TR-571

A Forward-Chaining Multiple-Context Reasoner and Its Application to Logic Design

by

Y. Ohta & K. Inoue

July, 1990

A Forward-Chaining Multiple-Context Reasoner and Its Application to Logic Design

Yoshihiko Ohta and Katsumi Inoue
ICOT Research Center
Institute for New Generation Computer Technology
Mita Kokusai Bldg. 21F
1-4-28 Mita, Minato-ku, Tokyo 108, Japan
Phone:+81-3-456-2514
{ohta, inoue}@icot.jp

Abstract

This paper presents an extended production system architecture which can deal with forward reasoning in multiple contexts. The proposed architecture consists of a compiler of clauses and default rules into a RETE-like network, a RETE-based inference engine and the ATMS. The feature of the method is that the inference engine gives intermediate justifications to the ATMS and stores intermediate dependent assumptions of two-input nodes in the RETE-like network, allowing faster multiple-context reasoning. By means of this method, the multiple-context reasoner called APRICOT/0 has been implemented. An experimental result under the logic design knowledge base shows that APRICOT/0 is about 6 to 10 times faster than a system with a simple combination of a production system and the ATMS.

Keywords: RETE algorithm, ATMS, multiple-context reasoner, logic circuit design

1 Introduction

An optimum solution to a given knowledge base may be efficiently obtained by dealing with all the possible contexts simultaneously. A multiple-context reasoner is a useful tool for building AI systems such as design systems where optimum solutions are often required. It is related to hypothetical reasoning [7] and default reasoning [11]. However, multiple-context reasoners usually take a lot of time to obtain solutions because they need to maintain the consistency of multiple contexts.

Recently, several multiple-context reasoners [4, 8, 5] have been developed by using the assumption-based truth maintenance system (ATMS) [2, 3]. The ATMS is a general facility for determining all the possible contexts and maintaining the consistency of multiple contexts. Each of those multiple-context reasoners consists of the ATMS and a problem solver that includes the domain-dependent knowledge base and an inference engine.

On the other hand, production system architectures are widely used as rule-based problem solvers when the knowledge bases are expressed as sets of rules. The RETE algorithm [6] was developed for use in production system interpreters. It is an efficient method for matching a large collection of objects with many conjunctive patterns. The production system usually reasons in a single context because the inference engine chooses a rule from applicable rules.

In order to use a production system as a problem solver for a multiple-context reasoner, the knowledge base should be extended to include *default rules*. Each default rule may be applied by making a corresponding assumption as long as it is consistent, regardless of application of other default rules, and thus allowing for multiple contexts. A multiple-context reasoner by [8] is a forward-chaining system with the ATMS and uses the RETE algorithm. However, this system separates the ATMS from the RETE network completely, so that there are some overheads between pattern matching in its RETE network and the ATMS manipulation.

This paper presents a new combination method between a forward-chaining inference engine and an ATMS for allowing faster multiple-context reasoning. The feature of the proposed combination method is that the inference engine gives intermediate *justifications* to the ATMS and stores the *labels* of intermediate ATMS nodes at two-input nodes in the RETE-like network. This method allows an efficient *label-updating process* for consistency maintenance as well as conjunctive pattern matching for the inference engine. By means of this method, the multiple-context reasoner which is called APRICOT/0 has been implemented on the PSI-II (Personal Sequential Inference) machine [10] in ESP (Extended Self-contained Prolog) [1]. The efficiency of APRICOT/0 is confirmed by an experimental comparison with a conventional multiple-context reasoner through a logic circuit design problem that is an example of multiple-context reasoning.

2 Input knowledge base

An input knowledge base of the multiple-context reasoner discussing in this paper, is defined with F which is a finite set of Horn clauses and D which is a set of normal default rules.

A Horn clause is a formula,

$$\alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow \beta \tag{1}$$

or

$$\alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow \bot, \tag{2}$$

where $\alpha_1, \cdots, \alpha_n (n \geq 0)$ and β represent atomic formulas, $\bot$ represents false, $\alpha_1 \wedge \cdots \wedge \alpha_n$ are called antecedents, and β is called a consequent. When $n \neq 0$, a Horn clause (1) is written in the following notation:

$$ID :: \alpha_1, \cdots, \alpha_n - > \beta. \tag{3}$$

When $n = 0$, a Horn clause (1) is written as:

$$\beta. \tag{4}$$

In this case, β is restricted to a ground atomic formula. A Horn clause (2) is written in the following notation:

$$ID :: \alpha_1, \cdots, \alpha_n - > [\,]. \tag{5}$$

Here, *ID* is a name identified with the clause.

A normal default rule [11] is an inference rule,

$$\frac{\alpha : \beta}{\beta}, \tag{6}$$

where α is restricted to a finite conjunction $\alpha_1 \wedge \cdots \wedge \alpha_n (n \geq 0)$ of atomic formulas, and β is restricted to an atomic formula. When $n \neq 0$ in (6), it is written as:

$$ID :: \alpha_1, \cdots, \alpha_n - > assume(\beta). \tag{7}$$

Here, *ID* is a name identified with the default rule. When $n = 0$ in (6), it is written as:

$$assume(\beta). \tag{8}$$

In (8), β is restricted to a ground atomic formula.

