Logic Circuit Synthesis Preserving Correctness using ET Rules

HIROSHI YOSHIKAWA
Hokkaido University
Faculty of Computer Science
North 11, West 5, Sapporo
JAPAN

KIYOSHI AKAMA
Hokkaido University
Information Initiative Center
North 11, West 5, Sapporo
JAPAN

HIROSHI MABUCHI
Iwate Prefectural University
Faculty of Software and Information Science
152-52 Sugo, Takizawa, Iwate
JAPAN

Abstract: A transformation–based method of circuit synthesis is proposed. A correct circuit is synthesized as a description similar to a netlist from a specification program, which is written in an ET program consisting of ET (equivalent transformation) rules. The circuit and the specification (the initial ET program) are equivalent finite state machines. The main part of the synthesis is equivalent transformation of ET programs preserving their procedural meaning. A technique for merging ET rules in an ET program is introduced to reduce the size of the circuit to be synthesized.

Key–Words: Logic Circuit Synthesis, ET Rules, Equivalent Transformation, Rewriting Rules

1 Introduction

A method for synthesis of digital circuits by using the ET [6] language which is based on a mathematical theory of the equivalent transformation is discussed. It is only a theoretical approach to synthesize a correct circuit from a specification, and so can not be used as a verification method to test correctness of circuits. It is located in the first step of the approach. The final goal is to transform a specification given by declarative descriptions and to obtain a circuit description at a gate level.

In the present LSI development process, it is necessary to verify HDL [1] descriptions by using a lot of simulations. A long test period and high costs are spent for those simulations in the development process. However, it is still difficult to guarantee that there is no bug due to the fact that simulation patterns that can be actually simulated are limited to a very small coverage. Therefore all possible patterns can not be covered. Still, the development process can not help but rely on simulations because it is not guaranteed that a procedure described with HDL is theoretically correct to a specification.

In the ET framework a procedure described with ET rules is correct to a specification. Thus, a correct circuit description can be theoretically made and quality of development improve if ET is used as a circuit description language. Moreover, research in the making of an ET program from a specification is still ongoing. As a result we can expect simulations to become unnecessary, development period to become shorter and correctness to be guaranteed theoretically. So our aim is to establish ET–based circuit development.

Preserving correctness is the most important key when a correct specification written in ET is transformed into a correct netlist. In this paper, a transformation–based method of circuit synthesis that preserves correctness is proposed, and some techniques for transforming ET rules into digital circuits are illustrated.

[1]HDL = Hardware Description Language. VHDL and Verilog HDL are de facto standards.

2 ET Rule

An ET rule is a rewriting rule which transforms a clause to another equivalent one. An ET program consists of a set of ET rules and is based on ET (*Equivalent Transformation*) computation model which guarantees correctness of computations. A clause is rewritten one after another by an ET program and eventually becomes a unit clause which is a constituent of a *meaning* of the program.

ET is also one of the declarative programming languages such as Prolog [4], CHR [5], etc., in which the correctness of their computations can be easily proven. Each ET rule is a correct component and does not depend on others in a program. These features contribute a great deal to the development of large systems without bugs, and support incremental program construction [7].

Moreover, an ET program can be transformed into another equivalent one. This technique is useful in the synthesis of circuits from ET programs.

3 Description

This section shows how to synthesize a circuit from an ET program by using an easy example '*factorial*'.

3.1 Factorial ET Program

To compute the factorial of n by an ET program, it is necessary to give a query clause of the form shown below:

$$fact(n, X) \leftarrow subfact(n, 1, X). \tag{1}$$

The term n is a positive integer given by the user and the term X is a variable for which the answer is substituted. This clause represents the logical expression that $fact(n, X)$ is true when $subfact(n, 1, X)$ is true. $fact(n, X)$ and $subfact(n, 1, X)$ are called *atoms* which represent relations of the terms in their arguments. For example, the atom $fact(n, X)$ signifies that X equals the factorial of n.

