INPUT/OUTPUT AUTOMATON AS A WEB SEARCH ENGINE

Dr. M. Thiyagarajan*, Chaitanya Raveendra and Dr.V.Thiagarasu*****

*Professor Emeritus and Dean Research, Nehru Group of Institution, Kuniamuthur, Coimbatore, Tamil Nadu, India, Email:m_thiyagarajan@yahoo.com
**Research Scholar, PhD (Karpagam University), NIITM, Nehru Colleges, Kuniamuthur, Coimbatore Tamil Nadu, India, Email: chaitanya2575@gmail.com
***Associate Professor (CS), Gobi Arts &Science College, Gobichettipalayam – 638 453, Email: profdravt@gmail.com

ABSTRACT: Web service execution can be undertaken by hardware consisting of many interconnected processes. The hardware design is a separate study based on the pieces of distributed algorithm which concurrently and independently and deals with limited amount of information. This system which has worked even if individual processes and communication channels operates at different speeds and some of the components fails. Earlier, the web services search by an automaton was restricted to software design under study by Thiyagarajan and Chaitanya. Now the process and hardware features will be addressed here. This leads to the design of I/O automation for hardware distribution algorithm. Sample illustration are studied and implemented.

KEYWORDS: I/O Automata, Channel I/O, Process I/O, Input Output Algorithms, I/O Models.

INTRODUCTION

Finite state automaton is better suited as conceptual model for web search activities. For all computations related to basic activities of special e-service, we need to understand the user categorization their authentication to the entire services to engage them to avail service rights.

I/O automaton model has been clearly defined with candy machines by Nancy Lynch and Mark.R.Tuttle. The Input Output automaton provides an appropriate model for discrete event systems consisting of concurrently operating components. Each system component is modeled as an I/O automaton, which is essentially (possibly infinite state) automaton with an action labeling each transition. An I/O automaton exhibits behavior when the environment observes certain restrictions on the production of inputs. I/O automaton may be non-deterministic, and indeed the non-determinism is an important part of the model's descriptive power. I/O automata can be composed to yield other I/O automata.

Nancy and Jennifer develop an automaton representing the resource allocation algorithm used to allocate the resources. A resource allocation algorithm decides which user gets which resources at which time; thus, it supplies the code for the trying and exit regions. A distributed resource allocation algorithm consists of one component for each user; the component communicates with each other by message passing. They develop an I/O model of the resource allocation module which is useful in stating the properties that concern the infinite behavior of the system, such as no-deadlock and no-lockout, and which supports modular algorithm design and verification.

Finite state machine model following in the earlier concepts of e-services by Daniela Berardi, Fabio De Rosa, Luca De Santis And Massimomecella, each interaction of an e-service is described by the triple < input command, internal computation, output message >; however, by adopting a black box approach, we skip over internal computations and represent only the input/output behavior of e-Service from the client view point, i.e., that have some effects towards the client: each interaction is therefore described by the pair < input command, output message >only.

Documents have been automated by the push down automata in the works of Francois. As e-services are context free their distribution can be context free grammar and by Basic definition of PDA is, Pushdown automata is a finite automata with auxiliary storage devices called stacks. (A stack is merely a pile. And symbols are normally placed on stacks rather than various colored discs.) The rules involving stacks and their contents are: a) Symbols must always be

placed upon the top of the stack. b) Only the top symbol of a stack can be read. c) No symbol other than the top one can be removed.

In the parallel run , we have the timed automata are developed to explain machine acceptance falling into consideration the time limits by Johan Bengtsson and Wang Yi, A timed automaton is essentially a finite automaton (that is a graph containing a finite set of nodes or locations and a finite set of labeled edges) extended with real-valued variables. Such an automaton may be considered as an abstract model of a timed system. The variables model the logical clocks in the systems that are initialized with zero when the system is started, and then increase synchronously with the same rate. Clock constraints i.e. guards on edges are used to restrict the behavior of the automaton.

All this approaches have not addressed the problem of distributed environment of e-services. This covers both synchronous and asynchronous systems as a whole. Classical distributed algorithm on these systems are developed by various point of time to solve the multi various environments. We worked on the region of xml documents handling in multi course environment under various possibilities and automaton of the sample case. We have translated the problem of I/O handling to the problem of document handler. The result will be a combination of the process and Channel automaton. Here we developed I/O automation on the lines of Nancy A lynch, a conceptual model for creating automated web service engine.

THEORETICAL UNDERPINNING

This section describes the definitions and notations used in this paper. This paper concentrates on the hardware implementation of the data access controls for every documents accessed by the user. The following subsections gives the description on asynchronous and synchronous systems, shared variables, shared memory, Input Output Automaton, Channel I/O and Process I/O.

Asynchronous And Synchronous System

Asynchronous interaction between the two parties need not agree on reaching specific parts of their respective computations. The caller computation does not wait. In asynchronous systems when a call is made by the user, it will eventually proceed with other computations. The process will inform the user about the result later. Thus all can work separately using different threads.

Synchronous interactions are conceptually simple, because they correspond to a single thread of execution. A conventional single-threaded computation can be naturally partitioned into components where one calls another. Synchronous Execution finishes the work within a single thread. When a call is made in synchronous mode, then it will wait till the output is ready. When it receives the output, it will continue execution with the result it received. Our work is the combination of synchronous and asynchronous execution. Data is partitioned and stored in rdf format which explains the asynchronous execution and the mode of message traversal is an synchronous interaction. Thus based on the asynchronous retrieval of the search information, synchronous combination of the information is send as the response.

