, a concurrent logic programming language developed in ICOT [Chikayama88]. KL1 consists of the following form of guarded Horn clauses.

$$H_1 : -G_{11}, ..., G_{1g_1} | B_{11}, ..., B_{1b_1}.$$
$$H_2 : -G_{21}, ..., G_{2g_2} | B_{21}, ..., B_{2b_2}.$$
$$\vdots$$
$$H_n : \quad G_{n1}, ..., G_{ng_n} | B_{n1}, ..., B_{nb_n}.$$

where each H_i, B_{ik} is an atomic formula and we call them *head, body goal* of a guarded Horn clause respectively, and G_i is a built-in predicate such as unification or comparative operator and we call it *guard goal* of a guarded Horn clause. We call '|' between guard and body a *commit operator* and we call the part before the commit operator *guard part* and the part after it *body part*.

An execution of program is provoked to call the following form of goal.

$$? - B_1, ..., B_k.$$

Then, an execution proceeds as reducing this goal to further goals in parallel. One reduction of a goal B_i is done as follows. Firstly, we check if a guard part of a guarded Horn clause succeeds in matching B_i. This is done by unification. However, if unification exports variable bindings outside, it will be suspended. Guard part succeeds if every unification succeeds without exporting variable bindings outside. Then, we replace B_i by the body part of the clause. If we find two or more matched clauses, then we choose one of

them non deterministically and replace B_i. If there is no more goal, then it ends the execution.

By this procedural semantics, we can represent asynchronized concurrent processes in the language. Each goal B_i can be regarded as a process and a common variable over goals can be regarded as a process channel over those processes. This property is common among concurrent logic programming languages [Shapiro83].

3.2 Implementation

In this subsection, we discuss implement issues of Konolige's refined algorithm. It is very important to investigate how to represent objects of TMS in his algorithm. In a naive implementation, we use a list structure to represent a set of justifications where each justification is an element of the list. However, in the naive implementation, we have to check all the justification in the list to propagate a state of a node. If a number of justifications are μ and an average occurrences of a node in a set of justifications are η, then this implementation becomes inefficient as η/μ (occurrence rate for relevant justifications for a node) increases. And we also need to copy most part of the list to change justifications. This problem can be regarded as a kind of *frame problem* in that we can not change a large structure data efficiently in declarative logic language.

However, we can avoid this problem by changing a representation of justifications in KL1 or other similar concurrent logic languages. Our approach is to translate each node in a justification into processes and relevant nodes are connected each other by process channels represented as common variables so that when a state of node is decided, we can propagate this state through this channel to relevant justifications directly. Even if we could use indexing scheme to access the relevant part of the list, there is still a problem of changing the content of the list where we have to copy most of the list. The problem also can be avoided by this implementation because changing the state of the process corresponds with changing the justification and this change does not affect any irrelevant processes.

We explain this idea by an example.
Consider a TMS $D = \langle \{p, q, r, s\}, \{j_1, j_2, j_3\}, \emptyset \rangle$ where $j_1 = \langle p, \{q\}, \{r\} \rangle$, $j_2 = \langle q, \{\}, \{r\} \rangle$ and $j_3 = \langle s, \{r\}, \{\} \rangle$.

We firstly compile this justifications into processes (Figure 1). The consequent node of a justification is translated into a *con_node* process and nodes of its inlist are translated into *in_node* processes and nodes of its outlist are translated into *out_node* processes. Nodes in the same justifications are connected each other (horizontal arrows in the figure, we call them *justification channels*), and also the same nodes are connected each other (thick arrows in the figure, we call them *node channels*), and a control process and *con_node* processes are connected (vertical arrows in the figure, we call them *control channels*).

Firstly, the control process tries to find an applicable justification through control channels. In this example, j_2 is found and control process provokes j_2 and lets r out and registers r as out.

j_2 becomes $\langle q, \{\}, \{\}\rangle$ and q is found to be in and this fact is reported to control process through control channels. Then, information of r is propagated through node channels. j_1 becomes $\langle p, \{q\}, \{\}\rangle$ by removing r from its outlist. j_3 is removed because r is in its inlist (Figure 2). These deletions of nodes are accomplished by terminating corresponding processes.

Then, from a consequent node of j_2, information of q is propagated through node channels. j_1 becomes $\langle p, \{\}, \{\}\rangle$ and p is found to be in via justification channel and this fact is reported to control process (Figure 3).