Procedure calls such as input/output built-in procedures and numerical calculations can be attached between the symbols " ::" and "$-$ >" of each notation (3);(5);(7). A collection of procedure calls is parenthesized between "{" and "}" . These procedure calls are evaluated by built-in and user-defined procedures.

3 ATMS

In the ATMS, the basic data structure called an *ATMS node* is the following:

$$\gamma_{datum} :< datum, label, justifications > .$$

Here, the datum is a ground formula given by the inference engine. The notation $assume(a(1))$ represents that a datum $a(1)$ is justified by an assumption $a(1)$. Here, the ATMS node corresponding to the assumption $a(1)$, distinguished from $\gamma_{a(1)}$, is represented $\Gamma_{a(1)}$. The inference engine gives the corresponding justification to the ATMS:

$$\Gamma_{a(1)} \Rightarrow \gamma_{a(1)}.$$

A justification corresponds to a Horn clause where the antecedents are conjunctions of ground atomic formulas and the consequent is a ground atomic formula.

A set of assumptions is called an *environment*. A *context* is the set formed by assumptions of a consistent environment combined with all data which can be derived from those assumptions. An inconsistent environment is called a *nogood*. A datum is said to hold in an environment if it can be derived from environments and the current set of Horn clauses corresponding to justifications.

Each datum is labeled with a minimal set of environments in which the datum holds. The minimal set of environments is called a *label* of the node corresponding to the datum. The following is a label-computing algorithm for the consequent node when a justification is given by the inference engine.

1. Supposing that the label of the consequent node is L and the label of the ith antecedent node are $L_i (i = 1, \cdots, n)$, the ATMS computes the following:

   $$L' = \{\bigcup_{i=1}^{n} E_i | E_i \in L_i\}.$$

   A set L'' which is the set removing nogoods and subsumed environments from $L' \cup L$, becomes the new label of the consequent node. If each environment is represented by bit-vectors, then the union of two sets can be computed by the OR operation of the two bit-vectors.

2. If $L = L''$, then this process is finished.

3. If the consequent node is $\gamma_{[\,]}$, then all new nogoods are removed from the label of every node.

4. If this node is not $\gamma_{[\,]}$, then this label-updating process propagates all consequent nodes of this node.

4 Multiple-context reasoner

An architecture of a multiple-context reasoner which inputs a knowledge base written in the notations shown in Section 2 and outputs all derivable ground atomic formulas with assumptions where the atomic formulas hold, is considered. The multiple-context reasoner consists of a compiler of clauses and default rules into a RETE-like network, a forward-chaining inference engine and the ATMS. The inference engine reasons in *multiple contexts* provided by the ATMS without conflict resolution of applying antagonistic inference rules and gives *justifications* to the ATMS. The ATMS maintains the consistency of the knowledge base with given justifications in multiple contexts.

We will describe two methods to combine a forward-chaining inference engine with the ATMS by considering the following knowledge base as an example:

$$F = \{k :: a(X), b(Y), c(Z) -> d(X, Y, Z)\}$$

and

$$D = \{assume(a(1)), assume(b(2)), assume(c(3)), assume(c(4))\}.$$

4.1 Simple method to combine an inference engine with the ATMS

Figure 1 shows a configuration of a multiple-context reasoner, where a forward-chaining inference engine is simply combined with the ATMS. This system is called a simple combination system(SCS). The following are the ATMS nodes corresponding to the elements of D :

$$\gamma_{a(1)} :< a(1), \{\{\Gamma_{a(1)}\}\}, \{(\Gamma_{a(1)})\} >,$$

$$\gamma_{b(2)} :< b(2), \{\{\Gamma_{b(2)}\}\}, \{(\Gamma_{b(2)})\} >,$$

$$\gamma_{c(3)} :< c(3), \{\{\Gamma_{c(3)}\}\}, \{(\Gamma_{c(3)})\} >$$

and

$$\gamma_{c(4)} :< c(4), \{\{\Gamma_{c(4)}\}\}, \{(\Gamma_{c(4)})\} > .$$

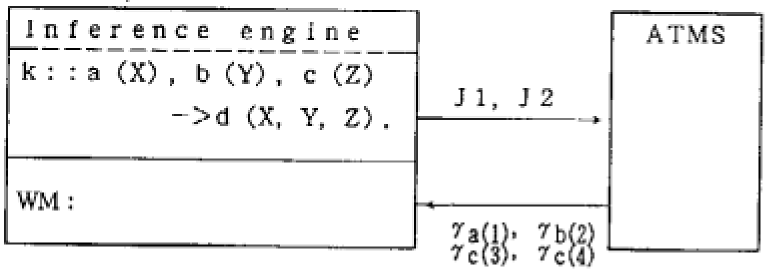

Inference engine
k : : a (X) , b (Y) , c (Z) ->d (X, Y, Z) .
WM :
J 1, J 2
ATMS
γa(1), γb(2) γc(3), γc(4)

Fig. 1 Configuration of a simple combination between the inference engine and the ATMS.

