Toward Proving the Correctness of TCP Protocol Using CTL

Rafat Alshorman
Department of Computer Science, Yarmouk University, Jordan

Abstract: *The use of the Internet requires two types of application programs. One is running in the first endpoint of the network connection and requesting services, via application programs, is called the client. The other, that provides the services, is called the server. These application programs that are in client and server communicate with each other under some system rules to exchange the services. In this research, we shall try to model the system rules of communications that are called protocol using model checker. The model checker represents the states of the clients, servers and system rules (protocol) as a Finite State Machine (FSM). The correctness conditions of the protocol are encoded into temporal logics formulae Computational Tree Logic (CTL). Then, Model checker interprets these temporal formulae over the FSM to check whether the correctness conditions are satisfied or not. Moreover, the introduced model of the protocol, in this paper, is modelling the concurrent synchronized clients and servers to be iterated infinite often.*

Keywords: *CTL, model checking, TCP protocols, correctness conditions, kripke structure.*

Received January 28, 2017; accepted March 21, 2017

1. Introduction and Related Work

As concurrent users request and provide the Internet services in terms of program applications, a system rules are needed to make this process work correctly. These system rules are called 'protocol'. In this paper, we introduce a technique, based on model checking and Computational Tree Logic (CTL), to prove the correctness conditions of the protocol. Most of trials to prove the correctness of network connections protocols consider the number of concurrent clients and servers to be bounded [6, 8, 9, 15]. Also, they use pure mathematical proof or computer simulation. The problem of using computer simulation is that all possible situations, of the protocol, cannot be covered. And, mathematical proofs required an expert people to conduct such proofs or sometimes it is difficult to express the abstract model of the protocol in mathematical Equations [3, 16]. In this research, we assume that the number of concurrent clients and servers is bounded, but they are iterated infinitely many times. The importance of such assumption has been recognised from nowadays applications where the clients and servers requesting and providing services in a continuous stream [1]. This research proposes a protocol, based on Transmission Control Protocol/ Internet Protocol (TCP/IP) protocol. TCP/IP protocol is the Internet Protocol Suite (usually abbreviated TCP/IP) which developed to be the standard and basis of the global Internet and computer networking [5]. We presume that the server is connection-oriented concurrent type. This means that several clients requesting services at the same time and the server serve multiple clients concurrently and independently using TCP (i.e., connection-oriented) as a transport layer protocol. The ultimate aim of this research is to give a new notion of how to prove protocols where number of requested services are iterated infinitely many times. This paper is organized as follows. In section 2, we shall introduce the model of concurrent clients and server. The Kripke structure and the CTL syntax and semantics are discussed in section 3. In section 4, we define the correctness conditions of the proposed protocol and their corresponding CTL formulae for infinitely many clients requesting services from a server. The NuSMV model for the proposed protocol is given in section 5, and the conclusions are drawn in section 6. NuSMV script for the proposed model is added in Appendix A.

2. A Model of Concurrent Clients and Servers

The model of clients and servers is one of the most popular models in the computer networking. In this model, the clients request a service (or multiple services) from a server such as email service, file transfer, etc., the servers provide the services for the requesting clients. Moreover, servers allow concurrent clients to obtain a given service without having to wait for the server to finish previous requests. Both clients and servers communicate with each other by application programs. To illustrate this model in a clearer way, assume that a user has two windows opened simultaneously and running two applications: one that retrieves and displays email, the other that connects to download a file. Each application is considered to be a client requesting a service from a

particular server. Or, assume that the server provides two different services: email and shared files. One or more clients can request these services from one terminal, with two or more opened applications, or from different terminal, with one request for each terminal. The concept of terminal, in this paper, means any device that terminates one endpoint and requests a service such as Personal Computer (PC), Personal Digital Assistant (PDA), Smart Phone, etc., See Figures 1 and 2.

2.1. The proposed Model

In this Subsection, we demonstrate the proposed model of the TCP/IP protocol that will be abstracted into a special kind of finite state machine called Kripke structure. And, the correctness conditions of the abstract model are encoded into CTL. This will be clarified in details in the next Section. In this research, we assume that the server provides two services P_1 and P_2. Each service has a socket i.e., soc_1 and soc_2. The Sockets are created and used to order the client requests as a queue. So, for each service in the server, there is a queue that contains the clients requests in the same order as they arrived. In this case, the client will wait its turn to serve. As the queue is a buffer, it will be bounded. If the queue is full of requests and a new request is arrived, then the server will reject that request. The server will be busy if and only if all sockets queues are full. We shall denote the state where the server does not have any request to serve as idle. This means that all sockets queues are empty. The server will be in process state if and only if each socket queue neither full nor empty. Moreover, we assume, in this research, that the clients concurrently request the services infinite often.

