http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 5 (2): 305-313, 2006
ISSN 1812-5638

Turing Model for Distributed Computing

Saeed ur Rahman Khan and Malik Sikander Hayat Khiyal
Department of Computer Science, International Islamic University, Islamabad, Pakistan

Abstract: Turing machine is model for general-purpose computing device. There are many implementations of Turing Machine using physical computer as well as many implementations of physical computing device using Turing Machine. In this study, we elaborate the simulation of computing model, using a high-level language as its native code, on Turing machine. We also make it possible to communicate among many of such simulations. Finally we present a deterministic Turing model that actually simulates distributed processing.

Key words: Turing machine, physical machine, simulations

INTRODUCTION

Turing Machine is a mathematical model of computing device. Turing Machine[1], as defined by Alan Turing in his historical study, is a 7 tuple as following:

$$TM = \{Q, \Sigma, \Gamma, \delta, S, B, f\}$$

Where, the symbols are sequentially the set of states, the input symbols, the tape symbols, the set of transition functions, the starting state, the blank symbol and finally the set of final states.

There are many simulations of computer using Turing Machine and simulation of Turing Machine using computer to prove them equivalent[2-4]. Here, we elaborate them. This study describes a computing model having many addressable identities as basic unit, Turing Machine. Then the complete Turing Machine makes it possible to make communications among these models.

Thus in future, we shall use only TM as the formal representation of what can be computed by any kind of computing device[4].

Our proposed model is a compound Turing Machine of many simple Turing Machines. We now describe systematically the characteristics of our model.

In modern computing model, different parts of a computing model is again a computing model themselves having their own addressable identities.

There are many kinds of physical computing devices, such as mini, micro, mainframe, super scalar etc. These may be general purpose and special purpose. Computing models may be physical and virtual (as software on other physical model). But the virtual models can also be described as a big algorithm, collection/sequence of small physically implemented algorithms.

In spite of these facts, there is no fundamental difference between data processing, storage and communication between any kinds of computing model mentioned above[5].

In a fully distributed environment, there can be global addressing systems to address a small component of a native node participating in the whole system[6]. There may be many mechanism adopted for this purpose.

This study describes using Turing Machine:
A computer with

- Infinity long sequence of words each with an address
- Infinity long address space.
- Program of computer is stored in some word of memory.

Each word or box of tape represents a simple token of Higher-level language (Unlike the models described in existing books that refers instruction of a memory as in the machine or assembly language.)

MODEL IN BRIEF

$$TM = \{Q^n, \Sigma, \Gamma, \delta, S, B, f\}$$

States: Q^n. The states are compound states of the states of its sub Turing Machines. So we may say states are of the following types: $q_1q_2q_3...q_n$ where, $q \in Q$.

Σ is set of input alphabets.
Γ is set of tape alphabets.
δ is set of moves. The moves are also compound moves of the following formats:

Corresponding Author: Dr. Malik Sikander Hayat Khiyal, Department of Computer Science, International Islamic University, Islamabad, Pakistan

$$\delta(p_1,[(a_1,m_1),w_1,x_1])(q_1,[(b_1,m_1),y_1,z_1],[D,D,D])_1$$
$$\delta(p_2,[(a_2,m_2),w_2,x_2])(q_2,[(b_2,m_2),y_2,z_2],[D,D,D])_2$$
$$\delta(p_3,[(a_3,m_3),w_3,x_3])(q_3,[(b_3,m_3),y_3,z_3],[D,D,D])_3$$
$$\vdots \qquad \vdots \qquad \vdots$$
$$\delta(p_n,[(a_n,m_n),w_n,x_n])(q_n,[(b_n,m_n),y_n,z_n],[D,D,D])_n$$

Where, $a,b,m,w,x,y,z \in \Gamma, q_1, q_2 \in Q, D \in \{L,R,N\}$

S is the starting state for all sub Turing Machines of the compound Turing Machine.
B is the blank symbol.
f is the final accepting state. But when all the sub Turing Machines goes to accepting state then the final Turing Machine is assumed to be at the final state. So the final state of the whole Turing Machine is $f_1 f_2 f_3 \ldots f_n$

FEATURES

Our aim is to make the Turing machine that having the following features:

- The TM is designed in so that its functionality will be nearest possible of real computer while remaining within the limitation of TM, the input code of TM (which is also the executable code) will be like that of Higher level Language code. Not all the functionality of Higher Level Language are discussed, but some of them to limit the complexity.
- Each token in the Higher Level Language Language will occupy first tuple of a block in the program tape, which the head will read at once along with the block number (token number).
- The TM is to simulate the computing functions like those are in a distributed environment.
- In literatures, a non-deterministic Turing Machine is known to be a parallel Turing machine. But in some other places it is shown that it is not actually simulation of a parallel processing with which we agree. We disagree with the statement that non deterministic Turing Machine to be a simulator of distributed computing environment, because of the following reasons

  In non-deterministic environment where there are several moves for a given configuration, the Turing Machine starts moving in all paths and thus creates branches only at that particular configuration.