And finally, j_1 is removed and execution is halted because there is no applicable justification and there is no contradiction(Figure 4)

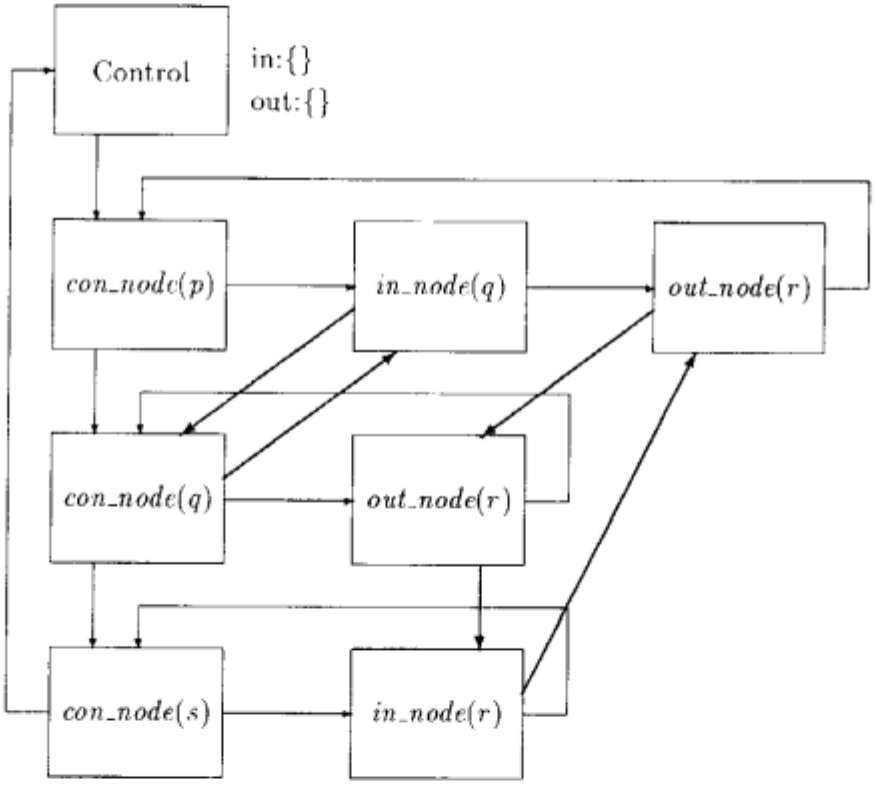

Figure 1: Initial State of Processes.

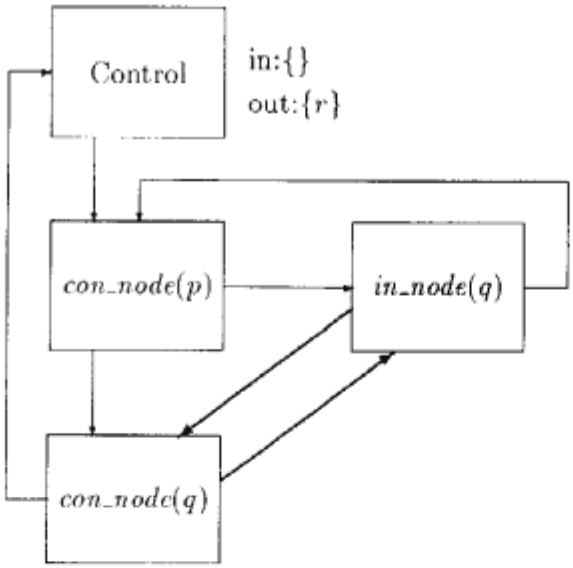

Figure 2: A State of Processes after Letting r out.

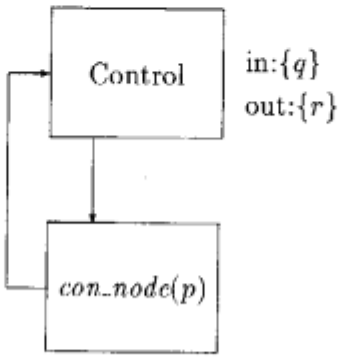

Figure 3: A State of Processes after Letting q in.

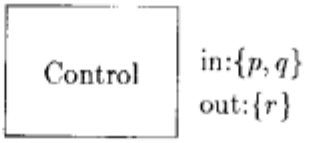

Figure 4: A Final State of Processes.

