Indiana University Purdue University Indianapolis

Department of Computer and Information Science

CSCI 53700

Fall 2018

TEAM PROJECT

ON

VERIFICATION OF WEB SERVICES USING TLA+

TEAM MEMBERS

Hima Varsha Anne

Jasvir Kaur

Sai Krishna Devanapally

Tejaswi Parasa

GUIDED BY

Prof. Dr.Rajeev Raje

Contents

1. PROBLEM STATEMENT

For any distributed system to be fault-tolerant, it is very important for the system designers to verify the correctness properties of that system which depends on checks, error-correction mechanisms and security measures at different programming levels. Similarly, for a distributed web service it is crucial to be error tolerant for it to run smoothly in real world scenarios. Considering this importance, a new method to verify the services using Temporal logic of actions has been introduced. Using TLA+, specification for individual components of a single web service are described. These components if combined could be well utilized for services like room or movie booking systems.

2. INTRODUCTION

It is very crucial for any distributed or concurrent system designers to be able to specify and verify the correctness of the system behavior. Many standards to develop such systems have been introduced but they lack application of formal methods. In order to verify these non-sequential systems, using simple and efficient formal properties that could incorporate the behavior of the systems and verify the correctness properties, the formal specification language TLA+ is used.

With the emerging technology, for any business application to be fully utilized it requires the support of compositing distributed and autonomous web services. Although there are number of formal modelling methods proposed for service composition such as Petrinet, OWL-S, Pi-Calculus, situation calculus etc., they have certain limitations to verify the completeness and correctness of a service. TLA+ helps in resolving these issues by specifying a system and its properties as logical formulas allowing conduction of reasoning without any translation to verify a system for correctness.

In this project, we put forward TLA+ specifications for four individual services like bank transaction system to verify the transaction properties, resource manager that makes sure one-to-one user and resource allocation, critical section that guarantees allocation of a resource to only one process and a packet transfer mechanism to ensure the serialization properties. These specifications are then checked with the TLC model checker whose results show the error probable cases based on the conditions provided.

2.1. TEMPORAL LOGIC OF ACTIONS

Temporal logic of actions, a brainchild of Leslie Lamport, is a logic for specifying and reasoning about the concurrent and distributed systems. It's syntax and few properties are summarized in about few lines. A typical specification of TLA is of the form: **Init /\ [] Next**

Where, Init – A formula to describe all initial states

Next – A formula that specifies all possible next states of a behavior in a system

In TLA, a formula is valid if and only if it satisfies all the mentioned behaviors. Any TLA spec contains the following typical symbols: /\ (and), \/ (or), ∃(there exists), ∀(for all), [](always true), <>(eventually true)

3. DESIGN AND IMPLEMENTATION DETAILS

3.1. Critical Section:

It is a section in which only one Resource manager can enter at one time. In this we wrote the specifications of critical section as well as Non critical section.

```
VARIABLE process

TCTypeOK == process \in [RM -> {"1", "0"}]

Init == process = [r \in RM |-> "0"] \* /\ [s \in RM |-> "0"]

Critical(r) == /\ process[r] = "1"
               /\ process'[r] = [s \in RM |-> IF s # r THEN process[r] = "0" ELSE process[s] = "0"]


Noncritical(r) ==  /\ process[r] = "0"
                   /\ process' =  [s \in RM |-> IF s = r THEN process[r] = "1" ELSE process[s] = "0"]
                 \*  /\ \E s \in RM CASE process[s] -> "1" [] OTHER process[r] = "0"

Next == \E r \in RM : Critical(r) \/ Noncritical(r)
```

Fig 1

Here, RM is the resource manager and r, s belongs to RM. If RM is in critical section the it belongs to 1 else it is 0. so, after entering the critical section, the next stage of RM will be in Non-critical section and vice versa. The above snippet shows Next stage which gives the OR operator between Critical and Non-critical section.