  The number of branches only depends on the number of moves available from that given configuration, otherwise not. If there come a multiple path configuration at any point, it will again branch in the number of moves those are available from that configuration.

  In actual distributed environment, the distributed processors (nodes) are not created by instructions. (In thread creation, there is actually no physical parallel processing occurs. Rather the created threads share the time of same processing node. Even if they use different node the different nodes are previously created by some other mechanism.)

  In distributed environment, the nodes are mostly previously statically created. If not then processing nodes are created on availability basis but not because of any instruction.

  We produce a deterministic Turing Machine having all the configurations with one move besides acting as distributed model.
- Our Turing Machine has each component as an independent computing model and those perform their tasks independently and while needed communicate with the other model. The main reason to unite the individual components and make the compound Turing Machine is, the components of the compound Turing Machine can communicate among themselves only not out of those. This will simulate the physical interconnected nodes those can communicate and make a distributed system. The components those are exchanging data among themselves can be ports of a single CPU or different CPUs connected by communication ports. But all of those should have an address.
- We have represented the reserve word tokens in this model with the help of Greek alphabets. But after implementation any other suitable token can be used in these places.

FUNCTIONALITIES

Computing machine has the following functions:

Variable definition/storage retrieval: To reduce the complexity, there are only two types of variable declarations here.

- Variables.
- Reference to other variable or reference.

Assignment operations: Basic arithmetic operations add/subtract/multiply divide: There is no explicit sub TM designed here for these.

Arithmetic and logical operations with brackets and indices: This is done by converting equations to postfix form.

Conditional blocks: We have only used 'if then else' type of conditional blocks here.

Loops: In this model loop is started by checking an initial value of a variable and termination is also done by checking that value. Inside the loop body, the variable is updated.

Function calling

- Functions in this model are only 'void' functions that mean those do not return value.
- Return value is only got from function, by parameters sent by reference.
- Function parameters only contain references to other variables with which functions send and receive values.

DESCRIPTION OF OUR TM

Our Turing Machine consists of components as stated before. This portion describes one component in detail. There are three tapes in this TM.

Program tape: This tape contains variable declarations, program and input. When the input instruction is called, the input value is stored in places reserved for input in this tape. This tape has two tracks. One for program and variable and another read only track contains sequence number of boxes of the former track. Point to be noted that the tape has one head with which it read both the track and write only on the first track second one is read only. In another way, we may also describe this tape as only one track having the boxes as tuple, which the TM head read. It writes only on the first part of the tuple, second part of the tuple contains the box number and this part is read only.

Postfix tape/Buffer tape: The arithmetic expressions are converted into postfix form before execution. The postfix equation is stored in this tape. At the beginning, this tape is empty. During the functionality, this tape can also be used for other purpose as well such as buffer while sending a buffer over a network. After finishing work, it is empty.

Stack tape: This tape acts as stack for this TM. At the beginning, this tape is empty. The functionality is same like the other tape but it is called stack tape as it is used as stack. Its way of functionality varies in the following way.

- When a value is pushed the tape head moves right and write the value or sometimes it writes and then moves when the current block is already empty.
- When a value is popped, the head erases the current value and moves left or it moves left and then erases the value if the current box is empty.
- At any time to read the top value just read the current box under the head or sometimes when the current box is empty it moves left and read the value.
- Even then, this is a tape so it can be used for other purpose as tape. Sometimes the push pop is merged with other moves.

As default the second and third tape head is kept over a blank box beside a filled box. When there is a value on either of these tapes that to be read constantly then that tapes head is kept on that box containing value.