Shared Variable

A *variable type* consist of

- a set V of values
- an initial value $v_0 \varepsilon V$
- a set of invocations
- a set of responses
- a function $f : invocations \times V \rightarrow responses \times V$

The function f says what happens when a given invocation arrives at the variable and the variable has a given value; f describes the new value the variable takes on and the response is returned. Note that a variable type is not an I/O automaton, even though some of its components look similar to I/O automaton components. In normal scenario, actions are concurrent. But in the case of an Input Output Automata, input and output is separate action and it won't occur at once. After receiving the input, it will read the input and performs the action described in the function f and then the response will be generated as an output and the I/O automata is modified for XML document transfer shared for any number of users.

Shared Memory

We have a shared memory system A. Shared variable x in system A means the set values x must be equal to the set V of values of the type, and that the set of initial values for x consists of just one element, v_0. Moreover, all the transitions involving x must be describable in terms of the invocations and responses allowed by the type. Namely, each action

involving x must be associated with some invocation A of the variable type. The shared variable x resides in the shared memory A. So each actions performed on the variable x must be considered separate. So for each invocation, this shared variable is written as the response before another invocation.

Channel Automaton

The channel associated with each directed edge of G is modeled as an I/O automaton C_{ij}. Its external interface consists of inputs of the form send$(m)_{i,j}$ and outputs of the form receive$(m)_{j,i}$. The Channel Automaton refers to the automaton of the message channel. Send and receive can be considered as the input and output of the Input Output Channel. Channel Automata is a part of Input Output Automata. The output after the execution of a process is the input to the channel. All the activities are different and are performed at a time and in an unpredictable order. One example is the FIFO Queue. The channel I/O carries the request or output of an XML data either from data base or another server or any process

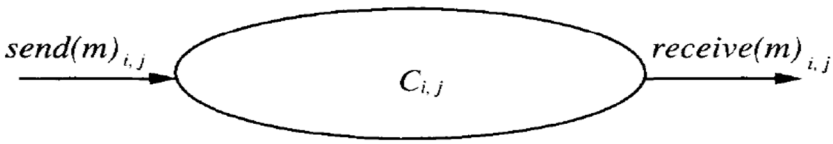

Fig.1: Channel I/O

Process Automaton

The process associated with each node 'i' is modeled as an I/O automaton, P_i. P_i usually has some input and output actions; this allows us to express problems to be solved by an asynchronous network. In addition P_i has outputs of the form send$(m)_{i,j,}$ where j is an outgoing neighbor of i and m is the message of the form receive$(m)_{j,i,}$ where j is an incoming neighbor. In Input Output automata, the process is the I/O hardware. The input value is taken from the output of the channel. The request for an XML is received from the I/O channel and processed by the processor.

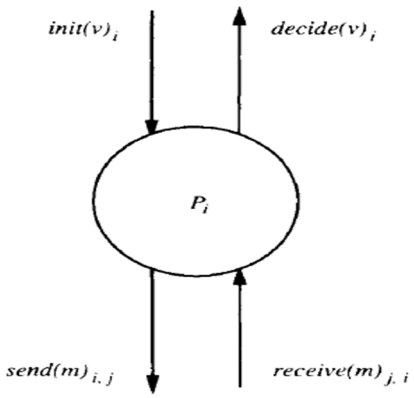

Fig.2: Process I/O

INPUT/OUTPUT MODEL

An I/O automaton models a distributed system component that can interact with other system components. It is a simple type of state machine in which the transitions are associated with named actions. The actions are classified as either input, output, or internal. The inputs and outputs are used for communication with the automaton's environment, while the internal actions are visible only to the automaton itself. The input actions are assumed not to be under the automaton's control, they just arrive from the outside while the automaton itself specifies what output and internal actions should be performed. We aim at developing an Input Output model for a distributed system which will have the document available and execution based on the mode of request. Here we develop an I/O model for a component that access the data based on his rights. For this data is classified or partitioned and the combination of each result produces the output required for the user. A model encompassing synchronous, asynchronous features, probabilistic and non-deterministic data by means of hardware design. Thus individual components are automated and each component work synchronously, the combination of all the synchronous output is collected for the final asynchronous system. We can

increase the component to any level based on the requirements of the user. Here, in this paper we develop an algorithm that read and writes an xml document using an I/O automaton and implemented using java.

Input/Output Automata

An *I/O automaton A* consists of five components:

- sig(A), a signature
- states(A), a set of states
- start (A), a nonempty subset of states (A) known as the start state or initial state.
- trans(A), a state-transition relation, where trans(A) $\subseteq$ states(A) $\times$ acts(sig(A)) $\times$ states(A); this must have the property that for every state s and every input action π, there is a transition (s, π, s') ε trans(A)
- tasks(A), a task partition, is an equivalence relation on local(sig(A)) having at most countable equivalence classes

The signature represents the input, output and the internal actions. Input and output are specified but the internal actions are specific to the automaton. States represents the processes or I/O hardware. Start state specifies the start state. Task partition includes the set of tasks to be processed by the automata. The combination of Channel I/O and process I/O is the Input Output Automata. The best example of a Channel I/O is the FIFO Channel. When it receives an input using send $(m)_{i,j}$, it adds the value or document to the queue and, the output action is performed using receive$(m)_{i,j}$ to process I/O. Example for the process I/O is the asynchronous system.