The following is a correct ET program that computes the factorial.

$$\begin{aligned} &subfact(N, M, F), \{N \neq 0\} \\ &\quad \Rightarrow \{K := N - 1,\ L := N \times M\}, \\ &\quad\quad subfact(K, L, F). \end{aligned} \tag{2}$$

$$subfact(N, M, F), \{N = 0\} \Rightarrow \{F := M\}. \tag{3}$$

This program consists of two ET rules – (2) a tail recursion rule and (3) a termination rule. These rules represent that the atom $subfact(N, M, F)$ in the left hand side of the arrow can be replaced by the equivalent atom in the right hand side. The capital letters F,K,L,M and N represent variables. The expressions $\{N \neq 0\}$ and $\{N = 0\}$ in the left hand side are called *conditional parts*. Each rule can be applied if and only if the conditional part is satisfied. The expression $\{K := N - 1, L : N \times M\}$ and $\{F := M\}$ in the right hand side are called *execution parts*, and are executed when the rule is applied.

For example, the computation process of the factorial of 3 by the program {(2),(3)} is shown below.

$$fact(3, X) \leftarrow subfact(3, 1, X).$$

$\Downarrow$ apply rule (2)

$$fact(3, X) \leftarrow subfact(2, 3, X).$$

$\Downarrow$ apply rule (2)

$$fact(3, X) \leftarrow subfact(1, 6, X).$$

$\Downarrow$ apply rule (2)

$$fact(3, X) \leftarrow subfact(0, 6, X).$$

$\Downarrow$ apply rule (3)

$$fact(3,\ 6) \leftarrow .$$

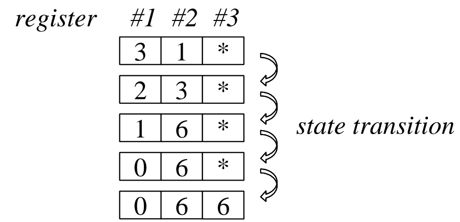

Figure 1: State Transition of Factorial

The rule(2) is applicable if and only if the conditional part is satisfied, $i.e.$ $N \neq 0$. Similarly the rule(3) is applicable if and only if $N = 0$. So the first three transitions are caused by the rule(2) and the last transition is caused by the rule(3).

The last clause in the process is a unit clause. It means that $fact(3, 6)$ is unconditionally true. All clauses in the computation process are mutually equivalent. Therefore the computation process by the ET program is regarded as a proof process.

We can define a *meaning* of an ET program, which consists of atoms deducted by the program. For example the meaning of the factorial program is the following.

$$\mathcal{M}_{fact} = \{fact(0, 1), fact(1, 1), fact(2, 2), \cdots\} \tag{4}$$

A correct circuit of the factorial calculates a value m which satisfies $fact(n, m) \in \mathcal{M}_{fact}$ for any given number n.

3.2 Basic Idea

An ET computation model is considered as a state transition model. A clause represents a certain state and an ET rule rewrites it to another. For instance the argument of $subfact$ changes whenever the rule is applied, or the rule does the role to change a state. This state transition is concisely expressible with registers as shown in Figure 1. The *register #1, #2 and #3* in the figure correspond to three arguments of $subfact$.

In this way, every clause can be expressed by using registers. So computation equivalent to an ET program can be executed on a certain *Finite State Machine*. In this view, we employ the *Moore FSM* [2] shown in Figure 2 to perform the equivalent state transition that a correct ET program does. The Moore FSM is a well known FSM and consists of three elements:

- **Flip Flop** is a memory element. It transmits the next state D to the current state Q on every rising edge of clock signal, and keeps the state Q during clock cycle even if the input value D changes.

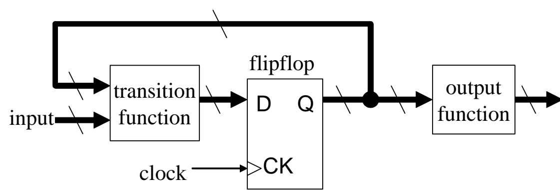

Figure 2: Moore Finite State Machine

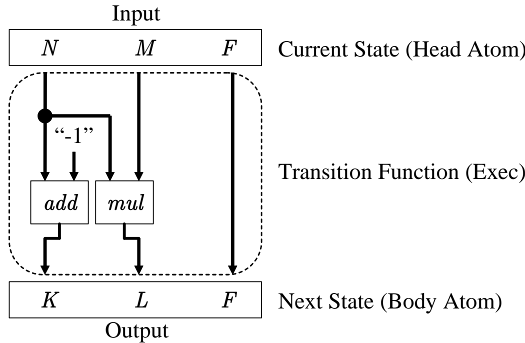

Figure 3: Transition Circuit Block Diagram corresponding to ET Rule (2)

- **Transition Function** calculates the next state from the current state Q and user input, and outputs it to D.
- **Output Function** calculates output values from the current state and outputs them to output ports of the circuit.