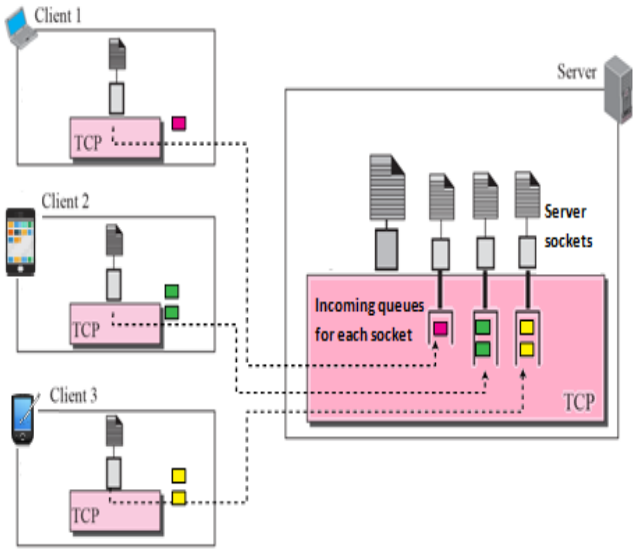

Figure 1. Clients and servers model.

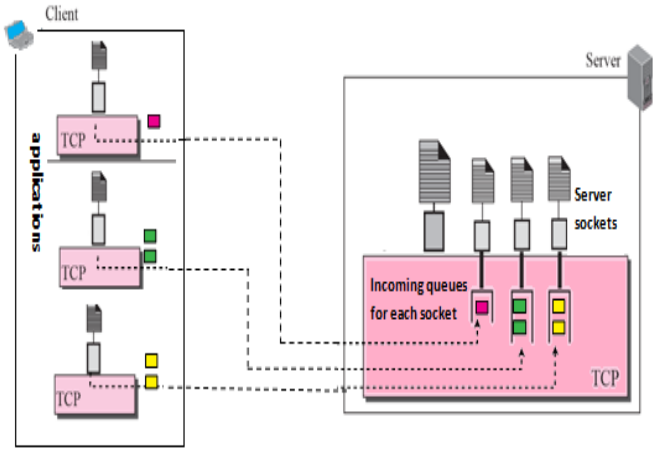

Figure 2. Application programs as clients.

This assumption means the repetitions (or iterations) of the n clients will constitute the infinite number of requests. This is not investigated in the literature. All researchers presume that the number of clients are bounded [6, 9]. Therefore, this research will give a stepping stone to model and prove unbounded number of clients that are requesting a services from server (servers). The importance of such assumption is emerged from nowadays applications where the number of requests incoming to the server in continuous stream and iterated infinitely many times. But, at any point in time, there is bound number of clients receiving the services. For that, it is very important to prove that this assumption keep the heavy use of the protocols is correct such as TCP/IP protocol [12, 17].

3. Kripke Structure and Temporal Logics

Kripke structure is a special type of finite state machine for representing finite state model. Each state, in kripke structure, is labelling with a set of atomic propositions that are true in this state [2, 10, 11]. Formally, it can be defined as follows

- *Definition* 1: A kripke structure M is defined by a tuple (S, I, R, AP, L), where

S: is a finite set of states

$I \subseteq S$: is a finite set of initial states.

R: is a total transition relation such that $R \subseteq S \times S$.

AP: is the set of atomic propositions

L: is a function which labels each state with the set of atomic propositions that are true in that state such that $L : S \to 2^{AP}$.

In this research, we shall model the proposed protocol using synchronous kripke structure. This means that the components of the protocol change their state variables simultaneously, such as clients, servers and sockets queues. To achieve that, we shall use NuSMV Model checker, see [4], to create a kripke structure that describing the proposed protocol. The correctness conditions and properties, that the proposed protocol

should satisfy, will be encoded into temporal logic formulae, as we will see in later in this paper. Now, given a description of a Kripke model M, path (in M) λ and a property (or correctness condition) expressed in a temporal logic formula Φ, the model checker decides whether $M, \lambda \models \Phi$ holds. This means that the formula Φ is interpreted over the kripke structure Malong path λ. The model checker returns true if formula Φ is satisfied. Otherwise, it returns false provided with counterexample.

3.1. CTL Syntax and Semantics

CTL is a temporal logic where the next time is branching. This means that every state has several successors. As CTL is interpreted over branching-time structures (like Trees), it contains path quantifiers to evaluate the formulae over the set of paths, see [14].