Operations of Automaton

This section describes the operations that can be performed for I/O automata. Composition and operation hiding are the two operations that can be performed for I/O automata.

Composition

Composition refers to the construction of complex systems by composing automata with individual system components. If the component did any actions on a single component, corresponding changes must be done in all the individual systems. Restrictions have to be checked before the composition. It allows infinite compositions since I/O automata are used to model logical systems as well as physical systems.

Formally, we define a countable collection $\{S_i\}$, $i \varepsilon I$ of signatures to be *compatible* if for all i, j ε I, i $\neq$ j, all of the following hold:

1. int(S_i) $\cap$ act (S_j) = 0
2. out(S) $\cap$out (s j) = 0
3. No action is contained in infinitely many sets acts(S_i)

We say that a collection of automata is compatible if their signatures are compatible. Consider a fixed index set I = { 1 , . . . , n} and let A be the composition of all the process automata P_i, $i \varepsilon I$, the channel automata $C_{i,j}$,$i,j \varepsilon I$. The following figure depicts the "architecture" for the special case where n = 3. The resulting composition is a single automaton representing a distributed system.

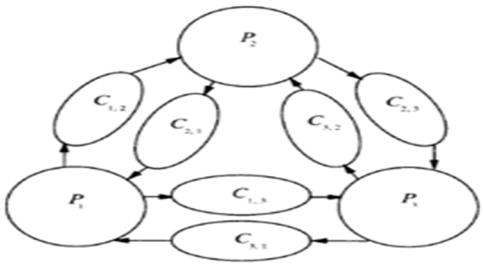

Fig.3 : Composition of services

The state of the system consists of a state for each process plus a state for each channel. Each transition of the system involves one of the following:

1. An init(v)$_I$ input action, which deposits a value in P_i's val(i) variable, val(i)$_i$
2. A send(v)$_{i,,j}$ output action, by which P_i's value val(i)$_I$ gets put into channel $C_{i,j}$
3. A receive(v)$_{i,,j}$ output action, by which the first message in $C_{i,j}$is removed and simultaneously placed into P_j's variable val(i)$_j$
4. A decide(v)$_I$ output action, by which P_i announces its current computed value.

Hiding

Hiding operation for signatures can be defined as " if S is a signature and $\sum\subseteq$ out(S), then hide$_{\sum}$,(S) is defined to be the new signature S', where in(S') -in(S), out(S') - out(S) - $\sum$, and int(S') - int(S) U $\sum$.. The hiding operation for I/O automata is now easy to define" if A is an automaton and (I) C $\subseteq$out(A), then hide$_{\phi}$(A) is the automaton A' obtained from A by replacing sig(A)with sig(A ') = hide$_{\phi}$(sig(A)). Hiding hides the output of a process of an I/O automaton. They will do it by considering it as internal actions and internal actions are ignorant to the I/O automata. The hiding output cannot be used for further computations.

Input/Output Algorithms

Here, we have Channel Automata and Process Automata to construct the I/O automata. I/O Automata is the combined effort of channel through which the data flow and the software process which describes the actions to be performed. The process I/O and Channel I/O works together as to the concept of asynchronous system. Each action is executed in the process and sends the response to the channel, which is the input to the channel and output from the process. The data for the actions is received from the channel which is the input to the process and output from the channel. The following two subsections describe the Channel I/O and process I/O. Each process and channel are the states of the automaton. This section describes the components of I/O. The input and output variables has been described using init() and decide() and the actions is specified using precondition and effects.

Channel I/O

The Channel I/O is a FIFO data structure which holds the data either for processing or after the execution. The pseudo code for the channel I/O in java has been provided in Appendix A. The pseudo code provided is a working example of a FIFO Queue. If an input is received in the channel it is added to the FIFO List. Considering the example of a document, then each document is added to the queue. When the document is ready to execute and if it is the first element of the queue, it will be removed from the queue. The Actions are described as follows:

a) Receive the input from send(m)$_{i,j}$ and then add to the Queue.
b) If this is the first element of the queue, then it is removed from the queue and send for further processing.

This has been specified using precondition and effect. Precondition describes the conditions to be true and effect is described as a subroutine. The Action signature in case of a document is as follows.

Input Actions : SEND($m_{i,j}$)
Output Actions : RECEIVE($m_{i,j}$)
Internal Actions : NONE

The transition relation is as follows :

SEND($m_{i,j}$)	Effect	:	Add document to the queue.
RECEIVE($m_{i,j}$)	Precondition	:	If $m_{i,j}$ is the first element
	Effect	:	Remove the element from Queue.

In our sample illustration we have implemented the process in java with an integer value as the input. Thus it receives the integer as the input to the channel and added to the FIFO. When it's about to receive by the process, it will removed from the queue, if it's the first element of the queue. This has been modified to add an XML document to the queue.

Process I/O

The process I/O explains the working of a process along with input and output. It consists of input and output for the data variable and the input and output for the channel. It defines the lines of execution to perform a particular task. The data from the channel is fetched and then perform the execution using the transitions described in the process. We have considered an example of accepting an integer as input and assigning to the list. The code has been implemented in java and the same has been inserted in Appendix B. It describes the working of Process I/O. Here, in the example it accepts the value and updated the vector value and send to the channel. The following actions explains the flow,

a) The value is accepted from the input.
b) The vector is updated with the value.

c) It's written back to the channel.

The Action signature is as follows.