3.2 Bank Transfer:

It tells about the specifications of transferring the amount from one account (say A) to another (say B) and we have main variables such as sender, receiver and balance. Here 'Case' tells that transfer procedure occurring between two different systems concurrently. We specified the fixed amount of both the account holders equal to 50 dollars and amount transferred to be 1$. Here we are checking the consistency of the system that total account before and after transfer is same that is 100$. Init tells the initial states of all different variables.

```
VARIABLES sender, receiver, balance

vars == << accHolders, account, pc, sender, receiver, balance >>

Case == (1..2)

Init == (* Global variables *)
        /\ accHolders = {"A", "B"}
        /\ account = [p \in accHolders |-> 50]
        (* Process Transaction *)
        /\ sender = [self \in 1..2 |-> "A"]
        /\ receiver = [self \in 1..2 |-> "B"]
        /\ balance \in [1..2 -> 1..account[sender[CHOOSE self \in  1..2 : TRUE]]]
        /\ pc = [self \in Case |-> "Transfer"]
```

Fig 2

Following is the code snippet for transfer and deposit states. Here balance tell the amount to be transferred and it should be equal to or less than the sender account. If it is less than the sender amount then it will subtract the balance amount from sender's account and add to the receiver's account. But the total amount before and after transfer is same.

```
Transfer(self) == /\ pc[self] = "Transfer"
                  /\ IF balance[self] <= account[sender[self]]
                        THEN /\ account' = [account EXCEPT ![sender[self]] = account[sender[self]] - balance[self]]
                             /\ pc' = [pc EXCEPT ![self] = "Deposit"] /\ UNCHANGED << accHolders, sender, receiver, balance >>
                        ELSE /\ pc' = [pc EXCEPT ![self] = "Done"]
                             /\ account' = account /\ UNCHANGED << accHolders, sender, receiver, balance >>

Deposit(self) == /\ pc[self] = "Deposit"
                 /\ account' = [account EXCEPT ![receiver[self]] = account[receiver[self]] + balance[self]]
                 /\ pc' = [pc EXCEPT ![self] = "Done"]
                 /\ UNCHANGED << accHolders, sender, receiver, balance >>

Transaction(self) == Transfer(self) \/ Deposit(self)
```

Fig 3

3.3. Resource Manager:

When a request from a client comes to a resource in distributed or multitenant environment, Resource Manager makes sure that only one request is allocated to a resource at any given point of time. When there are multiple requests to use the resource, next requests are kept in a queue. When the current request is done with the resource it deallocates the resource and the next request from the queue is allocated.

This specification checks the above-mentioned process by taking the queue value to Boolean that is when the queue is non empty its value is true and when it empty it is set to false. Resource has two states, one is allocate which means it is being used and the other is deallocate which means it is done processing and is ready for other requests to use. Request has two states one is processing and another is done which mean the request is in queue or the request has been completed respectively. The specification checks the validity of the system by making sure at any given point of time there is only single request using the resource.

```
Init == (* Global variables *)
        /\ queue = TRUE
        /\ resource = "allocate"
        /\ pc = [self \in ProcSet |-> CASE self = "resource" -> "Deallocate"
                                        [] self = "request" -> "Processing"]
(*Initial state of the system when their is a process in the queue and the Resource manager is empty*)

Deallocate == /\ pc["resource"] = "Deallocate"
         /\ IF queue
               THEN /\ resource' = NextOperation(resource)
                    /\ pc' = [pc EXCEPT ![“resource"] = "Deallocate"]/\ UNCHANGED queue
               ELSE /\ pc' = [pc EXCEPT ![“resource"] = "Done"]
                    /\ resource' = resource /\ UNCHANGED queue
(*Resource manager state after being done with current process*)

resource_ == Deallocate

Processing == /\ pc["request"] = "Processing"
         /\ resource = "release"
         /\ queue' = FALSE
         /\ pc' = [pc EXCEPT ![“request"] = "Done"]
         /\ resource' = resource
(*Resource manager state when the request gets accepted into the process manager*)
```

Fig 4

The above snippet shows the initial state of the system where there is a request in the queue and the resource is already in use and a program counter to keep a track of the state of the system at a given point of time. The deallocate and processing are different possible states of the system.