- Moves: Moves of a single component are of the following form:

$$\delta(q_1,[(a,m),w,x])(q_2,[(b,m),y,z],[D,D,D])$$

where: $a,b,m,w,x,y,z \in \Gamma, q_1, q_2 \in Q, D \in \{L,R,N\}$

- Tape symbols

There are tape symbols for each instruction and each reserve words and each value with which the program is written on the tape. No defined name such as variable name or function name can be repeated. So at same time a name cannot be used as variable name and function. Let alone overriding/overloading etc. Tape symbols can be categorized in the following categories:

- Value
- Reserve words / key words
- Instruction symbols / state symbols: Note that there is a tape symbol for some of the states if not for all.
- Brackets of different types e.g. {, }, (,).
- Operators: + - * / = > < ! :=

WORKING OF TM AND MOVES

From the left to right the global variables and the functions are written on the program tape, followed by # as the marker for starting point of the main program. When the main program is finished there is a '##' as end marker of main program. After that there is no more program written at the starting of computation. However, at run time when a function is called, the called function's code is copied after the '##'. The next function call will copy the called function after that. When a function returns (finish) it is erased from the place where it was copied. At same time the machine

starts reading the symbols at the program track of program tape from left to right and fetches '#'. When it gets it go to s state and after that goes to their respective states and executes the instructions.

$$\delta(S,[(a,m),B,B])(S,[(a,m),B,B],[R,N,N])$$

$$\delta(S,[(\#,m),B,B])(s,[(\#,m),B,B],[R,N,N])$$

Where, $s \in Q$, $a \in \Gamma - \#$, B= Blank symbol,
S=Starting state.
Moves at the end point.

$$\delta(s,[(\#,m),B,B])(s_{\#},[(\#,m),B,B],[R,N,N])$$

$$\delta(s_{\#},[(\#,m),B,B])(f,[(\#,m),B,B],[N,N,N])$$

Variable definition/storage retrieval
Variable is defined as follows:

$$\Delta V_n \Phi V_v \psi$$

Where:

- Δ the left end symbol of definition,
- V_n is a variable name,
- Φsymbol used as separator between value and name,
- V_v is a variable value,
- ψ the left point.

Each occupies one block of tape, as all of these are tokens. Whenever the tape head sense a Δ, it assumes that there is a variable definition at right and same for the other variable symbols for their respective purpose. It goes to Variable definition state and remains in that state until definition finishes.

$$\delta(s,[(\Delta,m),B,B])(v_d,[(\Delta,m),B,B],[R,N,N])$$

$$\delta(v_d,[(a,m),B,B])(v_d,[(a,m),B,B],[R,N,N])$$

$$\delta(v_d,[(\psi,m),B,B])(s,[(\psi,m),B,B],[R,N,N])$$

where $a \in \Gamma - \psi - \Delta$, v_d, $s \in Q$.

If any defined variable is used as global variable inside some function it must be defined before the function first call due to some restriction in the variable retrieving process.

Variable definition at function parameter or variable as reference to another variable:

- $\Delta V_n \Phi \delta V_{rn} \eta \psi$

Variable is read as follows: $\delta V_n \eta$
Where:

- δ the left end symbol of definition,
- V_n is a variable name,
- η the left point.

As the rule of the all programming languages variable must be defined before use. So from a call point the definition must be at the right side. So when TM read a variable, it first

- Push the return point. (The current box number)
- Push W or R for whether to read or write.
- Push the variable name.

When it starts reading variable it goes to the last point of variable first then it push the next box address (as return address) first then push whether to read or write, then push the variable name then it starts moving left going to 'finding variable state'. When it finds it, (Read a Φ followed by the matched name. Point to be noted that when it read Φ it know that there is a variable name at left, so it check the left box and if left box match the top-stack the definition of variable is found.) then it first delete the pushed name starts moving right two boxes and if the value is other then δ, it is the value. Otherwise it is the reference to another variable so it pushes the new name (the variable name at the reference) and again searches in the previous way. When it finds it, it pops the variable name from the third tape. Then it moves the 3rd tape head left. There it checks whether it was to be read or written.

- If it was to be read, it writes the value to the postfix equation.
- The expressions are solved by converting them to postfix form so the value should be written to postfix equation.
- Otherwise if it was to be written then it moves the value from postfix equation to the value part of the definition and erase from the postfix equation.
- After that it pop the stack so the next top is the memory location from where the variable was referenced. The state is also changed to state to go the place from where the variable was called.

Variable is read from state e. (Moves)