Input Actions : INIT$(v)_i$, RECEIVE$(v_{j,i})$
Output Actions : DECIDE$(v)_i$, SEND$(v_{i,j})$
Internal Actions : NONE

The transition relation for reading an xml document is as follows:

INIT$(v)_i$	Effect	:	file Val(i)=xmlfile1
RECEIVE $(v_{j,i})$	Effect	:	file Val(j)=xmlfile2
SEND$(v_{i,j})$	Precondition	:	If val(i)== xmlfile2
	Effect	:	Satiated=yes
DECIDE$(v)_I$	Precondition	:	for(j=1;j<=n, j++,Val(j)!= NULL) V=readxml(val(1), val(2),…, val(n)) Xmlfile2=V
	Effect	:	Satiated=yes

In the signature described above, we have four set of I/O. INIT$(v)_i$ carries the input xml document. When a file is received, it is stored in a temporary register, V. It receives the value from the channel using the RECEIVE $(v_{j,i})$ function. If the file is read, it is assigned to the xmlfile2. When DECIDE$(v)_i$ stores the output of the process, the xmlfile2 is assigned and after the execution of the statements satiated= 'yes' determines the end of the execution. SEND$(v_{i,j})$ carries the input the value to the channel. In our sample program, we received an integer as the input and if it belongs to i, then its written to I else it is written to node j. The java implementation for the process I/O has been developed and inserted in Appendix B.

ILLUSTRATION

The work concentrates on the hardware implementation of distributed algorithm. Distributed computing explains the distribution of tasks among multiple processes. Here the objective is to develop an automatic web search engine, in which the search parameters are distributed to different processors and executing the task concurrently. All the resources are distributed among the processes and they work simultaneously synchronously and produce the result. The illustration describes the distributed execution of XML documents among many processors. The data and the services are distributed. All these services work independently, connected by means of channels through hardware implementation.

Consider an xml file which contains the details as employee ID, Designation, Gender and Image. Thus each parameter of the employee file are partitioned into separate component. We have developed a toy example which has considered the case of a single component that handles the image. All the read, write and modify properties on that image component are developed. In the sample, we have performed a read operation on the image. The following steps describe the sequence and the data flow of reading an image. An action is the combination of process I/O and Channel I/O. As it is an I/O automaton, we have to specify all the action for the final result. Thus the above request can be explained in few steps.

1) Request came as an invocation.
2) Put it in the channel for process
3) Collected from the channel and received by the process
4) Process identified the request as read photo and sends it to channel for database access.
5) Accept as input to the database channel.
6) Data is collected from the channel by another process
7) Data is retrieved and output from the process is fed back to the parent process to the corresponding channel
8) Parent process receives the output and then generates the response and fed to the response queue for reply to the user.

This is the sequence of actions that is to be executed for the photo retrieval and this paper deals with automaton model for a single action. Based on this illustration, five components of the input output automaton model has been explained along with the diagram.

Signature

Signature is a triple consisting of input, output and internal actions. Input refers to the request from the user like view image, modify an image or changing the designation. In this case, input is the view image request of people belonging to age group of 20 and 26 from the author and after the group of executions we will receive the image of all the users

belonging to a particular age group, this is the output after the executions. An internal action corresponds to the sequence of steps in the execution for the generation of output. Sequence of steps and the conditions and status generation is explained in the transitions.

Transitions

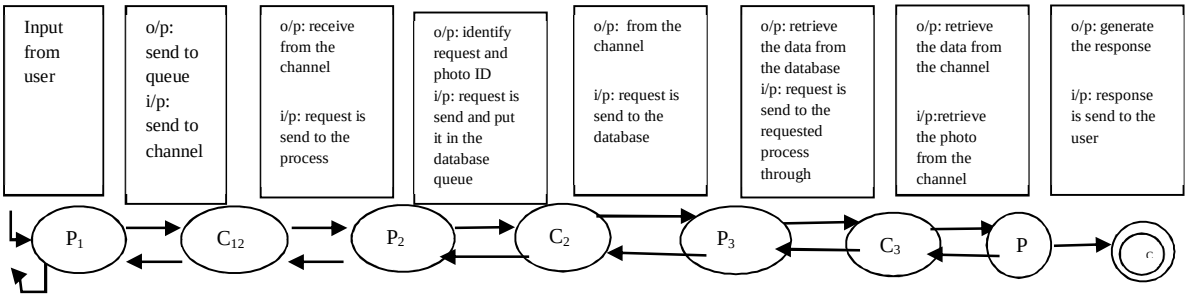

Fig. 4: Transition Diagram for a sample illustration of READ

The above transition diagram describes the internal actions to retrieve the photo from the database and send it to user for display. When a request came for the read, it's in the initial state, S_0. When a request came, it is fed to the channel and will wait in C_{12}. When the channel sends the message for processing, then it moves to P_2. This process identifies the request and sends a message to database in the FIFO Queue, state C_{23}. When it reaches the data base process will receive the request and collect the photo in process P_4. The output message along with the result is put into the output channel for our requested process, C_{34}. The output from the channel is taken and then received by state P_4 and the response to the user is generated from the received input and send as a response to the user.

Start State

The initial request triggers the automaton. The initial state is the request from the user. Request to view the image is the input to the state.

States

Each state represents the actions performed after the triggering of the event. States are FIFO Queue and xml documents.

Tasks

Task represents the classes of elements or it specifies the conditions till that process have to execute. Tasks represent the class description and the program will execute based on the local class for the specific request. In this example the local class is the group belonging to 20 and 26. We have classified the age group into A, B, C according to the age limits.