3.4. Packet Transfer:

This behavior which we are checking can also be termed as serialization behavior. Given a scenario where multiple requests come from the same client to a server and all the requests must be processed in the same order sent for correct execution or else the system fails. The reason for naming this specification as packet transfer is, this can even check a behavior of system when packets are transported from point A to point B and the sequence of execution must be in the same order as sent.

```
Init == \* The initial state
    /\ location = [Left |-> {"Packet1","Packet3","Packet2"}, Right |-> {}]
    /\ currentLocation = "Left"
    /\ carrier = ""
    /\ PreviousPacket = ""
    /\ log = << >>
```

Fig 5

The above snippet shows the initial state where packet 1, packet 2, packet 3 are on the client side which is mentioned as left side and they need to be transported to the right side. And the current location of the pointer is on the left side along with which packet is on the carrier and a log to keep a track of transactions.

```
Transfer1 == \* Sending the first available packet
     /\ carrier = ""
     /\ PreviousPacket # ""
     /\ Allowed(location[currentLocation])
     /\ currentLocation' = OtherLocation(currentLocation)
     /\ PreviousPacket' = ""
     /\ log' = Append(log, "")
     /\ PrintT(log')
     /\ UNCHANGED <<location, carrier>>
```

Fig 6

The above snippet shows the transfer of one packet from left side to right side and making sure the location, carrier of other processing on the system are unchanged. And the last transaction is added to the log.

4. RESULTS

4.1 Critical Section:

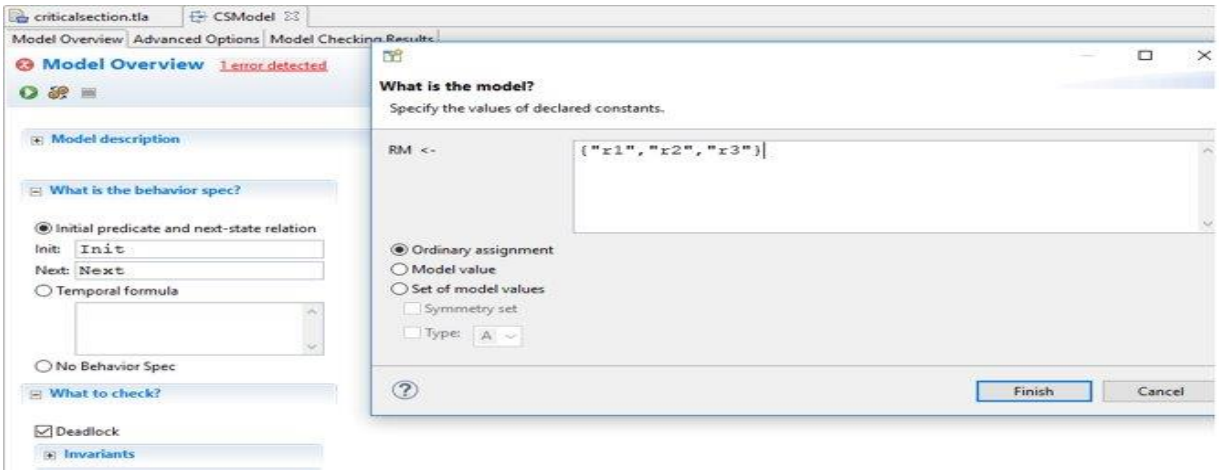


Fig 7

Above figure shows the output screen of model where we specify ‘r1’, ‘r2’, ‘r3’ as different models where they try to enter in critical section at same time then it throws an exception.