In the initial state, there are eight ATMS nodes with assumption nodes in the ATMS. The inference engine finds two ground instantiations,

$$\theta_1 = \{X := 1, Y := 2, Z := 3\}$$

and

$$\theta_1 = \{X := 1, Y := 2, Z := 4\},$$

of the clause k with believed data $a(1), b(2), c(3)$ and $c(4)$ provided by the ATMS. When the inference engine derives two ground atomic formulas $d(1,2,3)$ and $d(1,2,4)$ from the believed data and clauses $k\theta_1$ and $k\theta_2$, it gives the following justifications to the ATMS:

$$J1 : \gamma_{a(1)}, \gamma_{b(2)}, \gamma_{c(3)} \Rightarrow \gamma_{d(1,2,3)}$$

and

$$J2 : \gamma_{a(1)}, \gamma_{b(2)}, \gamma_{c(4)} \Rightarrow \gamma_{d(1,2,4)}.$$

Therefore, the ATMS creates two ATMS nodes,

$$\gamma_{d(1,2,3)} :< d(1,2,3), \{\{\Gamma_{a(1)}, \Gamma_{b(2)}, \Gamma_{c(3)}\}\}, \{(\gamma_{a(1)}, \gamma_{b(2)}, \gamma_{c(3)})\} >$$

and

$$\gamma_{d(1,2,4)} :< d(1,2,4), \{\{\Gamma_{a(1)}, \Gamma_{b(2)}, \Gamma_{c(4)}\}\}, \{(\gamma_{a(1)}, \gamma_{b(2)}, \gamma_{c(4)})\} > .$$

There are four union operations, $\{\Gamma_{a(1)}\} \cup \{\Gamma_{b(2)}\} \cup \{\Gamma_{c(3)}\}$ and $\{\Gamma_{a(1)}\} \cup \{\Gamma_{b(2)}\} \cup \{\Gamma_{c(4)}\}$, in these label-updating tasks of the ATMS.

4.2 Method to combine two-input nodes with the ATMS

A multiple-context reasoner implemented by means of the following method, is called APRICOT/0. The inference engine reasons by flowing tokens in the RETE-like network. Each token is an ATMS node corresponding to the ground formula. Horn clauses written in the notation (4) and normal default rules written in the notation (8) are passed to the inference engine. The inference engine has a queue for tokens. In the initial setting, the ATMS nodes corresponding to the clauses such as the notation (4) and the default rules such as notation (8) are added to the queue.

Horn clauses written in the notations (3),(5) and normal default rules written in the notation (7) are compiled into a RETE-like network. The RETE-like network consists of a root node, one-input nodes, two-input nodes and terminal nodes. These RETE nodes are described here:

- A root node has one slot:
  - Successors: A set of pointers to all one-input nodes.
- A one-input node has four slots:
  - Pattern: $freeze(\beta)$.
  - Successors: A set of pointers to two-input nodes.
  - Terminals: A set of pointers to terminal nodes, if they exist.
  - WM: A distributed working memory which is a set of ATMS nodes corresponding to ground atomic formulas $\beta\sigma$ where σ are ground instantiations of β.
- A two-input node has five slots:
  - Pattern: $freeze(\alpha \wedge \beta)$, where α is a conjunction of atomic formulas or an atomic formula and β is an atomic formula.
  - Successors: A set of pointers to two-input nodes.
  - Terminals: A set of pointers to terminal nodes, if they exist.
  - WM: A distributed working memory which is a set of ATMS nodes corresponding to ground formulas $(\alpha \wedge \beta)\lambda\sigma$ where λ are ground instantiations of α and σ are ground instantiations of β.

  - Predecessors: A set of pointers to the one-input node corresponding to β and the two-input (or one-input) node corresponding to α.

- A terminal node $\#ID$ works as a module executing the following tasks for a clause or a default rule named ID when tokens arrive at this node.
  - It evaluates the attached procedure calls.
  - It gives justifications to the ATMS.
  - It adds new derived ATMS nodes to the queue.

Here, $freeze(\alpha)$ is a ground formula substituting new constants(for example, the constant is \$) into all variables included in α.

Figure 2 shows the RETE-like network corresponding to the example and the configuration of combinations between two-input nodes and the ATMS.