3.2. CTL Syntax

A CTL formula Φ consists of a set of atomic propositions, that are used in the proposed protocol, such as *idle*, *req*, *rej*, *wait*, *rec*, *comp*, p_1 and p_2, ordinary Boolean operations $\neg, \vee, \wedge, \rightarrow, \top, \perp$, quantifiers Existential (E), A (Universal), and temporal operators X, F, G and U. Formulae in *CTL* can be generated by:

$$\phi ::= pr_i \mid \neg\Phi \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 \wedge \Phi_2 \mid \mathrm{AX}\Phi \mid \mathrm{E}[\Phi_1 \mathrm{U} \Phi_2]$$

$$\mathrm{EX}\Phi \mid \mathrm{AF}\Phi \mid \mathrm{EF}\Phi \mid \mathrm{AG}\Phi \mid \mathrm{EG}\Phi \mid \mathrm{A}[\Phi_1 \mathrm{U} \Phi_2]$$ Where pr_1, pr_2,... are any atomic propositions used in the proposed protocol.

3.3. Semantics of CTL

We will start by defining when an atomic proposition pr true at a state/time s_i such that:

$M, s_i \models pr$ iff $\in L(s_i)$, for all $pr \in pr_i$.

The semantics for the other ordinary operators are defined as follows:

$$M, s_i \models \neg \phi \quad \text{iff } M, s_i \not\models \phi$$

$$M, s_i \models \phi \wedge \psi \text{ iff } M, s_i \models \phi and \; M, s_i \models \psi$$

$$M, s_i \models \phi \vee \psi \text{ iff } M, s_i \models = \phi \text{ or } M, s_i \models \psi$$

$$M, s_i \models \phi \Rightarrow \psi \text{ iff if } M, s_i \models \phi \text{ then } M, s_i \models \psi \; M, s_i \models \top$$

$$M, s_i \not\models \perp$$

The CTL operators have the following semantics where $\lambda=(s_i, s_i+1,\ldots)$ is a generic path outgoing from states$s_i$in the model M

$$M, s_i \models \mathrm{AX}\Phi \text{ iff } \forall \lambda = (s_i, s_i + 1, \ldots) \; M, s_i + 1 \models \Phi$$

$$M, s_i \models \mathrm{EX}\Phi \text{ iff } \exists \lambda = (s_i, s_i + 1, \ldots) \; M, s_i + 1 \models \Phi$$

$$M, s_i \models \mathrm{AF}\Phi \text{ iff } \forall \lambda = (s_i, s_i + 1, \ldots), \exists \; j \geq i \; M, s_j \models \Phi$$

$$M, s_i \models \mathrm{EF}\Phi \text{ iff } \exists \lambda = (s_i, s_i + 1, \ldots), \exists \; j \geq i \; M, s_j \models \Phi$$

$M, s_i \models \mathrm{AG}\phi$ iff , $\forall \lambda = (s_i, s_i + 1, \ldots)$, and$\forall j$ $j \geq i, M, s_j \models \phi$

$M, s_i \models \mathrm{EG}\phi$ iff $\exists \lambda = (s_i, s_i + 1, \ldots)$, and $\forall j$ $j \geq i, M, s_j \models \phi$.

$M, s_i \models \mathrm{A}[\phi_1 \mathrm{U} \phi_2]$ iff $\forall \lambda = (s_i, s_i + 1, \ldots)$, $\exists$ $j \geq i$ such that $M, s_j \models \phi_2$ and,$\forall k, i \leq k < j$, $M, s_k \models \phi_1$

$M, s_i \models \mathrm{E}[\phi_1 \mathrm{U} \phi_2]$ iff $\exists \lambda = (s_i, s_i + 1, \ldots)$such that,$\exists \; j \geq i, M, s_j \models \phi_2$ and,$\forall k, i \leq k < j$, $M, s_k \models \phi_1$.

Now, we can notice that the future in CTL is branching or there is more than one path we can go through [14].

4. The Model of Iterated Concurrent Clients and Server

In this section, we shall presume that the clients request the services from the server in a continuous manner. As the number of clients and services are finite and the requests are not ending, each request will iterate infinitely many time. The proof of such model is not trivial. So, we need to take this new constraint in our account. Also, to express the correctness conditions of such model, we need to choose a language contains operators that can deal with this constraint. In this paper, we shall use CTL (as expressiveness language) to encode and express the model and its properties with the additional constraint. Now, It is important to conduct proofs, of infinite many requests iterated in the server, using fully automated techniques (Model Checkers) to avoid disadvantages of manual (or traditional) proofs.

