International Journal Of Engineering Research & Management Technology ISSN: 2348-4039

Email: editor@ijermt.org July- 2014 Volume 1, Issue-4 www.ijermt.org

Modeling Access Control Behavior of Component based System in Uniframe Framework Using Temporal Logic of Actions

Dhowmya Bhatt
Research Scholar (CSE)
Mewar University, Chittorgarh

Dr. Ekata Gupta
Associate Professor
Krishna Institute of Engineering and Technology

ABSTRACT –
Uniframe is an approach that facilitates the interaction of heterogeneous software components. In a Uniframe framework, the entire working of the integrated system is transparent. While logic programming language like Prolog can be used to express the fact that the system resources will depend on the access guards and the policies developed to govern them, the Temporal Logic of Actions (TLA) is used to validate properties of the proposed system so that the behavioral aspects of the system implemented with access properties does not interfere with the component interactions at any level. This aspect of dynamicity is significant because of the fact that in an integrated system, as the components interact to execute an user command the access behavior should not interfere with the interfacing base of the components that have been pre-designed in a specific fashion by the system integrator. The research proposed in this paper will focus on the fact to verify the performance of the system using TLA after the access characteristics are partially implemented. The whole of the system performance will not be monitored and tested instead a portion of the proposed model will be taken and used for implementing TLA.

Keywords – Uniframe, access control, behavior, TLA, Liveness, component specifications

1. TLA – ORIGIN, DEFINITIONS AND IMPLEMENTATION

The temporal Logic of Actions (TLA) is a concurrent algorithm developed by Leslie Lamport in late eighties. Concurrent algorithms are those which are described with the program and if the algorithm is correct then the program will satisfy the properties that it has to. TLA followed this concept with modifications that the algorithms and their properties (Programs) can be expressed in the form of Logic Formulas. Applying the concepts of concurrency here, if the algorithm is correct then the formula related to the algorithm also implies the formula specifying the property. [ALEX02]. The TLA specifications are of two parts. The first part refers to the safety property and the latter is the vital Liveness property that refers to the next action that must take place. In short Liveness drives the actions step by step in any designed model. The newer version of the TLA is now called TLA+. Before completely going into the verification of access behavior of a system, certain general detail about TLA is done in the following content of this paper.
The general specification of a system can be given in TLA+ as given below and the terms used in the expression above are self-explanatory.

$^{\Delta}\text{Spec} = \text{Init} \wedge \Box[\text{Next}]\_{\text{variables}} \wedge \text{Liveness}$

- Init – represents the first or initial predicate and refers to a range of all possible values of variables in the system.
- Next – represents the relation between the current state of all variables and the consequent or the next state of the modelled system.
- □ – is called a box-pronounce and guarantees the fact that the formula used is always true for all its cases.
- Live ness – is a special feature of TLA+. This specifies the liveness essentials. Liveness is a property through which the system will be prompted to take the next action. Without the feature of Liveness,

the system might remain idle and might not move on or take the necessary action to proceed to the next step.

- [Next] $_{\text{variables}}$ – specifies that the Next is YES or TRUE for the system behaviour with access control implementation. This grants that the system does not execute weird behaviour.
- Init$\wedge$ □ [Next] $_{\text{variables}}$ – states that the initial step is TRUE and if the initial step satisfies this Init then every proceeding step will satisfy next.

Because of its significance and the fact that Liveness property makes sure that the model performs the necessary action by executing the next command, Liveness is represented as

Liveness == WF$_{\text{variable}}$ (Next)
WF – is the Weak Fairness expression of any Variable. So WF$_{\text{variable}}$ (D) states that is D is enabled then the step D occurs and also concurrency follows and is maintained.
After stating Liveness in the above mentioned form, the model will act and execute the necessary step. But without this property it may be assumed that model and system specification by the user will not integrate and will simply remain as two different entities and system specification will not direct the model to perform any action. TLA+ has a combination of expressions and notations that can be used to model a system with the access control point of view and the aim of this research is to analyse and describe the access control behaviour of the model with the state changes.