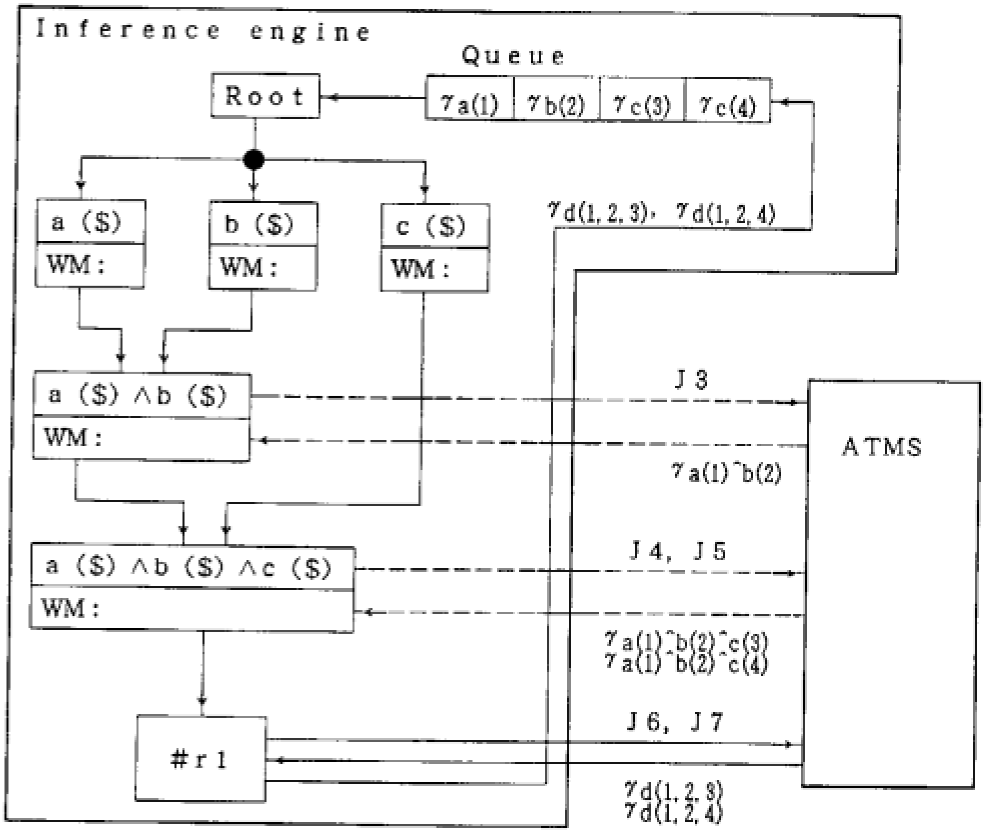


Fig. 2 Configuration of combinations between the two-input nodes and the ATMS.

In this method, two-input nodes in the RETE-like network for the inference engine give intermediate justifications to the ATMS and store the intermediate ATMS nodes. When a token $\gamma_{\alpha\lambda}$, where λ is a ground instantiation of α, arrives at a two-input node whose the pattern is $freeze(\alpha \wedge \beta)$, the inference engine gives the following intermediate justification to the ATMS at the two-input node:

$$\gamma_{\alpha\lambda}, \gamma_{\beta\sigma} \Rightarrow \gamma_{(\alpha\wedge\beta)\lambda\sigma}.$$

Here, σ is a ground instantiation of β. The ATMS node $\gamma_{\beta\sigma}$ can be obtained by referring to the distributed working memory (WM) through the predecessor link. After updating the label, the ATMS returns the ATMS node $\gamma_{(\alpha\wedge\beta)\lambda\sigma}$ to the two-input node. When the two-input node receives the ATMS node, this ATMS node is added to this WM of the two-input node.

Those behaviors are explained with the example in detail. The initial state of the reasoner is also shown in Figure 2. The inference engine starts the reasoning by taking the first token $\gamma_{a(1)}$ from the queue. The token arrives at all one-input nodes through the root node. In the one-input node $a(\$)$, for example, the frozen pattern is melted so that the pattern becomes a formula $a(A)$ substituting new variable A for the symbol \$. The unification between the melted pattern $a(A)$ and the datum $a(1)$ of the token is tried. As the melted pattern $a(A)$ is unifiable with the datum $a(1)$, the WM of the one-input node stores the token. The token $\gamma_{a(1)}$ is passed to the successor $a(\$) \wedge b(\$)$.

The inference engine takes the next token $\gamma_{b(2)}$ from the queue. This token is stored in the WM of the one-input node $b(\$)$, then flows into the successor $a(\$) \wedge b(\$)$. The two-input node, which gets an element $\gamma_{a(1)}$ by referring to the WM of another predecessor $a(\$)$, gives the following intermediate justification to the ATMS:

$$J3 : \gamma_{a(1)}, \gamma_{b(2)} \Rightarrow \gamma_{a(1)\wedge b(2)}.$$

The ATMS computes $\{\Gamma_{a(1)}\} \cup \{\Gamma_{b(2)}\}$ at the step 1 of label-updating process. When the label-updating process is finished, the ATMS creates the ATMS node,

$$\gamma_{a(1)\wedge b(2)} :< a(1) \wedge b(2), \{\{\Gamma_{a(1)}, \Gamma_{b(2)}\}\}, \{(\gamma_{a(1)}, \gamma_{b(2)})\} > .$$

The token $\gamma_{a(1)\wedge b(2)}$ corresponding to this intermediate ATMS node is stored in the WM of the two-input node, then is passed to the successor $a(\$) \wedge b(\$) \wedge c(\$)$.