Thus, the above sample is intended for single mode of request and the data have many properties and we need to develop the cases for all the properties of the employee. The following two model describes the concurrent execution for read–write and read-write-modify for each set of requests. The task specifies the classes for access as the classes are divided into disjoint subsets.

Screen Shots

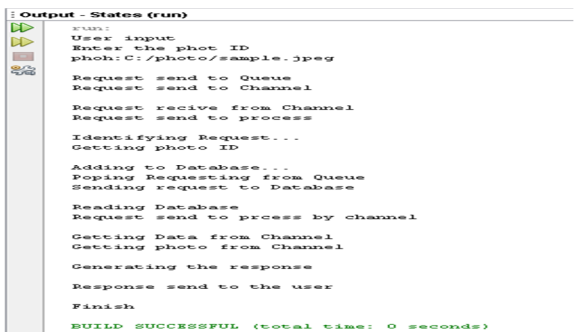

Fig. 5: Screen shot for read illustration

Analysis

The above screen shot displays the image following the request from the user. Each process interacts with its I/O Channel for a specific service and delivers the result. It exhibits the property of parallelism. In the above example, user wants to view the image and have the image ID as the input. It passes through each channel and process and collects the photo from database and send the image to the requested user. Each input, output and the status has been described in the screenshot.

I/O MODELS

An I/O model in which components of a file are partitioned into m classes and the accesses based on the request clause. This partitioned components works in parallel for the combination requests and produces the result in a non-deterministic, probabilistic and uncertain environment. Request can be anything like view image of all females or view the age of females belonging to software engineers. I/O automaton helps to implement the above three properties. Actions that can be performed are automated with all the properties and their sequence is explained in internal actions. This paper discusses the I/O model for read, write and modify operations. It must deal with shared access for the document during the parallel request. A concurrent model has been developed for read-write and read-modify-write. Each component has been implemented as hardware. Thus we can generalize our result with any set of actions, any property and the request. Following describes the two models for concurrent accesses.

Read-Write I/O Model

This section describes the I/O automaton model for read and writes operations on an XML document. For an I/O automaton, we have to describe the process I/O and Channel I/O. Each data that is traversed in the web is in the form of an XML document. This XML document is retrieved from the server and get back to the user. We have to restrict the document view through a hardware implementation. This has been done using a java program to perform read and write is presented in the Appendix C. we have to perform two operations on a document, read and write. First we will read the document and then write the contents of the file.

In the program described in the Appendix C, when the document reaches the system, it's received and wait in the Queue by the channel I/O using the send() function. When it is the first element in the queue, then it's removed from the queue a then send to process using the receive() function. All the classes are put in the dotask() function. The data have to be shared by the users in the system. We use the concept of shared variable for concurrent operations in an Xml document. The document is received using the init() function . Status is changed from idle to read. The document is read using readXML() and response is generated. If the response is unknown the status is changed to write and then perform the write operation using writeXML(). When the status is write, it will write the data as response and then changed the status to done. If the status is done, then output is sent for next action.

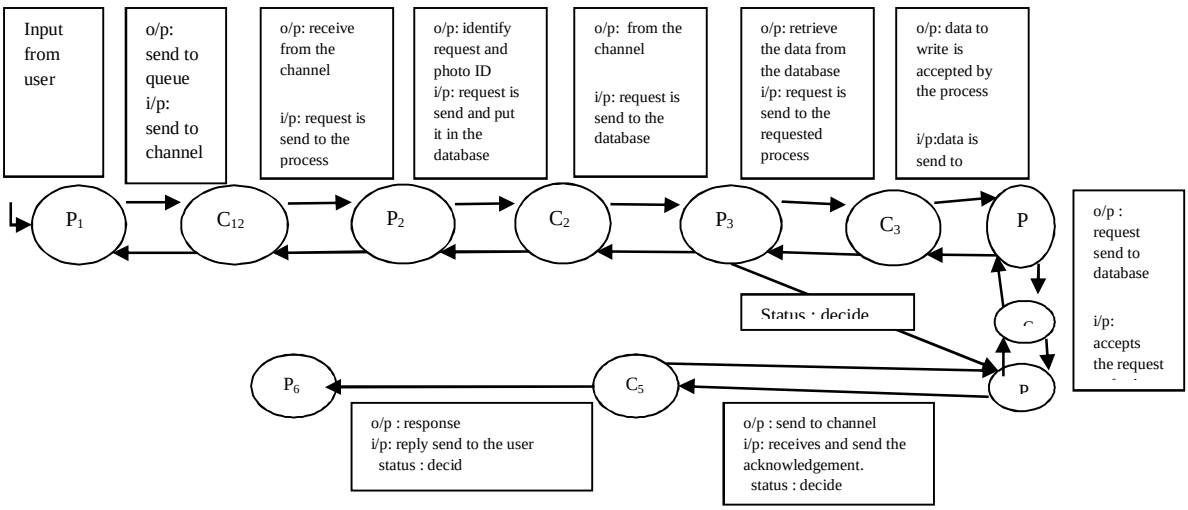

Fig. 6 : Transition diagram for READ – WRITE I/O Model

Screen Shot

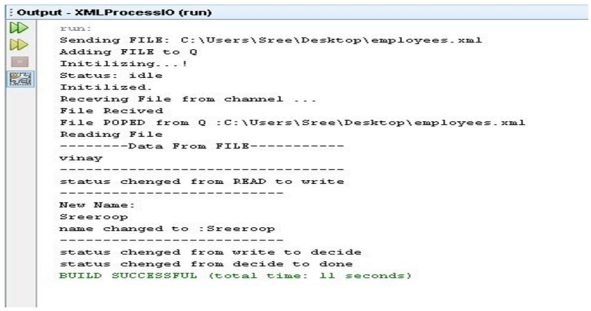

Fig.7: screen shot for READ – WRITE MODEL

Analysis

We create an XML file named employees.XML. At first the status is idle and then initialized and the status is changed to read when it collects the file from the channel using send() function. The data is collected for read operation. It performs the read operation and the response is displayed. Then we perform the write operation with a new name and the response is written to the XML file and displayed. It will check for the status and the operations are performed if and only if the status is read. The traversal of the document is displayed in the screenshot. The