2. TLA – ANLAYSIS and PROBLEM SPECIFIC IMPLEMENTATION

Each specification is written as an individual module of TLA. A small example is taken to illustrate TLA specification.. A setup called TLA checker validates each TLA execution. The TLA model Checker to execute properly when a configuration file is specified that contains the value of each constant and that should identify the predicate value that specifies the particular specification.
The Configuration file in TLA is of the form specified below. The Specifications are given by the keyword Spec and the values of changing states are given by Invariant. A range of values is defined in the Value Set and at any given time the value of the Invariant after the initial execution should not exceed this set of values. There is a set of User Names and in which only certain users will be authorized.

```
SPECIFICATION Spec
INVARIANT Type Invariant
CONSTANTS
ValueSet = {1,2,3,4,5,6,7,8,9,10,11,12}
UserNames = {"dbhatt","khari","dpriya"}

AuthorizedUsers= {"dbhatt","khari"}
```

The example of a TLA Specification attempting to do a particular operation, say for example addition is given in the following table form.

```
EXTENDS Normal's
CONSTANTS Value Set, UserNames, AuthorizedUsers
VARIABLES value, username
Type Invariant == /\ value \in ValueSet
/\ username \in User Names
Init == /\ value =0
/\ username \in User Names
Check (u) == ud \in Authorized Users
Change == /\ Check (username)
/\ value' = value + 1
/\ UNCHANGED<<username>>
Next == /\ Change
```

Spec == Init /\ [][Next]_<< value, username>>

- EXTENDS – is a reserved word that will instruct the model checker to load those modules in addition to the current one that has predicates that have to be used in the current module.
- Normal's – is used to define an operation say for example addition.
- CONSTANTS – refers to those values that will not change. This usually mentioned in the configuration file. In this research, the configuration file is given separately.
- VARIABLES – refers to the changing values. It is understood that those variables in the extended module is also a part of current state or any state of the model in which it is used.
- /\ - represents logical AND operation.
- Type Invariant – this represents that value and username are the elements of the set of Value set and User Name.
- Check (u) – is the primary access control point. If ud is in the set of authorized users then Check(u) = TRUE
- Change – is a combination of three expressions. Firstly it checks for the Authorized Users. Secondly the value of the state changes. Value is the value of the variable in the current state and Value' is the value of expression in next state. Thirdly in all these state changes, the username should remain unchanged.
- <<>> – states that there may be a number of elements but all of them should be separated by commas.

Now that the definitions of TLA specifications are described, the next step is discuss how these specifications are executed in the various states of a model. The execution starts with Init which represents the initial state of the model. The change command is enabled when <<>> is TRUE. Next is used in all cases given as □ [Next].If there is not a Next then the system will not proceed to the net step and more of a deadlock situation. Also when a specific user name is not on the authorized list, the Next will enable further searching otherwise which the system will stop proceeding eventually leading to deadlock. The execution of the initial state will result in an invariant that is not a part of the Value Set. So under these two circumstances the model checker will discard this particular case.
But in reality this cannot always be possible to have the invariants remain as a single value instead they can be put within a range of values taken as the ValueSet. So this problem can be eliminated by changing the Change predicate to No Change. The values now can range anywhere from 0-12. This solves two purposes, one that the invariant policy is not changed and also the value of the invariant is within the range of the Value Set. The TLA specifications for No Change can be given as,

```
EXTENDS Normal's
CONSTANTS Value Set, User Names, Authorized Users
VARIABLES value, username
Type Invariant == /\ value \in Value Set
/\ username \in User Names
Init == /\ value =0
/\ username \in User Names
Check (u) == ud \in Authorized Users
Change == /\ Check (username)
 /\ value' = IF value#12 THEN value+1 ELSE 0
/\ UNCHANGED<<username>>
No Change == UNCHANGED<< value, user name>>
Next == \/ Change
\/ No Change
Spec == Init /\ [][Next]_<< value, username>>
```

The next step will be to provoke the model to function and execution to take place. So to the above predicate, the Liveness expression has to be added. The syntax to activate the Liveness expression so that the designed system will take the necessary continuous action is $\wedge WF_{<<value,name>>}(Next)$. This basic property discussion will serve as a roadmap to understanding TLA+. Further with regard to the research, there are few more important characteristics of TLA that will help to discuss the study taken to implement access control. One is the record that allows grouping of information as one variable. Data sending and receiving is one form where information has to be clubbed and shared. And also the information if the data is sent or received will have to be known by the channel. If there states are prepared, directed and responded and the channel that transmits the information can be represented through a record form. The initial state prepared can be given by[state|->"prepared", data|->""]

Consider a model that implements a set of components for an organization containing employee records. The model component will transmit and receive the response from the other components through