Any desired finite state machine can be obtained by deciding the transition function and the output function appropriately.

Then, how should the transition function and the output function be decided? An ET rule is considered to be a state transition rule as previously stated. The head atom of an ET rule represents the current state and a body atom represents the next state. Arguments of the atom are assigned to registers of FSM. The execution part of the rule can be considered as a partial transition function on condition that the conditional part of the ET rule is filled. Figure 3 shows these concepts. Each partial transition function is made from each execution part. If all the partial transition functions are combined, it becomes a complete transition function. Generally an execution part of an ET rule is a partial transition function. It calculates a next state candidate. A complete transition function circuit is made by combining all the partial transition functions with switch circuits. The switch circuit selects an appropriate state among the candidates. The switch is controlled by the conditional part. Figure 4 shows

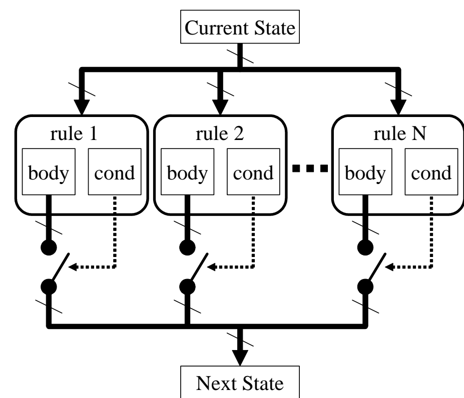

Figure 4: General Transition Function Circuit

this concept. But our proposed method does not need any switches. We merge ET rules into one instead of adding switches. This allows an optimization over the rules.

An output function only passes the values in some registers to output ports. There is nothing special to consider. For instance the output function of the factorial circuit is only wire which connects the output of the register #3 to the output port of the circuit.

3.3 Modify Rules to Fit Practical Circuit

To make the circuit practicable, a mechanism to inform of a computation end and a mechanism to hold a computation result are added.

First of all, the current ET program {(2),(3)} should be transformed to a new ET program {(5),(6),(7)} by adding an end flag.

$$subfact(\mathbf{0}, N, M, F), \{N \neq 0\} \Rightarrow \{K := N-1,\ L := N \times M\}, subfact(\mathbf{0}, K, L, F). \quad (5)$$

$$subfact(\mathbf{0}, N, M, F), \{N = 0\} \Rightarrow \{F := M\}, \boldsymbol{subfact(1, N, M, F)}. \quad (6)$$

$$\boldsymbol{subfact(1, N, M, F)} \Rightarrow . \quad (7)$$

Rules (5) and (6) are derived from rules (2) and (3) respectively by the following operations.

1. Insert a flag **0** in the head of the argument in each $subfact$.
2. Add an atom to the body of the termination rule. This atom is the same as the head atom except

that the flag is changed to **1**. The rule then becomes a recursive rule.

Rule(7) is a new termination rule. It is applied only when the computation ends. A query clause to the new ET program {(5),(6),(7)} is the following.

$$fact(n, X) \leftarrow subfact(0, n, 1, X). \quad (8)$$

Though proof is omitted, the computation of query(8) by the program {(5),(6),(7)} is equivalent to the computation of query(1) by the program {(2),(3)} because the state transition is the same by excluding the end flag. That is, the meaning of this program is the same as $\mathcal{M}_{fact}$ defined by equation (4).

Rule(7) is then modified as follows so that the circuit can hold a result.

$$subfact(1, N, M, F) \Rightarrow \boldsymbol{subfact(1,N,M,F)}\ . \quad (9)$$

The program becomes interminable. However the computation of the program is regarded as terminable one because the atom $subfact(1, N, M, F)$ is considered as a *terminal state*. The above modification enables the circuit to hold an answer permanently.