To demonstrate that, as in section 2, we shall give the following example:

Consider that we have a sever S that contains m services P_1, P_2,... P_m. Each service has a socket i.e., soc_1 and soc_2. Also, we have n clients C_1, C_2,..C_n. Now, we shall denote to the client C_i requesting service P_j by R_i^j. Moreover, we shall denote to the sequence of

requesting a service P_j from a sever S by R_S^j.The sequence of will be of the form

$$R_S^j = \underbrace{\cdots R_l^j R_i^j R_k^j \cdots R_k^j \cdots R_i^j \cdots R_i^j R_l^j R_k^j}_{\text{Clients may iterate infinitely many}} \cdots$$

$1 \le j \le m, 1 \le i, k, l \le n$. This depiction corresponds a path in the kripke structure that we discussed in Section 3. Now we can use CTL to encode the correctness conditions and the properties of the proposed model. Thus, model checker, such as NuSMV, can be used to verify whether these conditions and properties are true or false. In case of false, counterexample will be generated by model checker to demonstrate the set of states, in the model, that may aggregate together to violate the one of the correctness conditions that is encoded in CTL.

4.1. Correctness Conditions of the Proposed Model

In this section, we shall introduce some correctness conditions for synchronous processes in general and for the proposed model in specific. These conditions will form a framework (or stepping stone) for specifying such synchronous processes and protocols that are dealt with. Now, we shall assume that the proposed model will be correct (or working in a correct manner) if and only if the following conditions and properties are satisfied:

1. Each client requests a service should eventually complete it. This condition is encoded into CTL as follows:

$$\omega_1 = \bigwedge_{1 \le i \le n} \bigwedge_{1 \le j \le m} AG\,(R_i^j \Rightarrow AF\,Co_i^j) \quad (1)$$

We add an extra proposition called Co_i^j to indicate that the client C_i complete the execution of service P_j. This condition asserts that the clients will eventually progress and will not starved forever. Moreover, if the above condition satisfied for each client, then we can say that the proposed protocol is starvation free.

2. In this paper, we assume that the client requesting the processes infinitely often. This is can be encoded as follows:

$$\omega_2 = \bigwedge_{1 \le i \le n} \bigwedge_{1 \le j \le m} (AG\,R_i^j \Rightarrow AF\,Co_i^j) \wedge (Co_i^j \Rightarrow AF\,R_i^j) \quad (2)$$

The above condition is asserted that if the client C_i requested the service P_j and completed the execution of it, then the client C_i will iterate the request infinitely often

3. If the client C_i request process P_j from the server and the socket queue, associated to that process, is not full, then the process will be added to the queue. This can be encoded such that:

$$\omega_3 = \bigwedge_{1 \le i \le n} \bigwedge_{1 \le j \le m} AG((qu_j = t \wedge R_i^j) \Rightarrow AX\,(qu_j = (t+1))) \quad (3)$$

The proposition qu_j denotes to the socket queue that is associated to the service P_j. $t$is the number of clients requesting the service P_j and $t < k$, where k is the size of the queue.

4. If the socket queue has an empty space for any process P_j, then the server should not reject the client:

$$\omega_4 = \bigwedge_{1 \le i \le n} \mathrm{AG!}(\mathrm{C}_i^R \wedge \mathrm{qu}_i \ne \mathrm{k}) \quad (4)$$

The proposition c_i^R denotes to the case that the is rejected (or in reject state)

5. The server will not enter an unreachable state (busy):

$$\omega_5 = \bigwedge_{1 \le i \le n} \mathrm{AG}\ (\mathrm{AF} \neg(\mathrm{S}^b)) \quad (5)$$

The proposition S^b means that the server S is in busy state.

6. If the client is rejected, the client will eventually progress:

$$\omega_6 = \bigwedge_{1 \le i \le n} \bigwedge_{1 \le j \le m} AG\,(C_i^R \Rightarrow AF\,Co_i^j) \quad (6)$$

Now, we shall build a model checking called $TCP_{i,j}$ which can be reduced to CTL model checking. So, given a Kripke structure M, a states$_a$, and a formula $\emptyset \in \mathrm{TCP_{i,j}}$, we have that

M is a $TCP_{i,j}$ structure,

iff, for all $s_a \in S$,

$$M, s_a \models \emptyset,$$

Where

$$\emptyset = \bigwedge_{1 \le i \le 6} \omega_i.$$

Now, it is clear that $\emptyset \in$CTL.