[state|->"directed", data|->"read employee record"]
[state|->"responded", data|->"employee record data"]

At any point of time to access the state of a channel the channel. State syntax is used. The previous, current or the next state of the channel can be accessed by variable channel expression.channel' = [channel EXCEPT!.State="prepared"] contains the expression! Which refers to the channel variations? The task is to implement the channel states to a set of employees. If E= {dbhatt, khari, dpriya} and the command set C={"save", "load"} then a set of information can be generated taking all the combinations of E and C through the expression commands = [e:Users, c:Commands]. The format in which the E and C can be combined and the changes done after the state value changes for the user dbhatt can be given by,

```
Commands = {
 [e|->"dbhatt", c|->"load"]
[e|->"dbhatt", c|->"save"]
                                        }
```

This can be done similarly for the other two users in the set as

```
Commands = {
 [e|->"khari", c|->"load"]
[e|->"khari", c|->"save"]
 [e|->"dpriya", c|->"load"]
[e|->"dpriya", c|->"save"]
                                        }
```

Once the user and the command functions are tested for state changes the next step is to model the variable to a function that can store the user security attribute. Now the function values can be mapped range of values by :> symbol and double-@ is used to group the multiple functions to a singleton. Taking the employee credentials used in [DHO02] the attribute of employee nala who is a peon canbe given as,

```
Attribute = ( "accessid" :> {"Nala"} @@
"Roles" :> {"Peon"}@@
"Groups" :> {"Employee"}
)
```

Now after the priorities have been assigned, retrieving the security feature in TLA is done by the operator "Choose". This will help select only those elements from the set of attributes that has dbhatt as its access Id.

User = CHOOSE x \in attributes: "dbhatt" \in x["access ID"]

For any information in general a particular portion can be accessible by [section] command in TLA

It is understood that there are sub-channels in a channel that will require the usage of multiple thread. This is done by @ as we mention as attribute ("roles"). The whole prospect of doing this is to present the channel in the form of function with the sub channels as domain that can contain a set of threads. This can be expressed by

Channel = [t\ in Threads |-> [state|->"prepared", data |->""]]

The access control policy is evaluated through testing a set of logical conditions and evaluating the result to YES or NO. This also implements the access control that enables only a particular employee to read and save any general information.

Policy Evaluator (e,p,o) == \/ (/\ r["section"]="gei"
/\ o="save"
/\ "owner" \in e["roles"]
)
\/ (/\ p[“section”]="gei"
/\ o="load"
/\ (\/ "employee" \in a["groups"]
\/ "peon" \in a["groups"]))

The next step will be to implement the guards wherever necessary and for this the guards must be given with certain information regarding the authen city of the user.

ERHS_ Guard (a, r, o) == Policy Evaluator ([a EXCEPT
!["roles"]=IF (r["er"] \in a["accessed"])
THEN a["roles"] \cup {"owner"}
ELSE a["roles"]], r, o)
Here the ERHS_ Guard operator is be a guard in the record server protecting access to any particular resource, and this can be used in other TLA expressions to control access related authorizations.

3. TLA IMPLEMENTATIONS in REAL TIME STUDY

The model designed in [Dho01] is used as such in establishing the real time study of an organization implementing TLA. Using certain functions in TLA to represent thread for the model specified in [Dho01] is inevitable. The thread is commonly known to have three states – dormant, ready and active. So each one function will represent one state of the thread. The first state is the dormant where the thread is available for the user. So when the process begins, all the threads will beautomatcally initialized to the dormant state. gt_threads = [t \in Threads |-> "dormant"]

The second variable contains the information about the ownership and purpose of each thread which is represented as a function mapping each thread to a record that holds the security attributes, the employee id for the information that is being accessed, the action that is to be taken within the system, and any additional data that needed to carry out the query.

gt_threads_data'= [gt_threads_data
EXCEPT![1]= [r |-> "dbhatt",
p |-> ("accessid" :> {"Khari"} @@
"Roles" :> {} @@
"Groups" :> {"Employee"}),
c |-> "load",
d |-> [gei |-> ""]]
d is the record field that will hold the data of the employee record containing the general employee information are loaded from the servers in the system that are initially left empty when the thread is assigned first . The next step is component specification and breaking each component into six parts. The generally used parts in component specifications are constant, variable, invariant, initialization, extension and the guard.