Read-Modify-Write I/O Model

In READ-MODIFY-WRITE model, we are performing three operations in an XML document. This section describes the concurrent model for three concurrent requests for the same vector document. An arrival of multiple form of request on the xml document and its nature and sequences are described. First we will perform the read operation, and then modify operation and then the write operation. The sample program has been written in Appendix D.

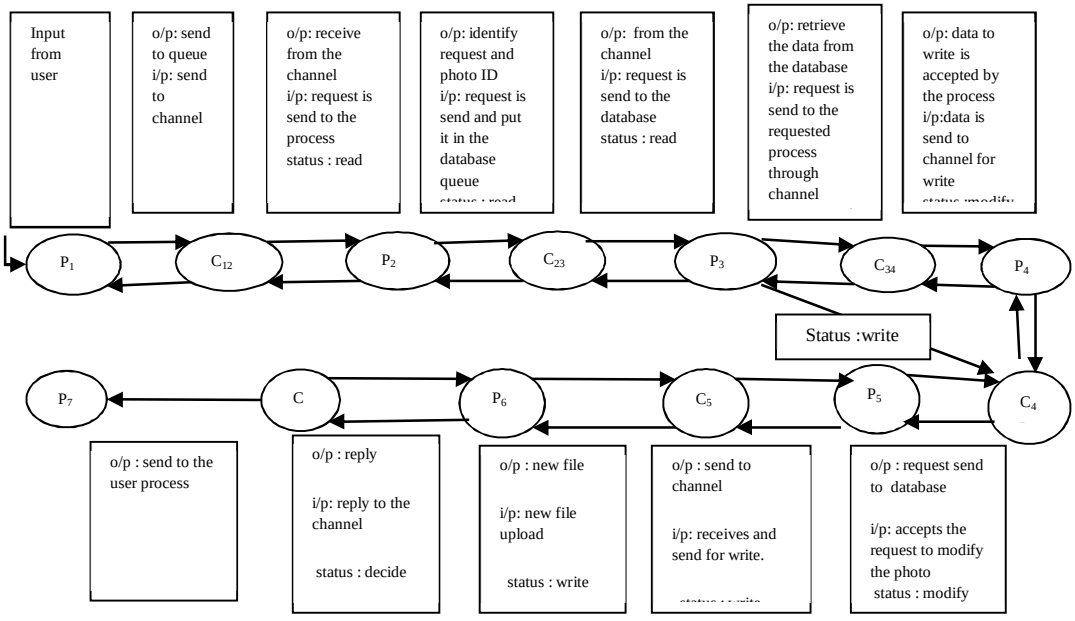

Fig. 8 : Transition diagram for READ – MODITY – WRITE I/O Model

Thus along with the program for read and write operations in READ-WRITE I/O model, we are adding a modifyXML() function which will update the data with the new value. Similar to I/O model it will send the value to channel using a send() function and then add the XML document to the queue. The first element of the queue is removed from FIFO list and send to process. The input is accepted using an init() function and at the initial state the status is idle.

When it need to perform the read, the status will be changed from idle to read and then performs the readXML and exit. If response is unknown, it will change the status to modify and perform the modifyXML() operation. After the work is over it will send the response as output. If the document didn't belong to any of the above and status is write, it will perform the write operation using writeXML().

Screenshot

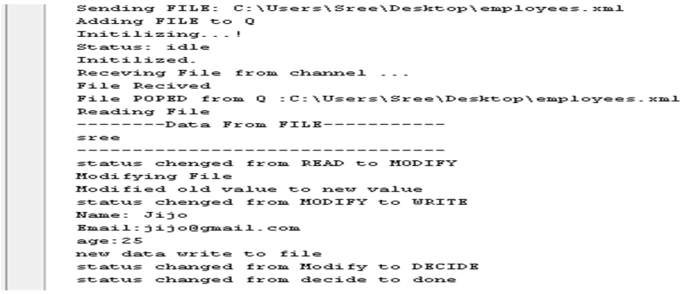

Fig.9 : Screen shot for READ – MODIFY – WRITE Model

Analysis

We create an XML file named employees.XML and have name as the content. When a request for the document arrives, it is checked against the preconditions and its corresponding effects are executed. At first the status is idle. At first, the statuses is checked and perform the read operation and then changes the status to read. After the read operation, modify operation is performed and the status is changed to modify and then the write operation is performed and then the status is set to decide and the operation is over on that document. The output is displayed in the screenshot.