The inference engine takes the next token $\gamma_{b(3)}$ from the queue. This token is stored in the WM of the one-input node $b(\$)$, then is passed to the successor $a(\$) \wedge b(\$) \wedge c(\$)$. The two-input node, which has obtained an element $\gamma_{a(1)\wedge b(2)}$ by referring to the WM of another predecessor $a(\$) \wedge b(\$)$, gives the following intermediate justification to the ATMS:

$$J4 : \gamma_{a(1)\wedge b(2)}, \gamma_{c(3)} \Rightarrow \gamma_{a(1)\wedge b(2)\wedge c(3)}.$$

The ATMS computes $\{\Gamma_{a(1)}, \Gamma_{b(2)}\} \cup \{\Gamma_{c(3)}\}$ in the label-updating process. Note that the cost for an OR operation is unaffected by the number of assumptions by using the bit-vectors. When the label-updating process is finished, the ATMS creates the ATMS node,

$$\gamma_{a(1)\wedge b(2)\wedge c(3)} :< a(1) \wedge b(2) \wedge c(3), \{\{\Gamma_{a(1)}, \Gamma_{b(2)}, \Gamma_{c(3)}\}\}, \{(\gamma_{a(1)\wedge b(2)}, \gamma_{c(3)})\} >,$$

which is stored in the WM of the two-input node. The token $\gamma_{a(1)\wedge b(2)\wedge c(3)}$ is then passed to the terminal node #k since the two-input node has the terminal node. The terminal node #k derives $d(1,2,3)$ and gives the following justification to the ATMS:

$$J5 : \gamma_{a(1)\wedge b(2)\wedge c(3)} \Rightarrow \gamma_{d(1,2,3)}.$$

The ATMS creates the ATMS node,

$$\gamma_{d(1,2,3)} :< d(1,2,3), \{\{\Gamma_{a(1)}, \Gamma_{b(2)}, \Gamma_{c(3)}\}\}, \{(\gamma_{a(1)\wedge b(2)\wedge c(3)})\} > .$$

The terminal node adds the new ATMS node $\gamma_{d(1,2,3)}$ to the queue.

The inference engine takes the next token $\gamma_{c(4)}$ from the queue. In the same way as the token $\gamma_{c(3)}$, the new token is stored in the WM of the one-input node $c(\$)$, then the two-input node $a(\$) \wedge b(\$) \wedge c(\$)$ gives the following intermediate justification to the ATMS:

$$J6 : \gamma_{a(1)\wedge b(2)}, \gamma_{c(4)} \Rightarrow \gamma_{a(1)\wedge b(2)\wedge c(4)}.$$

The ATMS computes $\{\Gamma_{a(1)}, \Gamma_{b(2)}\} \cup \{\Gamma_{c(4)}\}$ in the label-updating process, and creates the ATMS node,

$$\gamma_{a(1)\wedge b(2)\wedge c(4)} :< a(1) \wedge b(2) \wedge c(4), \{\{\Gamma_{a(1)}, \Gamma_{b(2)}, \Gamma_{c(4)}\}\}, \{(\gamma_{a(1)\wedge b(2)}, \gamma_{c(4)})\} >,$$

which is stored in the WM of the two-input node. The token $\gamma_{a(1)\wedge b(2)\wedge c(4)}$ is passed to the terminal node #k. The terminal node #k derives $d(1,2,4)$ and gives the following justification to the ATMS:

$$J7 : \gamma_{a(1)\wedge b(2)\wedge c(4)} \Rightarrow \gamma_{d(1,2,4)}.$$

The ATMS creates the ATMS node,

$$\gamma_{d(1,2,4)} :< d(1,2,4), \{\{\Gamma_{a(1)}, \Gamma_{b(2)}, \Gamma_{c(4)}\}\}, \{(\gamma_{a(1)\wedge b(2)\wedge c(4)})\} > .$$

The terminal node add the new ATMS node $\gamma_{d(1,2,4)}$ to the queue.

The inference engine takes the next token $\gamma_{d(1,2,3)}$ from the queue, but this token is not unifiable one for all one-input nodes. The last token $\gamma_{d(1,2,4)}$ is the same as $\gamma_{d(1,2,3)}$. As the queue becomes empty, the inference engine stops. The reasoner outputs the ATMS nodes $\gamma_{d(1,2,3)}$ and $\gamma_{d(1,2,4)}$.

There are three union operations for two sets in all the label-updating tasks in the ATMS. The number of union operations is less than in the simple combination system when a one-input node passes to its successor, which is a successor of another two-input node. When the RETE-like network includes a two-input node shared by some clauses or default rules, we can see that the system with combinations between two-input nodes and the ATMS is also efficient.