3.4 Merge ET Rules into One

Rules (5),(6) and (9) can be merged into one. We only have to select a substitution expression from each execution part by using the evaluation result of each conditional part. This modification is useful for synthesizing the transition function and optimizing it.

$$\begin{aligned}
&subfact(I,N,M,F)\\
&\Rightarrow \{B1 := (I == 0)\ and\ (N \neq 0),\\
&\quad B2 := (I == 0)\ and\ (N == 0),\\
&\quad J := if\ \{B1\}\ 0\ else\ if\ \{B2\}\ 1\ else\ 1,\\
&\quad K := if\ \{B1\}\ (N-1)\\
&\qquad else\ if\ \{B2\}\ N\\
&\qquad else\ N,\\
&\quad L := if\ \{B1\}\ (M \times N)\\
&\qquad else\ if\ \{B2\}\ M\\
&\qquad else\ M,\\
&\quad H := if\ \{B1\}\ F\ else\ if\ \{B2\}\ M\ else\ F\\
&\},\\
&subfact(J,K,L,H). \quad (10)
\end{aligned}$$

An appropriate expression is chosen by $B1$ and $B2$ [2], where $B1$ and $B2$ are evaluation results of the conditional parts of rules(2) and (3) respectively. An evaluation result of rule(9) is not necessary because rule(9) is applicable if and only if all other rules are not applicable.

The merged rule(10) is quite the equivalent of the program {(5),(6),(9)}. That is, the execution part of rule(10) shows the complete transition function of the circuit.

3.5 Break Down into Bit Operations

If the width of the bit of each variable and data is given, then execution parts can be expressed by logical expressions.

The width of the bit at the interface is decided by the specification. There are a variety of techniques for deciding the width of the bit of internal variable and data, such as the fixed bit method,the inference method, etc. However we don't mention the technique because it is not the essence of this paper. Here, let's assume that the bit width is decided as follows.

$$\begin{aligned}
&subfact(I_{[1]},N_{[2]},M_{[3]},F_{[3]})\\
&\Rightarrow \{B1_{[1]} := (I_{[1]} == 0_{[1]})\ and\ (N_{[2]} \neq 0_{[2]}),\\
&\quad B2_{[1]} := (I_{[1]} == 0_{[1]})\ and\ (N_{[2]} == 0_{[2]}),\\
&\quad J_{[1]} := if\ \{B1_{[1]}\}\ 0_{[1]}\ else\ if\ \{B2_{[1]}\}\ 1_{[1]}\ else\ 1_{[1]},\\
&\quad K_{[2]} := if\ \{B1_{[1]}\}\ (N_{[2]} - 1_{[2]})\\
&\qquad else\ if\ \{B2_{[1]}\}\ N_{[2]}\\
&\qquad else\ N_{[2]},\\
&\quad L_{[3]} := if\ \{B1_{[1]}\}\ (M_{[3]} \times N_{[2]})\\
&\qquad else\ if\ \{B2_{[1]}\}\ M_{[3]}\\
&\qquad else\ M_{[3]},\\
&\quad H_{[3]} := if\ \{B1_{[1]}\}\ F_{[3]}\ else\ if\ \{B2_{[1]}\}\ M_{[3]}\ else\ F_{[3]}\\
&\},\\
&subfact(J_{[1]},K_{[2]},L_{[3]},H_{[3]}). \quad (11)
\end{aligned}$$

Suffix $[n]$ shows that the data is n bit. This rule is a special rule that limits the range of the data that rule(10) accepts. Each argument of $subfact$ is assigned to a register. Each register consists of flip flops of the amount corresponding to its suffix.

When the width of the bit is given to data, the execution part can be expressed by boolean operators such as AND, OR, NOT, etc. For example, because a comparison expression $v := (a_{[n]} \neq b_{[n]})$ is equivalent to the equation $v = (a_1 \oplus b_1) + \cdots + (a_n \oplus b_n)$, [3] the comparison can be expressed by logical atoms as follows:

$$\begin{aligned}
&xor(a_1, b_1, X_1), xor(a_2, b_2, X_2), ..., xor(a_n, b_n, X_n)\\
&, or(X_1, X_2, V_2), or(V_2, X_3, V_3), ..., or(V_{n-1}, X_n, v)
\end{aligned}$$

[2] The expression $(if\{e\}A\ else\ B)$ returns A if $e = 1$ and returns B otherwise.

[3] There exists at least one pair such as $a_i \neq b_i$ when $a_{[n]} \neq b_{[n]}$. The expression $(a_i \neq b_i)$ equivalent to $(a_i \oplus b_i)$.