FINITE STATE MACHINE MODEL

Finite state machine is a machine that accepts the language through the execution of each task with each state concurrent with the state of machine. Finite state machine accepts the language in the form of regular expression. Language consists of words which are basically drawn by input symbols. Machine undergoes successive state until all the symbols of the words arrived at the final state of the machine which is the accepting state of the machine. Set of all words is the member of the language. We consider read as a regular expression and constructed regular expressions for write and modify. In our earlier work, we have done the software implementation using the finite state machine using the prolog. We have the set of XML documents and we have constructed a finite state machine to perform the read, write and modify an XML document. Thus the users can perform three operations and we have constructed the transitions, state diagram an executed it with the help of java, XML and prolog. The following two sections describe the finite state machine for read-write and read-modify-write operations.

Finite State Read-Write Model

In the implementation of the finite state machine, we have a set of access rights against the set of XML documents. Language has been developed which is to be accepted by the finite machine. It describes the sequence of symbols accepted by the machine. Thus the language accepted by finite state automaton for read and write can be defined as L={username.password.c2.24135.Ha.b.g.G3.1.0c1.1.0.110.xml for read and write}. The request has been processed through each symbol till it reaches the accepting states. We constructed 11 states to implement the finite state machines for read and write. It has to traverse through each state and will accept the machine determined by the language. When we have to perform a read and write operation, if it accepts the above language and documents are fetched by pushing all the relevant documents using push down automata.

Finite State Read-Modify-Write Model

The language to be accepted for the finite state machine for read, modify and write has been developed. Thus the language accepted by finite state machine can be defined as L={usernamepassword.c1.12345.Ha.b.g.G2.1.0c1.1.0.111.xml for RWM}. The language determines the accepting condition to fetch the documents to perform read, write and modify operations. If the language is accepted, the access control matrix will have the list of documents that can perform the operation and will collect the whole document using the push down automata.

COMPARISON

Finite State Machine model is the software implementation of deterministic algorithm for fetching the documents based on the access right for each user class. An I/O automaton is the hardware representation of the non-deterministic algorithm. The finite state machine didn't take into account the asynchronous mode of operation. All can view, write or modify data and the concurrent access has to be dealt and the I/O automaton takes care of non-deterministic, stochastic and asynchronous mode of operations. The I/O automaton uses the shared variable and shared memory system to implement the concurrent access of the resources. The finite state machine didn't take care of the part of distribution of the shared documents. In finite state automata, we need to have the list of documents and the storage of the documents. the documents has to be taken from the stack and response The distribution aspects has not been met in the finite state implementation.

CONCLUSION

Taking into consideration the distributed nature of web service resource allocation and distribution ,we have developed a novel I/O automation. This is compared with the created models developed by us and found that it explains the simultaneous distribution of XML documents. This also includes time, lock-out, mutual exclusive properties. The component composition of shared system through shared processes. We have implemented the finite state automata in our earlier work and compared with the I/O automata. We can implement I/O automata for any document irrespective of the old or new based on the access. Thus we can develop a generalized hardware for automated access specific documents.

REFERENCES

A Rajeev, D.L.David, "*Theory Of Timed Automata*" Computer Science Department, Stanford University, Theoretical Computer Science 126, (1994), pp.183-235

B Johan & Yi Wang, "*Timed Automata: Semantics, Algorithms And Tools*", Uppsala University {Johanb,Yi} @It. Uu.Se.ACPN, (2003) LNCS 3098, Pp.87-124, (2004)

B Daniela, R D Fabio, S De Luca & M Massimo, "*Finite State Automata As Conceptual Model For E-Services*", Integrated Design and Process Technology, IDPT, USA, June (2003)

Chandy .K.M and Misra. J, "*The Drinking Philosopher's problem*", ACM Transactions on Programming and Systems, 6(4) pp.632-646, October (1984).

H Y.Joseph and Z.D.Lenove, "*A Little knowledge goes a long way, Knowledge based proofs for a family of protocols*", Journal of the ACM,39(3): pp. 449-478, July (1992).

H.E. John, M Rajeev, U. D. Jeffrey, "*Introduction To Automata Theory, Languages And Computation*". Pearson Education, Delhi, (2003)

L Nancy, "*DistributedAlgorithms*", Harcourt Asia Pvt Ltd, Morgan Kaufmann, India, (2000)

L Nancy, D.A. Richard, and M.J. Merritt, "*Cryptographic protocols*", In the Proceedings of the 14th Annual ACM Symposium on Theory of Computing ,PP:383-400,San Francisco, May (1982).

L. Nancy, T.R. Mark, "*An Introduction to Input/Output Automata*", CWI-Quarterly, 2(3): pp. 219-246, (September 1989).

L. Nancy, W. L. Jennifer, "*Synthesis of Efficient Drinking Philosophers Algorithm*", MIT/LCS/TM-417, 545 Technology Square, Cambridge, Massachusetts, 02139, November (1989).

M Saayan, K Ratnesh, B Samik, "*Automated Choreographer Synthesis For Web Services Composition Using I/O automata*", Department Of Electrical And Computer Engineering &Department Of Computer Science ,Ames, Iowa State Universityames, Iowa, (2009)

P.L. Gary, "*Concurrent Reading while writing*", ACM Transactions on Programming Languages and Systems, 5(1): pp.46-55, (1983)

P Francois, "*XML Processing Using Visibly Pushdown Automata*", Computer & Decision Engineering (CoDe) Department

S P. Munindr, H N. Michael, "Service-Oriented Computing Sematics, Processes, Agents", John Wiley Sons, INC, UK, 2005

Thiyagarajan. M, R. Chaitanya , "*Finite State Machine As Conceptual Model For Web Service Access*", Jokull Journal, Volume 65 :Issue No 4, May 2015