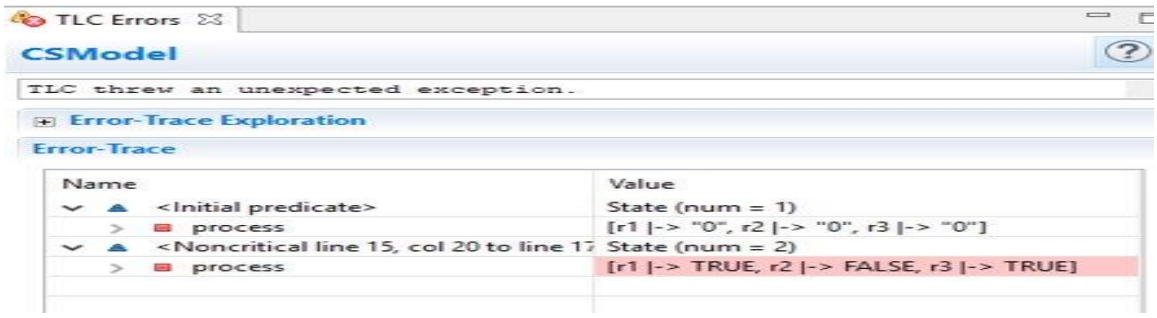


Fig 8

4.2 Bank Transfer:

We put an invariant ‘ account[“A”] + account[“B”] =100’ in the model and when we execute it will show an error trace upon the occurrence of this invariant as shown in following figure.

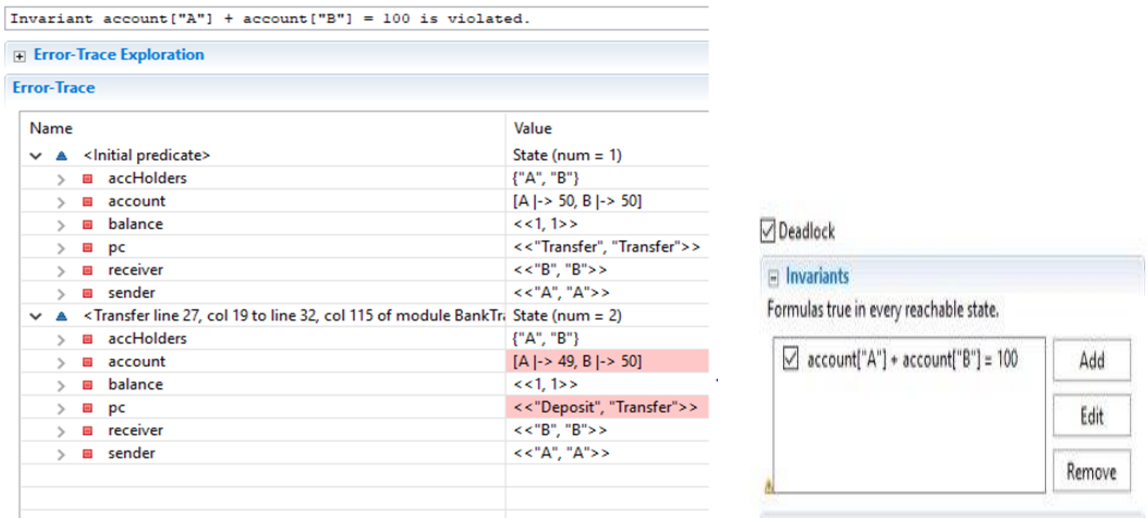


Fig 9

4.3 Resource Manager:

Being a scenario where the behavior doesn’t have a termination condition since, if all the resources are allocated to only one request the specification doesn’t come to an end. So, we will be checking the behavior of the system using invariants which will act as a termination condition, if the system changes from the state specified invariant the check will come to an end and the error trace will be displayed.

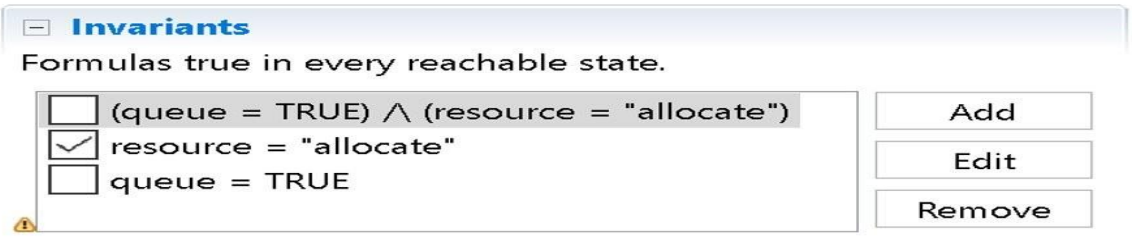


Fig 10: Invariants for Resource Manager Spec

For the scenario where the resource allocation is set as an invariant, the model checker results is as follows:

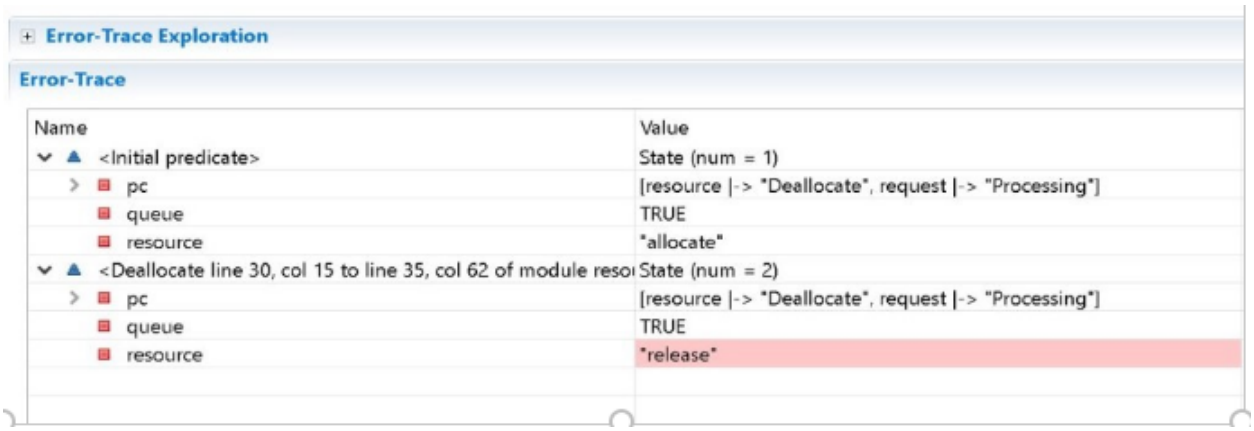


Fig 11: Error Trace for invariant ’resource = allocate’

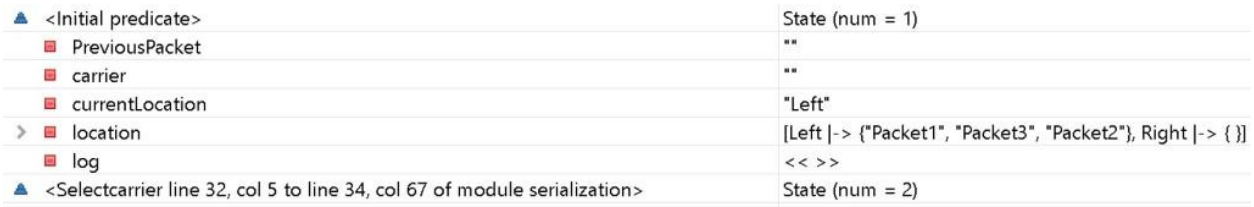
The above error trace shows that the state of resource has been changed from allocate to release they're by violating the invariant stopping the execution of the model. In this similar way we can check different behaviors of the system and we can also check if the system has run into a deadlock. In this specification there is no deadlock and if any of the conditions is violated and the system runs into a deadlock, we can identify it with ease.

4.4 Packet Transfer:

Packet transfer behavior can be checked with the help of a deadlock. Here the deadlock occurs when there are no packets on the client side to send to the server side. If there is no deadlock checked, then the system will run into around 4475 states and terminate when there is are no possible states to go ahead.

Fig 12: Deadlock in TLC model checker

Here we are checking a scenario of sending 3 packets from client to server. To check the behavior of the system the result must be receiving all the packets in same order of sending them.