Similarly the expression $v := (a_{[n]} == b_{[n]})$ can be expressed as follows:

$$xor(a_1, b_1, X_1), xor(a_2, b_2, X_2), ..., xor(a_n, b_n, X_n)$$
$$, or(X_1, X_2, V_2), or(V_2, X_3, V_3), ..., or(V_{n-1}, X_n, Y)$$
$$, not(Y, v)$$

In case of the expression $k_{[n]} := if\ \{e\}\ a_{[n]}\ else\ b_{[n]}$, it is equivalent to the equation $k_i = (e \cdot a_i) + (\overline{e} \cdot b_i)$ for each bit i. So the equivalent expression is:

$$and(e, a_i, A_i), not(e, X), and(X, b_i, B_i), or(A_i, B_i, k_i)$$

operations such as addition, subtraction and multiplication can be expressed by atoms, too. There are a variety of ways to express those operations though details are omitted. [4] These transformations of expressions and operations are carried out systematically. Consequently rule(11) becomes as follows:

$$\begin{aligned}
&subfact([i_1], [n_2, n_1], [m_3, m_2, m_1], [f_3, f_2, f_1]) \\
&\Rightarrow \{xor(i_1, 0, X1), not(X1, Y1), \\
&\quad xor(n_2, 0, X2), xor(n_1, 0, X3), or(X2, X3, Y2), \\
&\quad and(Y1, Y2, B1), \\
&\qquad \cdots \ll \text{syncopated} \gg \cdots \\
&\quad and(B2, m_3, Z3), not(B2, U), and(U, f_3, W3), \\
&\quad or(Z3, W3, V3), \\
&\quad and(B1, f_3, P3), not(B1, S), and(S, V3, Q3), \\
&\quad or(P3, Q3, h_3), \\
&\quad \}, \\
&subfact([j_1], [k_2, k_1], [l_3, l_2, l_1], [h_3, h_2, h_1]). \qquad (12)
\end{aligned}$$

Thus, the execution part is expressed only by the boolean, and the number of flip flops is determined. In this case we need nine flip flops. Also this expression shows the connection between the logic gates. It means that the execution part of this rule represents a *netlist*. That is, it is a circuit description which is executable by ET.

3.6 Simple Optimization

The transition function circuit can be synthesized from rule(12). However this circuit has a lot of inefficient gates. For example, the gate $xor(i_1, 0, X1)$ is inefficient because $X1$ always equals i_1. Consequently the circuit needs simple optimization by using ET, which is the rewriting rule. Here are some ET rules for optimization:

$$xor(A, 0, C) \Rightarrow \{C := A\}.$$
$$xor(A, 1, C) \Rightarrow not(A, C).$$
$$xor(A, A, C) \Rightarrow \{C := 0\}.$$
$$\vdots$$

Moreover, ET allows the multi head rewriting rule, and the following rewriting rules are possible:

$$\begin{aligned}
¬(A, N), not(N, X) \Rightarrow \{X := A\}, not(A, N). \\
¬(A, N), and(A, N, X) \Rightarrow \{X := 0\}. \\
¬(A, N), not(B, M), and(N, M, X) \\
&\qquad \Rightarrow or(A, B, Y), not(Y, X), not(A, N), not(B, M). \\
&\qquad\qquad \vdots
\end{aligned}$$

In this manner ET rules can be used for simple optimization of the circuit. Stronger optimization is also possible by program transformation on ET. [5] However, it is omitted because it is another study.

Finally the rule shown below is obtained.

$$\begin{aligned}
&subfact([i_1], [n_2, n_1], [m_3, m_2, m_1], [f_3, f_2, f_1]) \\
&\Rightarrow \{not(i_1, Y1), or(n_2, n_1, Y2), and(Y1, Y2, B1), \\
&\quad or(n_2, n_1, Y3), or(i_1, Y3, Y4), not(Y4, B2), \\
&\qquad \cdots \ll \text{syncopated} \gg \cdots \\
&\quad \}, \\
&subfact([j_1], [k_2, k_1], [l_3, l_2, l_1], [h_3, h_2, h_1]). \qquad (13)
\end{aligned}$$

The transition function circuit synthesized from this program is more compact than the one synthesized from the program {(5),(6),(9)}.

3.7 How to Start the Computation

The computation starts when an initial value is set to each register. The initial value is determined by the query clause. For example, the computation of factorial of 3 starts when each register is set in 0, 3, 1 and X (don't care).