$$\delta(e,[(\delta,m),B,B])(v_r,[(\delta,m),B,B],[R,N,N])$$

$$\delta(v_r,[(a,m),B,B])(v_r,[(a,m),B,B],[R,N,N])$$

$$\delta(v_r,[(\eta,m),B,B])(v_{rpr},[(\eta,m),B,B],[R,N,N])$$

$$a \in \Gamma - \eta, e, v_r, v_{rpr} \in Q$$

$\delta(v_{rpr},[(a,m),B,B])(v_{rpr},[(a,m),B,m],[L,N,R])$

$\delta(v_{rpr},[(\eta,m),B,B])(v_{pn},[(\eta,m),B,R],[L,N,R])$

$\delta(v_{pn},[(a,m),B,B])(v_{fv},[(a,m),B,a],[L,N,N])$

$\delta(v_{fv},[(a,m),B,x])(v_{fv},[(a,m),B,x],[L,N,N])$

$a \neq x,\ a \in \Gamma - \phi$

$\delta(v_{fv},[(\phi,m),B,x])(v_{fn},[(\phi,m),B,x],[L,N,N])$

$\delta(v_{fn},[(a,m),B,x])(v_{fv},[(a,m),B,x],[L,N,N])$

$\delta(v_{fn},[(a,m),B,a])(v_{n},[(a,m),B,a],[R,N,N])$

$\delta(v_{n},[(\phi,m),B,x])(v_{n},[(\phi,m),B,B],[R,N,L])$

$\delta(v_{n},[(\delta,m),B,x])(v_{r-n},[(\delta,m),B,B],[R,N,R])$

$\delta(v_{r-n},[(a,m),B,B])(v_{fv},[(a,m),B,a],[L,N,N])$

$\delta(v_{n},[(a,m),B,R])(v_{egr},[(a,m),a,B],[R,R,L])$

$\delta(v_{n},[(a,m),B,W])(v_{wgr},[(a,m),B,B],[N,L,L])$

$\delta(v_{wgr},[(a,m),w,x])(v_{sgr},[(a,m),w,x],[R,R,N])$

$\delta(v_{egr},[(a,m),B,x])(v_{egr},[(a,m),B,x],[R,R,N])$

$\delta(v_{egr},[(a,m),B,m])(e,[(a,m),B,B],[N,N,N])$

$\delta(v_{sgr},[(a,m),B,x])(v_{sgr},[(a,m),B,x],[R,N,N])$

$\delta(v_{sgr},[(a,m),B,m])(s,[(a,m),B,B],[N,N,N])$

Variable updating: In any statement the Lvalue is calculated and updated. In this case no variable is updated at the right hand side of the assignment operator. So when we finish calculating the right hand side of the assignment operator, we fetch the variable by the previous way and assign the value there at the value part. This will be discussed in the statement part. How to write variable is already discussed in the previous part.

Variable length is each variable will occupy one block. In this TM, address space, variable value length, function names and all other reserve word, keyword length all is assumed to occupy one block as those are tokens of the native code of Turing Machine.

Statements: If we check the source code of any language, it is a series of statements just controlled by some flow control statements. Statements are executed in normal state. At the starting of program the TM is at normal state. At the beginning, inside a block (whether a function call or loop or any other class of block it might be) the TM is at normal state (s state) executing statements. Statements are of the following categories:

Assignment statement: (Variable := expression.): in s state if the TM sense a variable it assumes that it is an Assignment statement. So until it sense a ":=", it waits. Then it goes to e state and moves after that are discussed in the variable part.

$\delta(s,[(\delta,m),B,B])(s_t,[(\delta,m),B,B],[R,N,N])$

$\delta(s_t,[(a,m),B,B])(s_t,[(a,m),B,B],[R,N,N])$

$a \in \Gamma - \{:,=,\delta\}$

Move to just read a variable and write to postfix tape/ buffer $\delta(s_t,[(:,m),B,B])(s,[(:,m),B,B],[R,N,N])$

$\delta(s,[(\delta,m),B,B])(e,[(\delta,m),B,;],[R,N,R])$

- Further moves are discussed at expression part.
- Other moves regarding assignment are discussed in evaluation part.

Read variable from user./Write expression to output: These operations are same as send and receive which is discussed later. The difference is at the send/receive node number. It is the port address of the input and output port.

Send/Receive expression to another Turing Machine: Send: To make the send operation easy, in this model the data stream to be sent is first gathered into the buffer (second tape) which is empty before and after this. The variable reading and sending to tape 2 is discussed in variable read part. Then it reads the send symbol and goes to send state and start sending until the buffer (second tape) is empty. After it is empty it goes to s state again. The moves are discussed below.

Send

$\delta(s,[(\Lambda,m),B,B])(e,[(\Lambda,m),B,R],[R,N,R])$

$\delta(e,[(\Xi,m),B,B])(\Xi,[(\Xi,m),B,B],[R,N,N])$

$\delta(\Xi,[(n,m),B,B])(\Xi_n,[(n,m),B,B],[N,L,N])$

$\delta(\Xi_n,[(n,m),x,B])(\Xi_n,[(n,m),B,B],[N,L,N])$.....