3.1 MODELING COMPONENTS UISNG TLA

Every individual component in the system has some variables which are used by that particular component only. Even in TLA these variables are not hidden and so these are used only by a single component. All the other components in the system already would have known about this and so these variables cannot be changed by them. But it is important that the components specify about such variables clearly and assign values for them so that confusion is avoided. The specific elements of the components may be constants also. Apart of constants and variables the component may also consist of invariants. For example in the General Terminal of the model taken for study, contains no variables but only constants uses two different predicates to initialize dormant thread and another predicate to validate the thread.

```
GT_Init == /\ gt_threads = [t \in Threads |-> "dormant"]
        /\ gt_threads_data=[t \in Threads |-> "empty"]
GT_Inv == /\ gt_threads \in [Threads ->Thread States]
```

Now after the threads are initialized, the threads have to PICK the user commands and process them. The second function will be READ and send data across the channel. If both these conditions are satisfied then the state of the thread is changed to active. Then the state of the channel will now become SENT and the Employee Record Server (ERS) is given with the information that the data has been Sent.

```
GT_Pick_Request(t) == /\ gt_threads[t]="dormant"
         /\ Cardinality (command_set)#0
/\ gt_threads_data'=[gt_threads_data EXCEPT ![t]= CHOOSE x \in
command_set:x#[p|->("":>""),r|->"",c|->"",d|->""]]
/\ command_set' = command_set \ {gt_threads_data'[t]}
/\ gt_threads'=[gt_threads EXCEPT ![t]="ready"]
/\ UNCHANGED<<principals,attributes>> ]
/\ UNCHANGED<<ut_ers_save>>

GT_Send_Read(t) == /\ gt_threads[t]="ready"
/\ gt_threads_data[t].c="read"
/\ gt_ers_read[t].state = "replied"
/\ gt_threads'=[gt_threads EXCEPT ![t]="dormant"]
/\ gt_ers_read'=[gt_ers_read EXCEPT ![t]=[state|->"ready",
data|->gt_threads_data[t],
Message |->("":>"")]] **//The message to be communicated is written here//**
/\ UNCHANGED<<gt_threads_data>>
/\ UNCHANGED<<command set, principals, attributes>>
/\ gt_threads'=[gt_threads EXCEPT ![t]="active"]
/\ gt_ers_read[t].state = "ready"
/\ gt_ers_read'=[gt_ers_read EXCEPT ![t]=[state|->"sent",
data|->gt_threads_data[t],
Message |->("":>"")]]
/\ UNCHANGED<<gt threads data>>
/\ UNCHANGED<<command set, principals, attributes>>
```

The next step is to receive a reply from the ERS for which still the thread has to be kept in an active state. The command is READ and the channel state is REPLIED. If the condition stands TRUE then the state of the channel is changed to READY.

```
GT_Receive_Read(t) == /\ gt_threads[t]="active
 /\ gt_threads_data[t].c="read"
 /\ gt_ers_read[t].state = "replied"
/\ gt_threads'=[gt_threads EXCEPT ![t]="dormant"]
/\ gt_ers_read'=[gt_ers_read EXCEPT ![t]=[state|->"ready",
data|->ut_threads_data[t],
Message |->("":>"")]]
/\ UNCHANGED<<gt_threads_data>>
/\ UNCHANGED<<command_set, principals, attributes>>

Waiting == /\ \A t \in Threads: gt_threads[t]="dormant"
/\ Cardinality (command_set)=0
/\ UNCHANGED<<gt_threads data, gt_threads>>
/\ UNCHANGED<<command set, principals, attributes>>
```

As the usual procedure of using threads suggests, the WAITING command is implemented as the system should not remain dead when there are no commands to process. The guards are also implanted is such a way that if the request to a particular server is authorized then the information is returned or else the query is denied by returning an empty function stating that the requested data is unavailable to this particular user. The guard will be implemented through a conditional statement.

```
IF ERS_Guard (gt_ers_read[t].data.p,
(“esr“ :>gt_ers_read[t].data.r @@ “section“ :> "gsi" @@ "part" :> "all" ),
gt_ers_read[t].data.c )
THEN gei[gt_ers_read[t].data.r]
ELSE ("":>"")
```

The above states were explained to indicate how models are built in TLA with channels, predicate system naming and access policies. Each component has its own policy specification and the system integrator all these together to form the system specification.