For this purpose, it is possible to add a supplemental circuit that sets the initial values to registers. However, the kind of circuit that is suitable is selected on a case by case basis.

4 Discussion

The ET program that can be treated by this method consists of only recursive rules r_r and termination

[4] RCA, CLA, CSA, Wallace Tree and Booth's Algorithm are well known methods.

[5] The ET program can be transformed into another ET program that is completely equivalent.

rules r_t of the following forms.

$$r_r : p(t), \{cond_r(t)\} \Rightarrow \{exec_r(t,s)\}, p(s).$$
$$r_t : p(t), \{cond_t(t)\} \Rightarrow \{exec_t(t)\}.$$

p represents a predicate. Each t and s represents a sequence of terms. $exec_r(t,s)$ executes both the substitution of θ_r and the calculation of $s = f(t)$. $exec_t(t)$ executes only the substitution of θ_t. Many programs can be described by the rules of this class even though the class is very limited. Also, it seems that it is also possible to transform a variety of ET rules into the above–mentioned class by program transformation. Therefore, practically the ET rule of this class is sufficient to use.

The ET program and the circuit execute an equivalent state transition. That is the reason why the correct circuit is synthesized from the correct ET program. The correctness of the computation of the circuit is proven by using a certain mapping from the registers to the clause. However, proof was omitted in this paper.

The selection of ET rules in the execution of programs is usually entrusted to the processing system. The selection mechanism of ET rules should be synthesized together for making circuits. To achieve this, the technique of merging ET rules is introduced. As a result, the selection mechanism of ET rules becomes a portion of the execution part of the rule. Moreover, optimization including the selection mechanism of ET rules becomes possible.

If the range of the input is at most finite numbers, the ET program can be partially evaluated and may become a set of only termination rules. In that case, there is no state transition in the computation and flip flop may become unnecessary. Consequently it may be possible that the entire circuit that contains flip flops can be easily optimized when using program transformation.

In order to develop a large circuit, smaller circuit blocks are usually combined. However it is difficult to describe a large circuit directly in this technique because the class of ET rules is limited. Therefore, it is necessary to transform it into a class that this method can treat. The large class of ET rules will be explored in future research.

5 Conclusion

To synthesize a correct circuit from a correct ET rule, the method for synthesis of a circuit was proposed. This is equivalent to a very small class of ET programs. Using the proposed method, the ET rule is transformed into a description similar to a netlist of logic gates. Also, optimization over two or more rules is achieved by the merging rules technique. Correctness is theoretically guaranteed due to the fact that the rule transformations in this method preserve meanings of ET programs. Circuit synthesis from correct ET rules was also illustrated.

References:

[1] Hoe, J.C. and Arvind, *Operation–centric hardware description and synthesis,* Computer–Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 23 (9). 1277–1288.

[2] Edward F. Moore *Gedanken–experiments on Sequential Machines,* Automata Studies, Annals of Mathematical Studies, no.34, Princeton University Press, Princeton, N. J., pp. 129–153, 1956.

[3] A.D.Booth, *A Signed Binary Multiplication Technique,* The Quarterly Journal of Mechanics and Applied Mathematics 1951 4(2):236-240

[4] Lloyd,J.W., *Foundations of Logic Programming,* Second Edition, Springer–Verlag, 1987.

[5] Frühwirth,T., *Theory and Practice of Constraint Handling Rules,* Journal of Logic Programming, Special Issue on Constraint Logic Programming, 37(1–3): pp.95–138, 1998.

[6] K.Akama, E.Nantajeewarawat, and H.Koike, *Program Synthesis Based on the Equivalent Transformation Computation Model,* Proceedings of 12th International Workshop on Logic Based Program Development and Transformation (LOPSTR 2002), pp. 285–304, 2002.

[7] K.Akama, E.Nantajeewarawat, H.Koike and K.Miura, *The Squeeze Method - A Method for Program Construction in the Equivalent Transformation Computation Model,* Proceedings of the 6th international conference on intelligent technologies (InTech'05),pp. 198–206, 2005

[8] E.Nantajeewarawat, and K.Akama, *State–Transition Computation Models and Program Correctness Thereom,* Proceedings of the 6th international conference on intelligent technologies (InTech'05), pp. 277–286, 2005.