4.3 Complexity considerations

The following is a more general complexity consideration for these architectures.

Assume that the number of environments of an ATMS node $< \alpha_i\theta_i^k, \ldots >$ is represented by $N(\alpha_i\theta_i^k)$. The sum of the numbers of the environments is defined as:

$$S(\alpha_i) \equiv \Sigma_k N(\alpha_i\theta_i^k), \tag{9}$$

where each ground substitution θ_i^k is applicable to α_i.

On SCS, the total number of the OR operation for label-updating for the clause (3) is:

$$C_s = (n-1)\prod_{i=1}^{n} S(\alpha_i). \tag{10}$$

On APRICOT/0, the total number of the OR operation for label-updating for the clause (3) is:

$$C_a = \sum_{i=2}^{n}\{S(\bigwedge_{j=1}^{i-1} \alpha_j) \cdot S(\alpha_i)\}. \tag{11}$$

Let $1 \leq j \leq n$: then

$$S(\bigwedge_{i=1}^{j} \alpha_i) \leq \prod_{i=1}^{j} S(\alpha_i) \tag{12}$$

because the right hand side may include nogoods. To analyze the worst case, assume that the set of nogoods is empty. Then,

$$\frac{C_a}{C_s} = \frac{\frac{1}{S(\alpha_3)\cdot S(\alpha_4)\cdots S(\alpha_n)} + \frac{1}{S(\alpha_4)\cdot S(\alpha_5)\cdots S(\alpha_n)} + \cdots + \frac{1}{S(\alpha_n)} + 1}{n-1}. \tag{13}$$

Hence, $C_a < C_s$ when there is at least one α_i $(i = 3, 4, \cdots, n)$ such that $S(\alpha_i) > 1$.

Suppose the input knowledge base includes a clause,

$$ID' :: \alpha_1, \cdots, \alpha_n, \alpha_{n+1} --> \beta, \tag{14}$$

which has all antecedents and the consequent of the clause (3) and contains an additional antecedent α_{n+1}. If labels of $< \beta\theta, \ldots >$ derived from the clause (3) have been computed by the ATMS, the total number of the OR operation for label-updating for the clause (14) is:

$$C'_a = S(\bigwedge_{j=1}^{n} \alpha_j) \cdot S(\alpha_{n+1}). \tag{15}$$

On SCS, the total number of the OR operation for label-updating for the clause (14) is:

$$C'_s = n \cdot \{\prod_{i=1}^{n} S(\alpha_i)\} \cdot S(\alpha_{n+1}). \tag{16}$$

Assuming that the set of nogoods is empty,

$$\frac{C'_a}{C'_s} = \frac{1}{n}. \tag{17}$$

Hence, $C_a < C_s$ when the RETE-like network includes at least one two-input node shared by some clauses or default rules.

5 Application to logic design

This section demonstrates that APRICOT/0 is an efficient system in multiple contexts through an application to a logic circuit design problem. In design problems such as logic design, many alternatives appear on different levels of a hierarchy. APRICOT/0 can deal with those alternatives as assumptions generated dynamically (by rules of the form (7)).

The design of logic circuits for calculating the greatest common divisor (GCD) of two integers expressed in 8 bits by using the Euclidean algorithm, is taken as an example of a logic design problem. The circuits have to satisfy given constraints on time and area. Constraints on area are expressed as inequalities in the basic cell count. Constraints on time are expressed as inequalities in the delay.

5.1 Representing the design knowledge

The knowledge base contains several kinds of knowledge: datapath design, component design, technology mapping, CMOS standard cells, and area and time constraints.

- Datapath design: The default rule *datapath_1* represents that the circuit for calculating the GCD can be constructed from the following components: a control box, an inverter, two registers, two multiplexers, a comparator and a subtracter with two multiplexers.

$$\begin{aligned}&\mathit{datapath\_1} :: \mathit{control\_box}(X1, N1),\ \mathit{inverter}(X2, N2),\\ &\quad \mathit{register}(X3, N3),\ \mathit{register}(X4, N4),\ \mathit{multiplexer}(X5, N5),\\ &\quad \mathit{multiplexer}(X6, N6),\ \mathit{comparator}(X7, N7),\\ &\quad \mathit{subtracter\_with\_MUX}(X8, N8) -> \\ &\quad \mathit{assume}(\mathit{calculator\_of\_GCD}([X1, X2, X3, X4, X5, X6, X7, X8],\\ &\qquad\qquad N1 + N2 + N3 + N4 + N5 + N6 + N7 + N8)).\end{aligned}$$

- Component design: The default rule *component_design_1* represents that a subtracter can be constructed with an adder and a one's complement.

$$\begin{aligned}&\mathit{component\_design\_1} :: \mathit{adder}(X1, N1),\ \mathit{ones\_complement}(X2, N2)\\ &\quad -> \mathit{assume}(\mathit{subtracter}([X1, X2], N1 + N2)).\end{aligned}$$

- Technology mapping: The clause *technology_mapping_1* represents that an 8-bit adder can be constructed with two 4-bit carry-lookahead-adder cells (*a4h*) connected serially.

$$\mathit{technology\_mapping\_1} :: \mathit{cell}(a4h, N1) -> \mathit{adder}([a4h, a4h], N1 * 2).$$

- CMOS standard cells: The clause *cell(a4h,50)* represents that an a 4-bit carry-lookahead-adder cell (*a4h*) is a CMOS standard cell whose the basic cell count is 50.