APPENDIX - A

Channel IO Pseudo code

```
ClassChannelIO
{
        String[] Qvalues={"m1","m2","m3","m4","m5","m6","m7","m8","m9"}
        String[] RQvalues=new String[10];
        intqSize=0;
        voiddoTask()
        {       for(String message:Qvalues)
                {
                reciveQValue(message);
                }
        }
        sendQValue(String meesage)
        {       addToQueue(meesage);
        }
        receiveQValue(String message)
        {       removeFirst(message);
        }
        addToQueue(String message)
        {
for(int i=0;i<RQvalues.length;i++)
         {      if(RQvalues[i].isEmpty())
                    {   RQvalues[i]=message;
                        qSize++;
                        break;          }               }
        }
        removeFirst(String message)
        {       if(qValue(0)==message)
                {  returnQvalues(i-1)
                } }
}
```

APPENDIX – B

Pseudo Code for Process IO

```
ClassProcessIO
{       String[] Valueset={"v1","v2","v3","....."vn",NULL};
        Object[] QChannellist={cq1,cq2,cq3,cq4,.......cqn}
        Queue Q;
        int[] cqSize={0,0,0,0,0.....};
        int i=0;
        Vector v;
        VoiddoTask()
        {       for every J !=i
                { sendQValue(Valueset(j),i,j);
                }
                decide(Valueset(j));
        }
        init(String value)
        {      Valueset(i)=v;
        }
        decide(String value)
        {       for(int j=1 j<=n;j++)
           {
```

```
        if(Valueset(j)!=Null)
              {   v=f(Valueset(1),.....Valueset(n));
              }
           }
        }
        sendQValue(String value,channeli,chanel j)
        {       Valueset(i)=v;
        }
        reciveQvalue(String value,channelj,channel i)
        {       Valueset(j)=v;
        }
}
```

APPENDIX – C

XML Read-Write Program

```
ClassXMLReadFile
    {  File input= test.xml;
       File output=null;
       String status="read";
       voiddoTask()
       {    for (File f:channel)
              {  send(f,i,j);
                 decids(file);
              }
       }
       VoidreadXML(File file)
       {       if (status=="read")
               {    if (x=0)
                       { output=input;
                         status="write";
                       }
                    else
                       {          output=x;
                         status="decids";
                       } }
       }
       void  writeXML(File file)
       {       if(status=="read" && file==input)
               {    output=input;
                    status="decids";
               }       }
       void decide(File file)
       {       if(status=="decids")
          {    output=file;
               status="done";
          }    }
       void send(File file,StringfromChannel,StringtoChannel)
       {       addFileToQ(file);
       }
       Voidreceive(File file)
       {       removeFileFromQ(file);
       }
}
```

APPENDIX – D

Read-Modify-WriteAlgorthms

```
classXMlReadFile
{           //input init(),receive()
            //Output decide(),send()
            voidreceiveXml(message,int j)
            {           if(qValue(0)==message)
                        {   returnqValue(i-1) }
            }
            voidinit()
             {   File input= XMLFile;
                File output=null;
                String status="idle";
                if(status=="idle")
            {       status="read";
             }    }    voidreadXML(File serviceFile)
      {          if (status=="read")
                    {   if (response=Null)
                            {  output=input;
                                    status="modify";
                                    } else
                                    {   output=response;
                                            status="decide";
                                    } }
            }
            void modify(File serviceFile)
            {      if (status=="modify")
                        {    if (response=null)
                                    {    output=input;
                                         status="write";
                                    }
                                    else
                                    {   output=response;
                                       status="decids";
                                    } }
            }
            void  writeXML(File serviceFile)
            {    if(status=="write" &&serviceFile !=null)
                        {  response=serviceFile;
                                    status="decide";   }           }
            void decide(File serviceFile)
            {      if (status=="decide")
                    {    output=serviceFile;
                        status="done";
                    } }
            voidsendXML( File response ,int i)
            {     qValue(i)=response;
            }
            voiddoTask()
            {    for (File f:channel)
                        { send(f,i,j);
                          decids(file);
                        } }
            }
```

Read-Modify-Write Algorithm

```
classXMLReadFile
{     //input init(),receive()
            //Output decide(),send()
            voidreceiveXml(message,int j)
            {           if(qValue(0)==message)
                        {    returnqValue(i-1)        }                                  }
            voidinit()
  {         File input= XMLFile;
            File output=null;
            String status="idle";
            if(  status=="idle")
            {    status="read";             }    }
            voidreadXML(File serviceFile)
            {                     if (status=="read")
                        {       if (response=Null)
                                    {   output=input;
                                    status="modify";
                                    }
                                else
                        {          output=response;
                                    status="decide";    }
                        }
            }
            void modify(File serviceFile)
            {
                        if (status=="modify")
                        {
                                    if (response=null)
                                    {
                                                output=input;
                                                status="write";
                                    }
                                    else
                                    {
                                                output=response;
                                                status="decids";
                                    }           }
            }
            void  writeXML(File serviceFile)
            {     if(status=="write" &&serviceFile !=null)
                        {    response=serviceFile;
                             status="decide";
                        }           }
            void decide(File serviceFile)
            {     if(status=="decide")
                  {    output=serviceFile;
                       status="done";
                  }      }
            VoidsendXML( File response ,int i)
            {      qValue(i)=response;   }
            VoiddoTask()
            {    for (File f:channel)
                        {  send(f,i,j);
                                    decids(file);     }
            }

}
```