3.2 COMPOSING SPECIFICTIONS DESIGNED THROUGH TLA:
Though policy specification is easy, composing these specifications is a tougher task. Each variable of a particular component adds to the entire system so some predicates will interpret for certain local but unchanged variables. Since components may be developed or added not by the same developer, it is essential that when one component takes one action, all the other components will continue unchanged. This has to done through interfacing user actions to the system. The full-fledgedsystem architecture provides the information about the expressions used in commands, security attributes, and also the operator for processing access control, guards and policies. TLA designs can be modeled to interact with other resource and thereby the components. For this a list of all the constants, predicates, variables used in the modules of Employee Record Server designed in [DHO01] is listed and considered. Implementing all these, the EIS can be given in a simplified form which will consists of,

- Constants, Variable, Commands
- threads, thread-states
- Employee information
- Staff information
- Attribute information
- Access information – guards, policy evaluation.

This can be expressed in a combination of User attributes and commands as,

```
EXTENDS: Finite-Sets, \TLC
CONSTANTS Threads, Thread States, Employee, Staff, Command
VARIABLES attribute principals, command set, gt_ers_read, gt_threads, gt_threads_data, gei
CH_Init == gt_ rs_read = [t \in Threads |-> [state|->"ready",
data |->[p|->("":>""),r|->"",c|->"", d|->""],
Message |->("":>"")]]**//content//**
User_Init == /\ attributes = {( "accessid" :> {"dbhatt"} @@
"roles" :> {} @@
"groups" :> {"Employee"} ),
**//User_Init == /\ attributes = {( "accessid" :> {"khari"} @@
"roles" :> {} @@
"groups" :> {"employee"} ),//**
( "accessid":> {"nala"} @@
"roles" :> {"peon"} @@
"groups" :> {"staff"} ),
/\ principals = Employee {\union Staff}
/\ command_set = [p: attributes, c:{"read"}, r:Employee, d:{[gei|->""]}]
User_Inv == /\ principals = Employee {\union Staff}
/\ command_set \in SUBSET
[p:attributes, c:Command, r:Employee,
d:{[gei|->""]}]
```