5. The Corresponding NuSMV Model

In this section, we shall describe the proposed model in the NuSMV language to make sure that the proposed TCP model meet the correctness conditions that we introduced in the previous section. NuSMV language is low-level language for describing a Finite State Machine (FSM). Usually, some aspects of the protocol are difficult to model. One of these aspects is the queue associated to each socket is modelled as a variable queuesoc1 (orqueuesoc2) being increment or decrement depending on the request and the completion of the service.

5.1. State Variables of NuSMV Model

In this subsection, we will provide description of some correctness conditions that have been written, in the previous section, against system model described in the NuSMV model. We have used CTL for this purpose. In general, temporal logics are suitable formalism for reasoning about critical and concurrent systems [7]. In the proposed TCP protocol, the desired properties and their conditions are (see section 4):

1. Each client requests a service should eventually complete it as in Equation (1)

$$\omega_1 = \bigwedge_{1 \le i \le n} \bigwedge_{1 \le j \le m} AG\,(R_i^j \Rightarrow AF\,Co_i^j)$$

This condition can be written in NuSMV language as follows:

SPEC AG(c1.state=req ->AF c1.state=comp)
SPEC AG(c2.state=req ->AF c2.state=comp)
SPEC AG(c3.state=req ->AF c3.state=comp)

In simple English, the SPEC denotes to CLT Specification in NuSMV language and the operators AG and AF have the same meaning as they defined in section 3.3. The other properties and conditions can be analyzed and encoded in a similar way. The full NuSMV model code is in Appendix A.

5.2. Observations and Results

While checking the correctness of the abovementioned properties and conditions in NuSMV, we found that all properties and conditions, that are introduced in section 4, hold in all situations, see Figure 3. Moreover, to show the powerful of NuSMV, we add the following condition:

SPEC AG (qu1.queuesoc1=-1 -> AX qu1.queuesoc1=-1)

This condition assert that, at any point in time, if the queue associated to the socket 1 is empty (the value of queuesoc1 equals -1), then always in the next state will remain empty. This means that no client will request the service number one from the server. The Model checker NuSMV falsifies this condition and give us counterexample, see Figure 4.

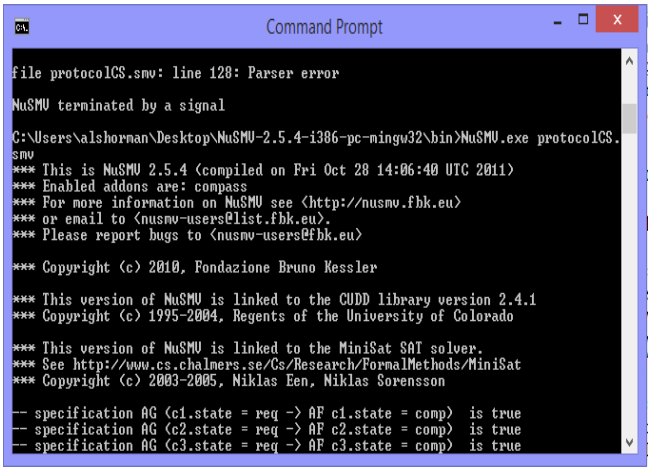

Figure 3. Example of NuSMV run for correct condition.

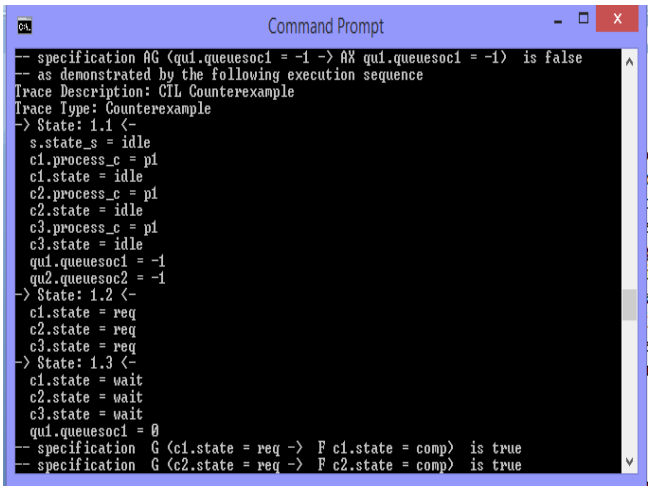

Figure 4. NuSMV Counterexample.

6. Conclusions

The ultimate objective of this study was to introduce and use CTL in the context of synchronous processes contending to share resources according to some protocol rules. This kind of protocols are widely used in networking, security and mobile computing. This study has given us a stepping stone to prove the correctness conditions of such protocols. Moreover, this study shows that, in some situations, we can encode and prove the correctness of infinitely many processes iterated in a concurrent system. In this paper, we have shown that the proposed TCP protocol and its properties can be encoded and proven using CTL. Also, we have introduced a transition structure to model the synchronous processes and its properties in terms of propositions. The proof part can be executed by NuSMV model checkers, to test whether the proposed protocol satisfies our correctness conditions or not. In case of no, counterexamples will be produced by model to show the errors. This represents the disprove example in traditional mathematical proofs. We have found that CTL is suitable for encoding synchronous processes.