$\delta(\Xi_n,[(n,m),B,B])(s,[(n,m),B,B],[R,N,N])$

Receive

$\delta(s,[(\Theta,m),B,B])(\Theta,[(\Theta,m),B,B],[R,N,N])$

$\delta(\Theta,[(n,m),B,B])(\Theta_n,[(n,m),B,B],[N,L,N])$

$\delta(\Theta_n,[(n,m),x,B])(\Theta_n,[(n,m),x,B],[N,R,N])$

$\delta(\Theta_n,[(n,m),B,B])(s,[(n,m),B,B],[R,N,N])$

$\delta(s_t,[(;,m),B,B])(s,[(;,m),B,B],[R,N,N])$

$\delta(s,[(a,m),B,B])(s_{-1},[(a,m),B,m],[L,N,R])$

$\delta(s_{-1},[(a,m),B,B])(s_{-2},[(a,m),B,W],[L,N,R])$

$\delta(s_{-2},[(\eta,m),B,B])(V_{pn},[(\eta,m),B,W],[L,N,N])$

Both together (only the communicating move)

$$\delta\left(\Theta_{n2},\left[(n2,m),x,B\right]\right)\left(\Theta_{n2},\left[(n2,m),x,B\right],\left[N,R,N\right]\right)_1$$

$$\delta\left(\Xi_{n1},\left[(n1,m),x,B\right]\right)\left(\Xi_{N1},\left[(n1,m),B,B\right],\left[N,L,N\right]\right)_2$$

We have stated before the components those are exchanging data among themselves can be ports of a single CPU or different CPUs connected by communication ports. But all of those should have an address.

Expressions: These are arithmetic operations with brackets and indices or condition checking. When an expression comes as a variable assignment statement a ';' is pushed (At the end it will be popped) then the expression is converted into a postfix form. Our expression will contain numeric values and operators. The numeric values will be a value in a box of tape. The operators will be:

- (: open parenthesis
-): closing parenthesis
- *: multiplication
- /: division
- +: addition
- -: subtraction

These operators have four levels of precedence.

- Highest: (,)
- Middle: *, -
- Low: +, -
- Lowest: < > =!

We will add one more operator to help with the processing of the program. That is a semicolon, ';', which will indicate the end of an expression.

To convert an expression from infix to postfix, we must first determine if the next (valid) item in the equation is a number or an operator.

If the item is a number, it is sent directly to the postfix equation. This is discussed in the variable part how to do it.

If the item is a operator, we use the stack to do one of the following:

- If the operator is: (, *, /, + or -
  Pop the stack until either the stack is empty, the operator of (at the top of the stack or the top value on the stack has lower precedence than the current operator. As each operator is popped from the stack, it is sent directly to the postfix equation. Then we push the current operator onto the stack.

$$\delta\left(e,\left[(o,m),B,B\right]\right)\left(e_o,\left[(o,m),B,B\right],\left[N,N,L\right]\right)$$

$$\delta\left(e_o,\left[(o,m),B,u\right]\right)\left(e_o,\left[(o,m),u,B\right],\left[N,R,L\right]\right)$$

$o \in \{+,-,*,/,<,>,=,!,(\}$ $u \in \{+,-,*,/,<,>,=,!\}$ u has equal or higher precedence to o.

$$\delta\left(e_o,\left[(o,m),B,d\right]\right)\left(e_{oo},\left[(o,m),B,d\right],\left[N,N,R\right]\right)$$

$$\delta\left(e_{oo},\left[(o,m),B,B\right]\right)\left(e,\left[(o,m),B,o\right],\left[R,N,R\right]\right)$$

$d \in \{+,-,*,/,<,>,=,!,(\}$ 'd' has lower precedence then o.

- If the operator is:)
  Pop the stack until pop an open parenthesis, (. The closing parenthesis is NOT pushed on the stack. All operators that are popped (except for the open parenthesis) are sent directly to the postfix equation. If there is not an open parenthesis on the stack (i.e. you empty the stack), the equation has an error of an unmatched closing parenthesis.

$$\delta\left(e,\left[(),m\right),B,B\right]\right)\left(e_),\left[(),m\right),B,B\right],\left[N,N,L\right]\right)$$

$$\delta\left(e_),\left[(),m\right),B,u\right]\right)\left(e_),\left[(),m\right),u,B\right],\left[N,R,L\right]\right)$$

$$\delta\left(e_),\left[(),m\right),B,(\right]\right)\left(e,\left[(),m\right),B,B\right],\left[R,N,N\right]\right)$$

- If the operator is ';' pop the stack until ';' at the top, sending all popped operators to the postfix equation other then ';'. Evaluate the expression and save the result into variable. Then pop the ';'. In case the expression contains variables the variables are retrieved and the value is sent to postfix equation.