using these compositions as the basics, the simplified TLA specifications for each of the modules proposed in the real time study can be put together and given as,

```
EXTENDS Finite Sets, TLC
CONSTANTS Threads, Thread States, Employee, Staff, Command
VARIABLES attribute principals, command set, gt_ers_read, gt_threads, gt_threads_data, gei
{
**\\Executable of principles and commands\\**
}
Policy Evaluator (e,p,o) ==\/ ( /\ r["section"]="gei"
/\ o="save"
/\ "owner" \in a["roles"] )
\/ ( /\ r["section"]="gei"
/\ o="load"
/\ ( \/ "employee" \in a["groups"]
\/ "staff" \in a["groups"]
) )
**\\read and send requests\\** GT_Init == /\ gt_threads = [t \in Threads |-> "dormant"]
/\ gt_threads_data=[t \in Threads |-> "empty"]
GT_Inv == /\ gt_threads \in [Threads ->Thread States]
GT_Prepared_Request(t) == /\ gt_threads[t]="dormant"
/\ Cardinality (command_set)#0
/\ gt_threads_data'=[gt_threads_data EXCEPT ![t]= CHOOSE x \in
command_set:x#[p|->("":>""),r|->"",c|->"",d|->""]]
/\command_set' = command_set \ {gt_threads_data'[t]}
/\ gt_threads'=[gt_threads EXCEPT ![t]="ready"]
/\ UNCHANGED<<principals,attributes>>
/\ UNCHANGED<<gt_ers_read>>
```

```
/\UNCHANGED<<gei>>
**\\Sending data\\** GT_Send_Read(t) == /\ gt_threads[t]="prepared"
/\ gt_threads_data[t].c="read"
/\ gt_threads'=[gt_threads EXCEPT ![t]="active"]
/\ gt_ers_read[t].state = "prepared"
/\ gt_ers_read'=[gt_ers_read EXCEPT ![t]=[state|->"sent",
data|->gt_threads_data[t],
Message |->("":>"")]]
/\ UNCHANGED<<gt_ threads_ data>>
/\ UNCHANGED<<command_ set, principals, attributes>>/\UNCHANGED<<gei>>
**\\data waiting due to netwrk traffic\\** Waiting == /\ \A t \in Threads: gt_threads[t]="dormant"
/\ Cardinality (command_set)=0
/\ UNCHANGED<<gt_threads_data, gt_threads>>
/\ UNCHANGED<<command_set, principals, attributes>>/\ UNCHANGED<<gei>>

GT_Receive_Read(t) == /\ gt_threads[t]="active"
/\ gt_threads_data[t].c="read"
/\ gt_rs_read[t].state = "responded"
/\ gt_threads'=[gt_ threads EXCEPT ![t]="dormant"]
/\ gt_ers_read'=[gt_ ers_ read EXCEPT ![t]=[state|->"ready",
data|->gt_threads_data[t],
Message |->("":>"")]]
/\ UNCHANGED<<gt_ threads_ data>>
/\ UNCHANGED<<command_ set, principals, attributes >>
```
/\UNCHANGED<<gei>>

```
General Terminal (t) == \/ GT_Pick_ Request (t) \/ GT_Send_ Read(t)
\/ GT_Respond_Read(t) \/ Waiting
ERS_Init == gei = ( "dbhatt" :><< "Divya", "Bhatt", "Ghaziabad", "GZB" >> @@
"Khari" :><< "Kumar", "Hari", "Delhi", "DL" >> @@
"priyad" :><< "Priya", "Singapore", "SNG" >> )
ERS_Read(t) ==
/\ gt_ers_read[t].state = "sent"
/\ gt_ers_read'=[gt_rs_read]
EXCEPT![t]=[state|->"responded",
data|->ut_rs_read [t].data,
Message |-> ( "gei" :> (IF RS_Guard (gt_ers_read [t].data.p,
("er" :>gt_ers_read[t].data.r,
"section" :> "gei",
"part" :> "all" ),
gt_ers_read[t].data.c)
THEN gei[ut_ers_read[t].data.r]
ELSE ("":>"") ) )]]
ERS_Guard(a,r,o) == Policy Evaluator( [ a EXCEPT
!["roles"]=IF (r["er"] \in a["accessID"])
THEN a["roles"] \cup {"owner"}
ELSE a["roles"]],
r, o )
Employee Record Server(t) == ERS_Read(t)
Type Invariant == /\ GT_Inv
```

```
/\ User_Inv
Init == /\ GT_Init
/\ ERS_Init /\ User_Init/\ CH_Init
Liveness == WF]_<<attributes, principals, command_set, gt_ers_read,
gt_threads, gt_threads_data, gei>>(NEXT)
Next == \E t \in Threads: ( GeneralTerminal(t) \/ Record Server(t) )
Spec == Init /\ [][Next]_<<attributes, principals, command_set, gt_ers_read,
ut_threads, ut_threads_data, gsi>> /\ Liveness
```

4. TLA MODEL CHECKING USING TLA+ CHECKER

The model implemented using TLA can be checked with the tools provided by the TLA+ itself expect fr the fact that they cannot be checked in an automatic fashion. While TLA supports existential operators, the model checker does not support a temporal existential operator. The function of the model checker can be explained through a simple syntax- Java tlc.TLC Small Employee Record and in model checker reverts by checking the semantics, properties and the possible initial states.

```
TLC Version 2.0 of Jul 29, 201
Model-checking
Parsing file SmallEmployeeRecord.tla
Parsing file C:\tla\tlasany\StandardModules\TLC.tla
Parsing file C:\tla\tlasany\StandardModules\FiniteSets.tla
Parsing file C:\tla\tlasany\StandardModules\Naturals.tla
Parsing file C:\tla\tlasany\StandardModules\Sequences.tla
Semantic processing of module Naturals
Semantic processing of module Sequences
Semantic processing of module TLC
Semantic processing of module Finite Sets
Semantic processing of module Small Employee Record
```

The model working can be checked by running this for finding out errors, the reachable states, number of states generated, if there are or not distinct states, if queue is formed or not. Depending on these parameters, a depth graph can be drawn. Most importantly it helps to establish if or not a change in the policy will deadlock the system. Also the safety parameters that indicate an impossible situation and the Liveness parameter that every action has a response is mainly tested using the TLA Checker. The last aim of the checker is to confirm the validity of the specifications with coordination to the access control policy, like the general information of the employee has to be available for him to read.

```
Read_self == \A t \in Threads :( (/\ ENABLED (GT_Send_Read(t))
/\ gt_threads_data[t].r \in ut_threads_data[t].p["ACCESSID"]
/\ ut_threads_data[t].c="read")
~>
(/\ ENABLED (GT_Receive_Read(t))
/\ gt_ers_read[t].message#("":>"")) )
```

5. CONCLUSION

The aim f this research paper was to reconnoitre the modelling of component behaviour and the proposed Employee Information system using the Temporal Logic of Actions. Towards the end of the research it was found that, it is possible to validate the composed TLA specifications to be free from dead locking the system and possesses the propose aliveness properties to provide the information. This will help the system integrator with information that is usually used to statically validate a system composition before the model deployment and installation and also allow for testing the access control policies that depend the access control functionalities.