<Transfer2 line 47, col 5 to line 55, col 19 of module serialization>	State (num = 3)
PreviousPacket	"Packet3"
carrier	""
currentLocation	"Right"
location	[Left |-> {"Packet1", "Packet2"}, Right |-> {"Packet3"}]
log	<<"Packet3">>
<Transfer2 line 47, col 5 to line 55, col 19 of module serialization>	State (num = 8)
PreviousPacket	"Packet3"
carrier	""
currentLocation	"Left"
location	[Left |-> {"Packet1", "Packet3"}, Right |-> {"Packet2"}]
log	<<"Packet3", "", "Packet2", "Packet3">>
<Transfer1 line 37, col 5 to line 44, col 38 of module serialization>	State (num = 11)
PreviousPacket	""
carrier	""
currentLocation	"Left"
location	[Left |-> {"Packet3"}, Right |-> {"Packet1", "Packet2"}]
log	<<"Packet3", "", "Packet2", "Packet3", "Packet1", "">>

Fig 13: TLC Model checker error trace for the deadlock condition

The above results show the wrong packet is transferred at the first try and as the model advances the correct packet is sent and at the end packet are received in the same order as sent and when all the packets are received from client side the model ends in a deadlock and system comes to an end.

5. CONCLUSION

In the expanding world of heterogeneous system connectivity between devices is a compulsory along with reliability of systems, as their size increases the reliability must not decrease. To assure that property TLA+ along with TLA toolbox and model checker act as the perfect way to check the behavior of a large system.

In this project the use of TLA+ to write the specifications of web service components can be used as individual templates to check the behavior of individual components of a web service. These specifications can also be modified to work in different scenarios depending on the requirement.

6. FUTURE WORK

The modelled individual web services could be combined into a single service using the composition techniques in TLA+ to verify the correctness properties. The verification of a composite web service is very crucial in real time applications. Using TLA, we could verify if the composed service satisfies the required properties.

7. CRITIQUE

This section will give overview of what the work that has been done in our project “Verification of web services using TLA+” and how it can be used in real and large-scale distributed systems.

The main aim of the project was to write a specification for a composite web service using TLA+ which can be used as a template to check the behavior various real time scenarios. To achieve this, first different components of a webservice have been identified. Examples of components can be given as bank transaction, resource manager and packet transfer. Once the components have been identified and the specifications are written accordingly with correctness, making sure all possible behaviors are defined. The second step is to combine all the different components into one single system and verify the behavior of a system as a complete entity. This complete system can be used to verify the behavior of large systems. This project can be used in large scale distributed systems by adding additional components such as data integrity checker, checking security protocols, validating requests and retry procedures.

In this project the part of integrating different components of a web service is not functioning properly due to an unknown lexical error. If the model was integrated completely the system will be able to check web service as a whole. By adding different components which we have specified above we can make a web service such as a booking system where a resource manager can make sure of only one client can book a ticket at a given time multiple clients are trying to book the same ticket, Bank transaction can be used to check the payment, packet transfer can be used to check the sequence of requests making sure first come first served policy is achieved.

To conclude, this project has a great potential to check the behavior of large-scale distributed systems and all their possible behaviors making sure no unwanted states reach in the system or a deadlock occurs in the system. By checking the specification of system for different stages in SDLC the correctness of a system can be assured from the start of a project.

8.REFERENCES

[1] https://lamport.azurewebsites.net/tla/tla.html

[2] “An Efficient Approach to Compose Web Services” Hongbing Wang, Hui Liu, Xiaohui Guo School of Computer Science & Engineering, Southeast University, China.

[3] “A Logic-based Approach to Web Services Composition” and Verification Hongbing Wang, Chen Wang, Yan Liu, The School of Computer Science & Engineering, Southeast University, China.

[4] “Specify and Compose Web Services by TLA1” Hongbing Wang, Hui Liu, Xiaohui Guo School of Computer Science & Engineering, Southeast University, China.

[5] “Describing and Verifying Web Service Composition using TLA Reasoning” Hongbing Wang, Qianzhao Zhou, Yanqi Shi School of Computer Science and Engineering, Southeast University, China