This approach gives an automatic proof method that can overcome the obstacles of the traditional mathematical proofs techniques such as the user should know how to use and apply mathematical theorems, human error, some properties and conditions cannot be modelled mathematically in a way as we intended, and sometimes, we are not be able to cover the all possible system situations and this will not consider to be a proof as in the simulation [13].

References

[1] Adalid D., Salmeron A., Gallardo M., and Merino P., "Using SPIN for Automated Debugging of Infinite Executions of Java Programs," *Journal of Systems and Software*, vol. 90, no. 5, pp. 61-75, 2014.

[2] Alshorman R. and Hussak W., "A CTL Specification of Serializability for Transactions Accessing Uniform Data," *International Journal of Computer Science and Engineering*, vol. 3, no. 5, pp. 26-32, 2009.

[3] Casoni M., Grazia C., Klapez M., and Patriciello N., "Implementation and Validation of TCP Options and Congestion Control Algorithms for Ns-3," *in Proceedings of the Workshop on ns-3*, Barcelona, pp.112-119. 2015.

[4] Cimatti A., Clarke E., Giunchiglia F., and Roveri M., "NuSMV: A New Symbolic model Verifier," *in Proceedings of the 11th International Conference on Computer Aided Verification*, London, pp. 495-499, 1999.

[5] Comer D., *Computer Networks and Internets*, Pearson, 2014.

[6] Debiao H., Jianhua C., and Jin H., "An ID-Based Client Authentication with Key Agreement Protocol for Mobile Client-Server Environment on ECC with Provable Security," *Information Fusion*, vol. 13, no. 3, pp. 223-230, 2010.

[7] Gnesi S., "Formal Specification and Verification of Complex Systems," *Electronic Notes in Theoretical Computer Science Netherlands*, vol. 80, pp. 294-298, 2003.

[8] Gray D., Hamilton G., and Sinclair D., "Four Logics and a Protocol," *in Proceedings of the 3rd Irish Conference on Formal Methods*, Swinton, pp. 79-102, 1999.

[9] He C., Sundararajan M., Datta A., Derek A., and Mitchell J., "A Modular Correctness Proof of IEEE 802.11i and TLS," *in Proceedings of the 12th ACM Conference on Computer and Communications Security*, Alexandria, pp. 2-15. 2005.

[10] Hussak W., "Monodic Temporal Logic with Quantified Propositional Variables," *Journal of Logic and Computation*, vol. 22, no. 3, pp. 517-544, 2012.

[11] Hussak W., "Serializable Histories in Quantified Propositional Temporal Logic," *International Journal of Computer Mathematics*, vol. 81, no. 10, pp. 1203-1211, 2004.

[12] Ibrahim S., Idris B., Munro M., and Deraman A., "Integrating Software Traceability For Change Impact Analysis," *The International Arab Journal of Information Technology*, vol. 2, no. 4, pp. 301-308, 2005.

[13] Kerber M., Lange C., and Rowat C., "An Introduction to Mechanized Reasoning," *Journal of Mathematical Economics*, vol. 66, pp. 26-39, 2016.

[14] Pucella R., "The Finite and the Infinite in Temporal Logic," *ACM SIGACT*, vol. 36, no. 1, pp. 86-99, 2005.

[15] Ran G., Zhang H., and Gong S., "Improving on LEACH Protocol of Wireless Sensor Networks Using Fuzzy Logic," *Journal of Information and Computational Science*, vol. 7, no. 2, pp. 767-775, 2010.

[16] Salmeron A. and Merino P., "Integrating Model Checking and simulation for Protocol Optimization," *Simulation: Transactions of the Society for Modeling and Simulation International*, vol. 91, no. 1, pp. 3-25, 2015.

[17] Shatnawi m., "Discrete Time NHPP Models for Software Reliability Growth Phenomenon," *The International Arab Journal of Information Technology*, vol. 6, no. 2, pp. 124-131, 2009.

Rafat Alshorman is an assistant professor in the department of computer science at Yarmouk University/Jordan. Dr. Alshorman completed his Ph.D. at Loughborough University/UK and his under graduate studies at Yarmouk University/Jordan. His research interests lie in the area of algorithms and mathematical models, ranging from theory to implementation, with a focus on checking the correctness conditions of concurrent and reactive systems. In recent years, he has focused on theoretical computer science such as Graph theory and Numerical analysis. Dr. Alshorman research interests are: 1. Mathematical methods in computer science 2. Temporal logics 3. Concurrent systems 4.Serializability of Transactions 5.Numerical analysis.