4 Preliminary Result

We compare our implementation of process representation of justifications with naive implementation of list representation of justifications in Konolige's algorithm.

We use a $\mathcal{N}$-queens problem. TMS representation $\langle N, J, C \rangle$ for $\mathcal{N}$-queens problem is as follows.

1. N consists of a position of queens.

$$N = \{q_{nm} | 1 \leq n \leq \mathcal{N}, 1 \leq m \leq \mathcal{N}\}$$

2. J consists of a set of the following rules.
$\langle q_{11}, \{\}, \{q_{12}, q_{13}, q_{14}\}\rangle$
$\langle q_{12}, \{\}, \{q_{11}, q_{13}, q_{14}\}\rangle$
$\langle q_{13}, \{\}, \{q_{11}, q_{12}, q_{14}\}\rangle$
$\langle q_{14}, \{\}, \{q_{11}, q_{12}, q_{13}\}\rangle$

$\vdots$

$\langle q_{44}, \{\}, \{q_{41}, q_{42}, q_{43}\}\rangle$

Each rule expresses that if a queen is not in the position in outlist, the queen must be in the position of consequent node.

3. C consists of a combination of positions of two queens such that those queens check each other (two queens are in the same column or the queens are put diagonally.

$\langle q_{11}, q_{21}\rangle$
$\langle q_{11}, q_{31}\rangle$
$\vdots$
$\langle q_{44}, q_{34}\rangle$
$\langle q_{11}, q_{22}\rangle$
$\langle q_{11}, q_{33}\rangle$
$\vdots$
$\langle q_{34}, q_{43}\rangle$

And we use a random function to choose an applicable justification. Since we use the same function for the implementation by process and the naive implementation by list, order to choose an applicable justification is same in both implementations.

Figure 5 shows the comparison. We measure an execution time to find the first solution for $\mathcal{N}$-queens problem by a simulator of KL1 on Symmetry Machine.

As $\mathcal{N}$ increases, effect of process representation increases. This is because the rate of relevant justifications for a node decreases as $\mathcal{N}$ increases and therefore, implementation by list has to do more checks which are irrelevant. However, we find the following defects for implementation by process.

1. Theoretically speaking, we should achieve an effect of an inverse of occurrence rate, that is, for example, the implementation by process should be about 100 times as fast as the implementation by list because occurrence rate is about 1%. However, the result shows the implementation by process is 7 times as fast as the implementation by list. Although we have not done close scrutiny of the implementation by process yet, a reason may be an overhead of process manipulation, that is, overhead of suspending processes and resuming processes.