- Area and time constraints:

- Constraints evaluated at the final step in each context: The clause *area_constraints* represents the knowledge about evaluating area constraints.

$$\begin{aligned}&\mathit{area\_constraints} :: \mathit{calculator\_of\_GCD}(X, N),\ \mathit{area\_limit}(M),\\&\quad \{N > M\} -> [\,].\end{aligned}$$

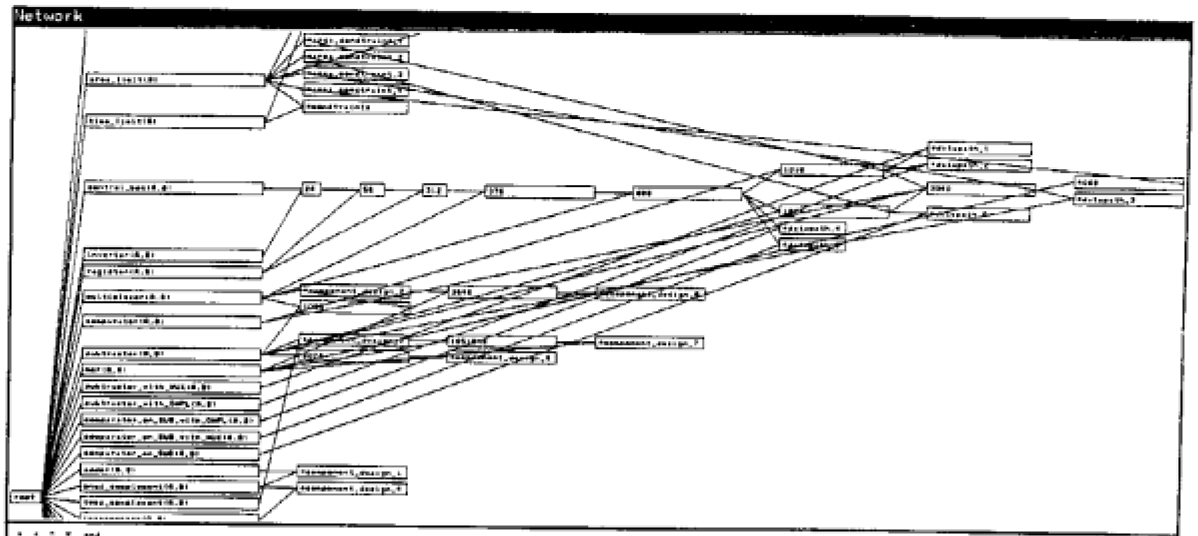
  The clause *time_constraints* represents the knowledge about evaluating time constraints.

$$\begin{aligned}&\mathit{time\_constraints} :: \mathit{calculator\_of\_GCD}(X, N),\ \mathit{time\_limit}(M),\\&\quad \{: \mathit{delay}(\#\mathit{simulator}, X, T), T > M\} -> [\,].\end{aligned}$$

- Area constraints evaluated earlier: To avoid redundant executions of problem-solving tasks which are shared among multiple contexts in the ATMS, the knowledge base includes the knowledge for area constraints evaluated earlier. These constraints make earlier pruning of inconsistent contexts possible.

$$\begin{aligned}&\mathit{area\_constraint\_1} :: \mathit{control\_box}(X1, N1),\ \mathit{inverter}(X2, N2),\\&\quad \mathit{register}(X3, N3),\ \mathit{register}(X4, N4),\ \mathit{multiplexer}(X5, N5),\\&\quad \mathit{multiplexer}(X6, N6),\ \mathit{comparator}(X7, N7),\ \mathit{area\_limit}(M),\\&\quad \{N1 + N2 + N3 + N4 + N5 + N6 + N7 > M\} -> [\,].\end{aligned}$$

There are 44 clauses and 10 default rules in the design knowledge base. The RETE-like network is partially shown in Figure 3.



- Boxes with melted patterns denote 1-input nodes.
- Boxes with numbers denote 2-input nodes that have no terminal node.
- Boxes with terminal nodes denote 2-input nodes that have a terminal node.

Fig. 3 Partial diagram of the RETE-like network corresponding to the design knowledge base.