Appendix A

```
-------------------------------------------------
MODULE client(qs1,qs2,st)
-------------------------------------------------
VAR
process_c:{p1,p2};
state: {idle, req, rej, wait, rec, comp};
-------------------------------------------------
ASSIGN
init(process_c) :={p1,p2};
init(state) := idle;
next(state) :=case
state=idle & process_c=p1 & qs1!=2: req;
state=req & process_c=p1 & qs1!=2: wait;
state=req & process_c=p1 & qs1=2: rej;
 --no space in socket 1
state=wait & process_c=p1 &st!=busy: rec;
state=rec : comp;
state=rej : req; --request again
state=comp :idle; --iterate infinitely often
state=idle & process_c=p2 & qs2!=2: req;
state=req & process_c=p2 & qs2!=2: wait;
state=req & process_c=p2 & qs2=2: rej; --no space in
socket 2
state=wait &process_c=p2 &st!=busy: rec;
TRUE : state;
esac;
-------------------------------------------------
next(process_c) :=case
process_c=p1 & state!=comp : p1;
process_c=p1 & state=comp : {p1,p2};
process_c=p2 & state!=comp : p2;
process_c=p2 & state=comp : {p1,p2};
 TRUE : process_c;
esac;
-------------------------------------------------
MODULE server (queuesoc1,queuesoc2)
-------------------------------------------------
VAR
state_s :{idle, pro, busy};
-------------------------------------------------
ASSIGN
init(state_s):=idle;
next(state_s) :=case
state_s=idle & queuesoc1 =-1 & queuesoc2 =-1 : idle;
state_s=idle & queuesoc1!=2 & queuesoc1!=2 : pro;
state_s=pro & queuesoc1=2 & queuesoc1=2 : busy;
TRUE: state_s;
esac;
-------------------------------------------------
MODULE Queue1 (st1,pr1,st2,pr2,st3,pr3)
-------------------------------------------------
VAR
queuesoc1 : -1..2;
-------------------------------------------------
ASSIGN

init (queuesoc1) := -1;

next(queuesoc1) :=case
queuesoc1=-1 & ((st1=req& pr1=p1)|(st2=req&
pr2=p1)|(st3=req& pr3=p1)) : 0;

queuesoc1=0 & ((st1=req& pr1=p1)|(st2=req&
pr2=p1)|(st3=req& pr3=p1)) : 1;

queuesoc1=1 & ((st1=req& pr1=p1)|(st2=req&
pr2=p1)|(st3=req& pr3=p1)) : 2;

queuesoc1=2 & ((st1=req& pr1=p1)|(st2=req&
pr2=p1)|(st3=req& pr3=p1)): queuesoc1;

queuesoc1=0 & ((st1=comp & pr1=p1)|(st2=comp &
pr2=p1)|(st3=comp & pr3=p1)) : -1;

queuesoc1=1 & ((st1=comp & pr1=p1)|(st2=comp &
pr2=p1)|(st3=comp & pr3=p1)) : 0;

queuesoc1=2 & ((st1=comp & pr1=p1)|(st2=comp &
pr2=p1)|(st3=comp & pr3=p1)) : 1;

queuesoc1=-1 & ((st1=comp & pr1=p1)|(st2=comp &
pr2=p1)|(st3=comp & pr3=p1)): queuesoc1;

TRUE: queuesoc1;
esac;
-------------------------------------------------
MODULE Queue2 (st1,pr1,st2,pr2,st3,pr3)
-------------------------------------------------
VAR
queuesoc2 : -1..2;
-------------------------------------------------
ASSIGN

init (queuesoc2):= -1;

next(queuesoc2) :=case

queuesoc2=-1 & ((st1=req& pr1=p2)|(st2=req&
pr2=p2)|(st3=req& pr3=p2)) : 0;

queuesoc2=0 & ((st1=req& pr1=p2)|(st2=req&
pr2=p2)|(st3=req& pr3=p2)) : 1;

queuesoc2=1 & ((st1=req& pr1=p2)|(st2=req&
pr2=p2)|(st3=req& pr3=p2)) : 2;

queuesoc2=2 & ((st1=req& pr1=p2)|(st2=req&
pr2=p2)|(st3=req& pr3=p2)) : queuesoc2;

queuesoc2=0 & ((st1=comp & pr1=p2)|(st2=comp &