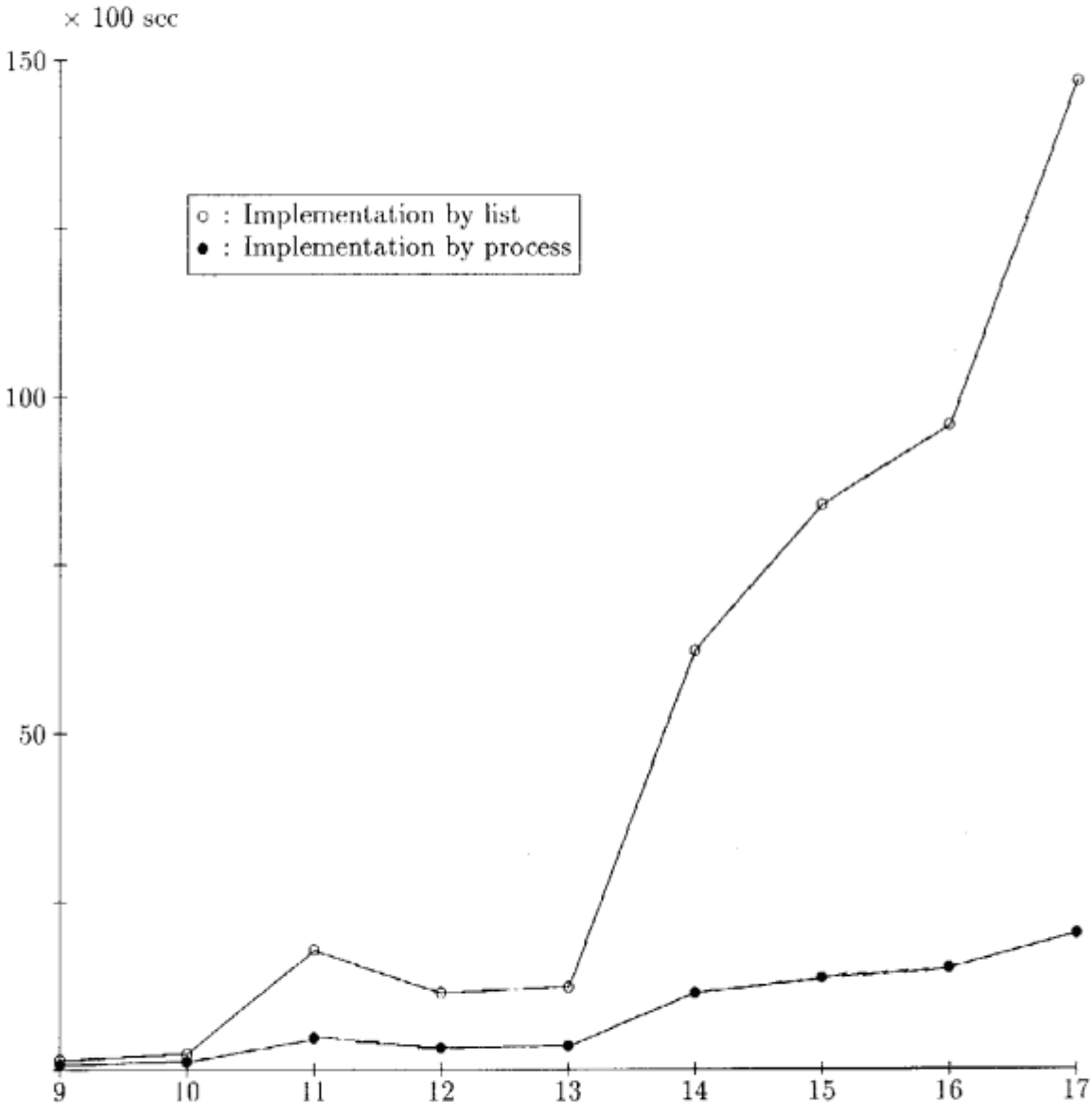

Figure 5: $\mathcal{N}$-queen

2. The implementation by processes need much more memory than the implementation by list because each node is represented by process instead of symbols. However, the rate of memory increase is constant.

5 Conclusion

We prove correctness for algorithms of calculating a well-founded state for TMS and provide an implementation by process representation of justification in KL1.

As future work, we are planning to do the following.

1. We would like to reduce overhead of process manipulation. This may be done by letting node processes in a justification into one process.

2. To parallelize the implementation, we execute the same problem in parallel with a different random function for each processor so that different search pass is explored. An analysis of probabilistic algorithm will be needed.

Acknowledgments

We are grateful to Katsumi Inoue and Nobuyuki Ichiyoshi from ICOT for helpful discussions on an earlier drafts of this paper.

References

[Chikayama88] Chikayama, T., Sato, H., and Miyazaki, T.: Overview of the Parallel Inference Machine Operating System(PIMOS), Proc. of Fifth Generation Computer Systems 1988 (FGCS'88), pp. 230 – 251 (1988).

[Doyle79] Doyle, J.: A Truth Maintenance System, *Artificial Intelligence*, **12**, pp. 231 – 272 (1979).

[Doyle83] Doyle, J.: The Ins and Outs of Reason Maintenance, *Proc. AAAI-83*, pp. 349 – 351 (1983).

[Fujiwara89] Fujiwara, Y. and Honiden, S.: Relating the TMS to Autoepistemic Logic, *Proc. IJCAI-89*, pp. 1199 – 1205 (1989).

[Genesereth87] Genesereth, M. R., Nilsson, N. J.: *Logical Foundations of Artificial Intelligence*, Morgan Kaufmann Publishers (1987).

[Reinfrank89] Reinfrank, M., Dressler, O. and Brewka, G.: On the Relation between Truth Maintenance and Autoepistemic Logic, *Proc. IJCAI-89*, pp. 1206 – 1212 (1989).

[Shapiro83] Shapiro, E. and Takeuchi, A.: Object Oriented Programming in Concurrent Prolog, *New Generation Computing*, **1**, pp. 25 – 48 (1983).