6. REFERENCES

1. Dhowmya Bhatt, Ekata Gupta "Empathy of Access Control Characteristics of Heterogeneous Software Components in Uniframe Framework, IJARCSSE May 2013
2. Dhowmya Bhatt, Ekata Gupta "Modeling Access Control Characteristics of Components in Uniframe Framework using Logic programming",
3. Alexander M. Crespi "An access control model for Uniframe Framework," M.S. Thesis, Department of Computer & Information Science, Indiana University Purdue University Indianapolis, May 2005.
4. Burt, Bryan Raje, Olson, and Auguston, "Model Driven Security- Unification of Authorization Models for Fine-Grain Access Control," Proceedings of EDOC - 2003, The VII IEEE International Enterprise Distributed Object Computing Conference, Brisbane, Australia, September 2003.
5. Sun, "QOS Composition and Decomposition in UniFrame," M. S. Thesis, Department of Computer & Information Science, Indiana University Purdue University Indianapolis, June, 2003.
6. Khan, K. Han, "Security Characterisation Framework for Trustworthy Component Based Software System," Technical Report No-CIT-35-2003, University of Western Sydney, School of Computing and Information Technology, May 2003.
7. Minsky, Ungureanu, "Unified Support for Heterogeneous Security Policies in Distributed Systems," Proceedings of the VII USENIX Security Symposium, Berkeley, California, 1998.
8. Robert, S. "Concepts of Programming Languages," Wesley Addison, 2002.
9. Harrington, Jensen "Cryptographic Access Control in a Distributed File System," Proceedings of the eighth ACM symposium on Access control models and technologies, pages 158-165, Italy, 2003.
10. Object Management Group (OMG), "Model-Driven Architecture: A Technical Perspective," Technical Report, OMG Document 1-2-01/04 of February 2001.
11. Sandhu, R., Coyne E., Feinstein, H., Youman, C. "Role-Based Access Control Models," IEEE Computer, vol. 29, no. 2, pp 38-47, 1996.
12. Katzke, S. "Future Directions of the Common Criteria (CC) and the Common Evaluation Methodology (CEM)," National Institute of Standards and Technology, 2002.
13. Abendroth, J., Jensen, C. "A Unified Security Framework for Networked Applications," Proceedings of the 2003 ACM symposium on Applied Computing, pages 351-357, Melbourne, Florida, 2003.
14. Huang, Zian., "The UniFrame System Generative Programming Framework," M.S. Thesis, Department of Computer & Information Science, Indiana University Purdue University Indianapolis, April 2003.
15. Raje, Rajat, Auguston, M., Bryant B., Olson, A., Burt, C. "A Unified Approach for the Integration of Distributed Heterogeneous Software Components," Proceedings of the 2001 Monterey Workshop Engineering Automation for Software Intensive System Integration, pages 109-119, Monterey, California, 2001.
16. (IETF), "A URN Namespace of Object Identifiers," RfC 3061, February 2001.

About The Authors

Ms. Dhowmya Bhatt received her B.Tech degree in IT in the year 2003 from M.K University and M.Tech in CS & IT in 2005 from M.S. University and was awarded as the "outstanding student" M.Tech batch. She started her career with the IT industry and later switched to teaching and presently has an experience of six years. She has authored many research papers in various International journals. Currently she is pursuing her doctoral degree and her research interests are Network security and access control.

Dr. Ekata Gupta is a doctoral degree holder and an Associate Professor in Krishna Institute of Engineering and Technology with a decade long career in the Teaching. She has participated in various National and International conferences all across India and has shared her research ideas. Dr. Ekata has authored above 30 research papers and her current area of interest is Neuro –Fuzzy, a comparison and combination of Neural Network and Fuzzy concepts. Her passion for research has taken her places and she has held various posts in her carrier. She is presently guiding about 5 research scholars who are pursuing their doctorate degree.