pr2=p2)|(st3=comp & pr3=p2)) : -1;
queuesoc2=1 & ((st1=comp & pr1=p2)|(st2=comp &
pr2=p2)|(st3=comp & pr3=p2)) : 0;
```

```
queuesoc2=2 & ((st1=comp & pr1=p2)|(st2=comp &
pr2=p2)|(st3=comp & pr3=p2)) : 1;

queuesoc2=-1 & ((st1=comp & pr1=p2)|(st2=comp &
pr2=p2)|(st3=comp & pr3=p2)) : queuesoc2;

TRUE: queuesoc2;
esac;
-------------------------------------------------
MODULE main
-------------------------------------------------
VAR
-------------------------------------------------
s:server(qu1.queuesoc1,qu2.queuesoc2);
c1: client(qu1.queuesoc1,qu2.queuesoc2,s.state_s );
c2 : client(qu1.queuesoc1,qu2.queuesoc2,s.state_s );
c3 : client(qu1.queuesoc1,qu2.queuesoc2,s.state_s );
qu1:
Queue1(c1.state,c1.process_c,c2.state,c2.process_c,c3.
state,c3.process_c);
qu2:
Queue2(c1.state,c1.process_c,c2.state,c2.process_c,c3.
state,c3.process_c);
-------------------------------------------------------
-------------SPECIFICATIONS-------------------------
-------------------------------------------------------
--Condition number 1

SPEC AG(c1.state=req ->AF c1.state=comp)

SPEC AG(c2.state=req ->AF c2.state=comp)

SPEC AG(c3.state=req ->AF c3.state=comp)

LTLSPEC G(c1.state=req ->F c1.state=comp)

LTLSPEC G(c2.state=req ->F c2.state=comp)

LTLSPEC G(c3.state=req ->F c3.state=comp)

-- Condition number 5

SPEC AG AF!(s.state_s=busy)

--Condition number 6

SPEC AG((c1.state=rej& c1.process_c=p1) ->AF
c1.state=comp)

SPEC AG((c1.state=rej& c1.process_c=p2) ->AF
c1.state=comp)

SPEC AG((c2.state=rej& c2.process_c=p1) ->AF
c2.state=comp)
SPEC AG((c2.state=rej& c2.process_c=p2) ->AF
c2.state=comp)
SPEC AG((c3.state=rej& c3.process_c=p1) ->AF
c3.state=comp)
SPEC AG((c3.state=rej& c3.process_c=p2) ->AF
c3.state=comp)
-- Condition number 4

SPEC AG!(c1.state=rej& c1.process_c=p1 &
qu1.queuesoc1!=2)

SPEC AG!(c1.state=rej& c1.process_c=p2 &
qu2.queuesoc2!=2)

SPEC AG!(c2.state=rej& c2.process_c=p1 &
qu1.queuesoc1!=2)

SPEC AG!(c2.state=rej& c2.process_c=p2 &
qu2.queuesoc2!=2)

SPEC AG!(c3.state=rej& c3.process_c=p1 &
qu1.queuesoc1!=2)

SPEC AG!(c3.state=rej& c3.process_c=p2 &
qu2.queuesoc2!=2)

-- Condition number 2

SPEC AG ((c1.state=req -> AF c1.state=comp) &
(c1.state=comp -> AF c1.state=req))

SPEC AG ((c2.state=req-> AF c2.state=comp) &
(c2.state=comp -> AF c2.state=req))

SPEC AG ((c3.state=req-> AF c3.state=comp) &
(c3.state=comp -> AF c3.state=req))

--Condition number3

SPEC AG ((qu1.queuesoc1=-1 & (c1.state=req&
c1.process_c=p1))-> AX (qu1.queuesoc1=0))

SPEC AG ((qu1.queuesoc1=0 & (c1.state=req&
c1.process_c=p1))-> AX (qu1.queuesoc1=1))

SPEC AG ((qu1.queuesoc1=1 & (c1.state=req&
c1.process_c=p1))-> AX (qu1.queuesoc1=2))

SPEC AG ((qu2.queuesoc2=-1 & (c1.state=req&
c1.process_c=p2))-> AX (qu2.queuesoc2=0))

SPEC AG ((qu2.queuesoc2=0 & (c1.state=req&
c1.process_c=p2))-> AX (qu2.queuesoc2=1))

SPEC AG ((qu2.queuesoc2=1 & (c1.state=req&
c1.process_c=p2))-> AX (qu2.queuesoc2=2))

-- condition that is produced counterexample(false)

SPEC AG (qu1.queuesoc1=-1-> AX qu1.queuesoc1=
-1)
```