$$\delta\left(e,\left[(;,m),B,B\right]\right)\left(e_{v-p},\left[(;,m),B,B\right],\left[N,N,L\right]\right)$$

$$\delta\left(e_{v-p},\left[(;,m),B,x\right]\right)\left(e_{v-p},\left[(;,m),x,B\right],\left[N,R,L\right]\right)$$

$x \in \{+,-,*,/,<,>,=,!\}$

$$\delta\left(e_{v-p}\left[(;,m),B,;\right]\right)\left(e_{v-r},\left[(;,m),B,B\right],\left[N,L,N\right]\right)$$

$$\delta\left(e_{v-r},\left[(;,m),u,B\right]\right)\left(e_{v-r},\left[(;,m),u,B\right],\left[N,L,N\right]\right)$$

$$\delta\left(e_{v-r},\left[(;,m),B,B\right]\right)\left(e_v,\left[(;,m),B,B\right],\left[N,R,N\right]\right)$$

Evaluating postfix equation: To evaluate a postfix expression, we will again need to use the stack. However, the stack will contain numeric values instead of operators.

- When a numeric value is encounter, the value is pushed on the stack.
- When an operator (other than ;) is encountered, two values are popped from the stack the operation is

performed on these two values and the result is pushed onto the stack.

- When the no operand is encountered, there should only be one value on the stack above the ';'. This value is the result of the expression.

$\delta(e_v,[(;,m),o,B])(e_{vo},[(;,m),B,B],[N,N,L])$,

$o \in \{+,-,*,/,<,>,=,!\}$

$\delta(e_{vo},[(;,m),B,n])(e_{vo-e},[(;,m),n,B],[N,N,L])$

n = numeric value

$\delta(e_{vo-e},[(;,m),n,n_1])(e_v,[(;,m),B,non_1],[N,N,R])$

$\delta(e_v,[(;,m),B,B])(e_{v-1},[(;,m),B,B],[R,N,L])$

$\delta(e_{v-1},[(a,m),B,n])(e_{v-2},[(a,m),n,m],[L,R,R])$

$\delta(e_{v-2},[(a,m),B,B])(e_{v-3},[(a,m),B,W],[L,N,R])$

$\delta(e_{v-3},[(a,m),B,B])(e_{v-3},[(a,m),B,B],[L,N,N])$

$\delta(e_{v-3},[(\eta,m),B,B])(v_{pn},[(\eta,m),B,B],[L,N,N])$

The next moves from here is discussed in the variable part. For every expression we first push ';' as we told before, then convert the expression into postfix then evaluate, then pop ';'.

Conditional blocks: Conditional blocks are of two types

Block like if else statements in C: when it reads an α it goes to expression state. In expression if it reads a β then it evaluates and the boolion result is in the third tape (stack) it also goes to the condition state. If the condition is false it goes to false state and do nothing until it goes to a normal state.('s') in false state it only moves right but when reads an α it just push an α and when it finds ε it just pops the α. If it finds ε in false state and the stack-top is false then it pops the false and goes to normal state. If it reads χ then it goes to else part so if before it was false (there was false at the top) then it goes to true state else it goes to false state. The moves are as follows:

$\delta(s,[(\alpha,m),B,B])(e,[(\alpha,m),B,B],[R,N,N])$

$\delta(e_v,[(\beta,m),B,B])(c,[(\beta,m),B,B],[R,N,L])$

$\delta(c,[(a,m),B,F])(s_f,[(a,m),B,F],[R,N,N])$

$\delta(s_f,[(a,m),B,w])(s_f,[(a,m),B,w],[R,N,N])$

$w \in \{F,\alpha\}$

$\delta(s_f,[(\alpha,m),B,F])(s_{f\alpha},[(\alpha,m),B,F],[R,N,R])$

$\delta(s_{f\alpha},[(a,m),B,B])(s_f,[(a,m),B,\alpha],[R,N,N])$

$\delta(s_f,[(\varepsilon,m),B,\alpha])(s_f,[(\varepsilon,m),B,B],[R,N,L])$

$\delta(s_f,[(\varepsilon,m),B,F])(s,[(\varepsilon,m),B,B],[R,N,N])$

$\delta(c,[(a,m),B,T])(s,[(a,m),B,B],[R,N,N])$

$\delta(s,[(\chi,m),B,B])(s_f,[(\chi,m),B,F],[R,N,R])$

$\delta(s_f,[(\chi,m),B,F])(s,[(\chi,m),B,B],[R,N,N])$

$\delta(s,[(\varepsilon,m),B,B])(s,[(\varepsilon,m),B,B],[R,N,N])$

Block like switch statements in C: For some simplicity reason of this design this part is not covered.