5.2 Experiment and Result

The design process is processed by two reasoners, SCS and APRICOT/0, in the bottom-up manner. First, all CMOS standard cells are mapped onto all possible subcomponents. Second, all generated subcomponents are mapped onto all possible components. The components are generated as assumptions. Third, all generated components are mapped onto all possible datapaths after the area constraints on basic cell counts of partial combinations of the components have been evaluated. The datapaths are generated as assumptions. Finally, the datapaths are evaluated by both constraints. The datapaths that satisfy the constraints become the solutions.

The result of the reasoning by the multiple-context reasoner is a set of all solutions that satisfy all given constraints. Therefore, we can pick the best solution of all the solutions.

Both APRICOT/0 and SCS have been implemented on the PSI-II machine in ESP. Table 1 shows the result of the experiment concerning reasoning times. When the basic cell count limit is 500 or 400, APRICOT/0 is about 6 times faster than SCS.

The reason why APRICOT/0 works efficiently for this logic design problem is the following. First, there are many alternatives for each component such as a comparator and a subtracter. For example in Figure 3, many tokens are passed from the one-input node *comparator*(A, B) to its successor "1016", that is also a successor of "888". Second, in this problem, the design knowledge is expressed in a hierarchy. At each level of the hierarchy, especially in datapath design, there are many knowledge whose components are shared by other knowledge. Therefore, the resulting RETE-like network contains lots of two-input nodes shared by several clauses or default rules. Third, area constraints can be evaluated incrementally, and thus make earlier pruning possible. When an ATMS node stored in the WM of a two-input node is no longer believed (that is, its label becomes empty), the ATMS node does not have to be stored any more and does not have to be passed to the successors. Therefore, when the basic cell count limit is 300, this earlier evaluation of constraints works extremery well, so that APRICOT/0 is about 10 times faster than SCS.

Table 1 Comparisons between whole reasoning times on APRICOT/0 and data on the simple combination system (SCS).

Circuit constraints		Number of	Reasoning times		Speed−up
Area limit [cell count]	Time limit [ns]	solutions	APRICOT/0 Ta [s]	SCS Ts [s]	Ts/Ta
500	60	47	17. 1	106. 1	6
400	50	24	16. 0	96. 6	6
300	40	2	3. 7	36. 1	10
300	60	6	3. 7	36. 3	10

6 Conclusion

We have shown a new architecture for multiple-context reasoners, which reduces overheads between pattern matching in a RETE network and the ATMS processing.

The experimental result shows that APRICOT/0 is about 6 to 10 times faster than SCS under the logic design knowledge base. The cost of whole label computations by APRICOT/0 is less than one by SCS in the following cases:

- When tokens are passed from a one-input node to its successor that is a successor of another two-input node.
- When the RETE-like network includes a two-input node shared by some clauses or default rules.

Acknowledgments
The problem domain of the logic design was suggested to the authors by Mr. Kenji Ikoma. The design knowledge of the logic circuit has been adapted from [9]. The implementation of the design knowledge base was greatly assisted by the continued efforts of Mr. Hiroshi Ohsaki and Mr. Makoto Nakashima. The authors would like to thank Dr. Koichi Furukawa for his useful comments. Finally, they would like to express their appreciation to Dr. Kazuhiro Fuchi, Director of the Institute for New Generation Computer Technology, who provided them with the opportunity to conduct this research.

References

[1] Chikayama, T., Unique Features of ESP, *Proceedings of the International Conference on Fifth Generation Computer Systems*, (1984), pp.292-298.

[2] de Kleer, J., An Assumption-based TMS, *Artificial Intelligence*, **28**, (1986), pp.127-162.

[3] de Kleer, J., Extending the ATMS, *Artificial Intelligence*, **28**, (1986), pp.163-196.

[4] de Kleer, J., Problem Solving with the ATMS, *Artificial Intelligence*, **28**, (1986), pp.197-224.

[5] Flann, N. S., Dietterich, T. G. and Corpron, D. R., Forward Chaining Logic Programming with the ATMS, *Proceedings of AAAI-87*, (1987), pp.24-29.

[6] Forgy, C. L., Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, *Artificial Intelligence*, **19**, (1982), pp.17-37.

[7] Inoue, K., Problem Solving with Hypothetical Reasoning, *Proceedings of the International Conference on Fifth Generation Computer Systems*, **3**, (1988), pp.1275-1281.

[8] Junker, U., Reasoning in Multiple Contexts, *GMD Working Paper No.334*,(1988).

[9] Maruyama, F., Kakuda, T., Masunaga, Y., Minoda, Y., Sawada, S. and Kawato, N., co-LODEX: A Cooperative Expert System for Logic Design, *Proceedings of the International Conference on Fifth Generation Computer Systems*, **3**, (1988), pp.1299-1306.

[10] Nakashima, H. and Nakajima, K., Hardware Architecture of the Sequential Inference Machine: PSI-II, *Proceedings of the Symposium on Logic Programming*, (1987), pp.104-113.

[11] Reiter, R., A Logic for Default Reasoning, *Artificial Intelligence*, **13**, (1980), pp.81-132.