Loops: When there is an instruction symbol for loop. It push the program tape number as return point. Then it checks the condition. Depending on the result of the check it goes to either true or false state and remains there till it gets end loop symbol. At end if it gets the end symbol while false state it pops the top (return point) else if it was in true state while reading the end loop symbol it goes back to the old point to check the condition again. The moves are as follows:

$\delta(s,[(\Pi,m),B,B])(e,[(\Pi,m),B,m],[R,N,R])$

$\delta(e_v,[(\Pi,m),B,B])(c_1,[(\Pi,m),B,B],[R,N,L])$

$\delta(c_1,[(a,m),B,F])(s_{lf},[(a,m),B,B],[R,N,L])$

$\delta(s_{lf},[(a,m),B,x])(s_{lf},[(a,m),B,x],[R,N,N])$

$\delta(s_{lf},[(\Omega,m),B,x])(s,[(\Omega,m),B,B],[R,N,N])$

$\delta(c_1,[(a,m),B,T])(s,[(a,m),B,B],[R,N,N])$,

$\delta(s,[(\Omega,m),B,B])(s_r,[(\Omega,m),B,B],[L,N,L])$,

$\delta(s_r,[(a,m),B,x])(s_r,[(a,m),B,x],[L,N,N])$,

$\delta(s_r,[(a,m),B,m])(s,[(a,m),B,B],[N,N,N])$

Function definition/calling: Function is defined before the main program starts. Unlike the variable definition, there is no move for function definition area. It is covered at S state before reading # and going to s state. Function is defined as follows:

$\gamma F_n \Phi(\Delta V_n \Phi \delta V_m \eta\psi, \Delta V_n \Phi \delta V_{rn} \eta\psi,..)\ \theta$ {Body block again coded as the main program is coded. }Ω

The parameter variables are all by reference wheather going in or out. That is there format is "$\Delta V_n \Phi \delta V_m \eta\psi$" Where:

- γ the left end symbol of definition,
- F_n is a function name,
- Φ symbol used as separator between function name and parameters,

- $\Delta V_n \Phi \delta V_{rn} \Re \psi$ is a variable referenced by value details of which is already discussed at variable part,
- θ the left point of definition before the body.
- Then comes the body of function as sequences of code.
- Ω the last point of body.
- Each occupies one block of tape, as all of these are tokens.

Function is used as follows: (The moves are discussed below the text.)

$$\lambda F_n(\delta V_n \eta, \delta V_n \eta \ldots.)\sigma$$

Where:

- λ the left end symbol of definition,
- F_n is a function name,
- $\delta V_n \eta$ is the variable call which is already discussed at variable part.
- σ the left point of function call.

When TM read a Function call it goes to function call state and then it reach to the end of the call point 'σ', then

push the current program point (block number of program tape), the box address as return point then it start moving left and push the parameters from the left side, at last push the Function name then it starts moving left going to state "finding function". When it finds it, (Read a Φ followed by the matched name. Point to be noted that when it read Φ it comes to know that there is a Function name at left, so it check the left box and if left box match the top-stack the definition is found.) it start moving right two boxes and then parameters are starting there. It first then push '#' then goes to the last part of definition and start coming back and pushing the whole function body until the starting of function. Then it goes to the end point of the tape, starts popping the whole function. When it pop # it assumes the next is parameters so it places parameters in appropriate places. Then it goes to the s state and start executing body. At the end (when it read Ω) it again goes to the return point from where it was called. We have told in the variable part that global variables using inside a function body must be defined before the function definition so there is no problem in global variable retrieval process.

Moves:

$$\delta(s,[(\lambda,m),B,B],(f,[(\lambda,m),B,B],[R,N,N])$$

$$\delta(f,[(a,m),B,B])(f,[(a,m),B,B],[R,N,N])$$

$$\delta(f,[(\sigma,m),B,B])(f_{pr},[(\sigma,m),B,B],[R,N,N])$$

$$\delta(f_{pr},[(a,m),B,B])(f_{pr},[(a,m),B,m],[L,N,R])$$

$$\delta(f_{pr},[(\sigma,m),B,B])(f_{pr},[(\sigma,m),B,B],[L,N,N])$$

$$\delta(f_{pr},[(),m),B,B])(f_{pr},[(),m),B,B],[L,N,N])$$

$$\delta(f_{pr},[(\delta,m),B,B])(f_{pr},[(\delta,m),B,\delta],[L,N,R])$$

$$\delta(f_{pr},[(\eta,m),B,B])(f_{pr},[(\eta,m),B,\eta],[L,N,R])$$

$$\delta(f_{pr},[(,,m),B,B])(f_{pr},[(,,m),B,B],[L,N,N])$$

$$\delta(f_{pr},[(\lambda,m),B,B])(f_{ff},[(\lambda,m),B,B],[L,N,L])$$

$$\delta(f_{ff},[(a,m),B,x])(f_{ff},[(a,m),B,x],[L,N,N])$$

$$\delta(f_{ff},[(\phi,m),B,x])(f_{fn},[(\phi,m),B,x],[L,N,N])$$

$$\delta(f_{fn},[(a,m),B,x])(f_{ff},[(a,m),B,x],[L,N,N])$$

$$\delta(f_{fn},[(a,m),B,a])(f_{n},[(a,m),B,\#],[R,N,R])$$

$$\delta(f_{n},[(a,m),B,B])(f_{n},[(a,m),B,B],[R,N,N])$$

$$\delta(f_{n},[(\Omega,m),B,B])(f_{p},[(\Omega,m),B,\Omega],[L,N,R])$$

$$\delta(f_{p},[(u,m),B,B])(f_{p},[(u,m),B,u],[L,N,R])$$

$$\delta(f_{p},[(\gamma,m),B,B])(f_{ge},[(\gamma,m),B,\gamma],[R,N,R])$$

$$\delta(f_{ge},[(x,m),B,B])(f_{ge},[(x,m),B,B],[R,N,N])$$

$$\delta(f_{ge},[(B,m),B,B])(f_{c},[(B,m),B,B],[N,N,L])$$

$$\delta(f_{c},[(B,m),B,u])(f_{c},[(u,m),B,B],[R,N,L])$$

$$\delta(f_{c},[(B,m),B,\#])(f_{gsf},[(B,m),B,\#],[L,N,N])$$

$$\delta(f_{gsf},[(u,m),B,\#])(f_{gsf},[(u,m),B,\#],[L,N,N])$$

$$\delta(f_{gsf},[(\gamma,m),B,\#])(f_{sp},[(\gamma,m),B,B],[R,N,L])$$

$$\delta(f_{sp},[(a,m),B,B])(f_{sp},[(a,m),B,B],[R,N,N])$$

$$\delta(f_{sp},[(\delta,m),B,B])(f_{sp-w},[(\delta,m),B,B],[R,N,L])$$

$$\delta(f_{sp-w},[(a,m),B,x])(f_{sp-w},[(a,m),B,B],[N,N,L])$$

$$\delta(f_{sp-w},[(a,m),B,\delta])(f_{sp-c},[(a,m),B,B],[N,N,L])$$

$$\delta(f_{sp-c},[(a,m),B,x])(f_{sp-l},[(x,m),B,B],[R,N,L])$$

$$\delta(f_{sp-l},[(x,m),B,\eta])(f_{sp},[(x,m),B,B],[N,N,N])$$

$$\delta(f_{sp},[(\theta,m),B,B])(s,[(\theta,m),B,B],[R,N,N])$$

$$\delta(s,[(\Omega,m),B,B])(s_r,[(\Omega,m),B,B],[L,N,L])$$

$$\delta(s_r,[(a,m),B,x])(s_r,[(a,m),B,x],[L,N,N])$$

$$\delta(s_r,[(a,m),B,m])(s,[(a,m),B,B],[N,N,N])$$

CONCLUSIONS

We have described one component of our whole Turing machine such components communicate among themselves by the moves described in the send receive

section. This way our whole Turing Machine works as a model of distributed computing.

As future work, we may include more complex tasks such as object-oriented structure.

REFERENCES

1. Turing, A.M., 1936, On computable numbers with an application to the entscheidungsproblem (Descision Problem). Proc. London Math. Soc., 42: 230-265.
2. Hopcroft, J.E., R. Motwani and J.D. Ullman, 2000. Introduction to Automata Theory Language and Computation. 2/E, Addision-Wesleg CO, USA.
3. Guetta, D., 2003. Turing Machine Development Environment. 19 Church Mount, London, N2 0RW, England.
4. Khiyal, M.S.H., 2004. Theory of Autometa and Computation. National Book Foundation, Islamabad.
5. Stallings, W., 1999. Data and Computer Communications. 6/E, Prentice Hall, Inc, USA.
6. Silberschatz, A., P.B. Galvin and G. Gagne, 2004. Operating System Concepts. 7/E, John Wiley and Sons.