Using CSP to Detect Errors in the TFTP

Lou Chen

***Abstract*—Trivial File Transfer Protocol (TFTP) is a simple lockstep file transfer protocol. In this paper we use PAT, a model checker for CSP, to detect errors in the TFTP. We model the protocol and a very general intruder as CSP processes, and use the model checker to test whether the intruder can successfully attack the protocol. We discover many different attacks leading to breaches of security.**

***Index Terms*—Security protocols, PAT, CSP, TFTP, model checking.**

I. Introduction

In this paper we consider a protocol due to Karen R. Sollins[6]. The protocol concerns a Server-Client system. In order for allows a client to get a file from or put a file onto a remote server. One of its primary uses is in the early stages of nodes booting from a local area network. TFTP has been used for this application because it is very simple to implement. TFTP was first standardized in 1981 [1] and the current specification for the protocol can be found in RFC 1350 [2]. In March 1995 the TFTP Option Extension RFC 1782 [3] updated later in May 1998 by RFC 2347 [4], defined the option negotiation mechanism which establishes the framework for file transfer options to be negotiated prior to the transfer using a mechanism which is consistent with TFTP's original specification.

The protocol is subject to a number of attacks; indeed, since TFTP includes no login or access control mechanisms, care must be taken in the rights granted to a TFTP server process so as not to violate the security of the server hosts file system. TFTP is often installed with controls such that only files that have public read access are available via TFTP and writing files via TFTP is disallowed. In this paper we present many different attacks, which makes the server or client cannot work anymore.

Our approach is to use the process algebra CSP [5], and its model checker PAT [6]. We encode the protocol in CSP, and produce a CSP description of the most general intruder that can interact with the protocol. We then use PAT to detect a number of attacks upon the protocol (PAT searches the state space of the system until it either finds an attack or exhausts the state space; this search is automatic in the sense that it does not require user guidance once the system has been modeled in CSP). Some of the attacks allow the intruder imitate another agent in a fake session; other attacks allow the intruder to learn the TID being used in a session between two other agents, and so eavesdrop on that session.

In the next section we describe the TFTP. In Section 3 we describe how the protocol can be modeled in CSP, and in Section 4 we use PAT, the model checker for CSP, to discover that an intruder can attack the protocol in a number of ways, leading to breaches of security. In Section 5 we provide a method to prevent these attacks, which we store the username or password in the data option field.

Manuscript is received February 12, 2018; revised May 10, 2018.

Lou Chen is with Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China (e-mail: 51151500006@stu.ecnu.edu.cn).

II. The TFTP

We give brief introduction to TFTP in Appendix A. The TFTP concerns two players: a *Server* Host, and a *Client* Host. The TFTP protocol for establish a session involves the exchange of three messages; It is illustrated below in Fig. 1.

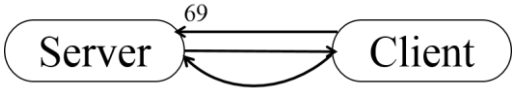

Fig. 1. Session.

TABLE I: Message

Message 1. (RRQ)	$Client \rightarrow Server:$ $RRQ.client.server.Port_c.Port_s.filename$
Message 2. (WRQ)	$Client \rightarrow Server:$ $WRQ.client.server.Port_c.Port_s.filename$
Message 3. (ACKc)	$Client \rightarrow Server:$ $ACKc.client.server.Port_c.Port_s.ndx$
Message 4. (ACKs)	$Server \rightarrow Client:$ $ACKs.server.client.Port_s.Port_c.ndx$
Message 5. (DATAc)	$Client \rightarrow Server:$ $DATAc.client.server.Port_c.Port_s.Data.ndx$
Message 6. (DATAs)	$Server \rightarrow Client:$ $DATAs.server.client.Port_s.Port_c.Data.ndx$
Message 7. (ERR)	$Server \rightarrow Client:$ $ERR.server.client.Port_s.Port_c.ErrType$

The message communicated between server and client can be defined as Table I. The acknowledgement message can be modelled into two messages, one is client send it to server and another is server it to client. We treat the data message like acknowledgement.

Any transfer begins with a request to read or write a file, which also serves to request a connection (Messages 1 or/and Message 2). If the server grants the request (Message 4), the connection is opened and the file is sent in fixed length blocks of 512 bytes (Message 5 or/and Message 6). Each data packet contains one block of data, and must be acknowledged by an acknowledgment packet before the next packet can be sent (Message 4 or/and Message 3). A

doi: 10.18178/jacn.2018.6.1.249

data packet of less than 512 bytes signals termination of a transfer. If a packet gets lost in the network, the intended recipient will timeout and may retransmit his last packet (which may be data or an acknowledgment), thus causing the sender of the lost packet to retransmit that lost packet. The sender has to keep just one packet on hand for retransmission, since the lock step acknowledgment guarantees that all older packets have been received. Notice that both machines involved in a transfer are considered senders and receivers. One sends data and receives acknowledgments, the other receives data and sends acknowledgements.

III. Modeling the Protocol in CSP

In this section we give a brief description of how we model the TFTP in CSP. We give a brief overview of CSP in Appendix B for the reader unfamiliar with the language, the syntax of CSP has evolved since [5]. We assume the existence of the sets $Client$ of clients, $Server$ of servers, $Port$ of ports, which are integer in $[1, 65535]$. We define different sorts of message, correspond to the steps of the protocol. Each massage includes a tag from the set $\{RRQ, WRQ, ACKc, ACKs, DATAc, DATAs, ERR\}$; subsequent fields depend upon which protocol step we are dealing with.

$$MSG1 \triangleq \{RRQ.client.server.Port_c.Port_s.filename \mid client \in Client, server \in Server, Port_c, Port_s \in Port\}$$

$$MSG2 \triangleq \{WRQ.client.server.Port_c.Port_s.filename \mid client \in Client, server \in Server, Port_c, Port_s \in Port\}$$

$$MSG3 \triangleq \{ACKc.client.server.Port_c.Port_s.ndx \mid client \in Client, server \in Server, Port_c, Port_s \in Port\}$$

$$MSG4 \triangleq \{ACKs.server.client.Port_s.Port_c.ndx \mid client \in Client, server \in Server, Port_c, Port_s \in Port\}$$

$$MSG5 \triangleq \{DATAc.client.server.Port_c.Port_s.Data.ndx \mid client \in Client, server \in Server, Port_c, Port_s \in Port\}$$

$$MSG6 \triangleq \{DATAs.server.client.Port_s.Port_c.Data.ndx \mid client \in Client, server \in Server, Port_c, Port_s \in Port\}$$

$$MSG7 \triangleq \{ERR.server.client.Port_s.Port_c.ErrType \mid client \in Client, server \in Server, Port_c, Port_s \in Port\}$$

$$MSG \triangleq \bigcup_{i=1..7} MSGi$$

We use three channels to model the communications in system:

- The channel $session$ will represent standard communications between two honest hosts.
- The channel $server\_fake\_session$ will represent the server taking part in fake session, where the intruder impersonates the client.
- The channel $client\_fake\_session$ will represent the client taking part in fake session, where the intruder impersonates the server.
- The channel $leak\_sission$ will represent the sessions might be overheard by the intruder.

We declare these channels:

$$channel \quad session, server\_fake\_session, client\_fake\_session, leak\_session : MSG$$

These channels are illustrated in Fig. 2.

The protocol should prevent these latter events from happening, but as we shall see, it fails in this respect.

We now produce a CSP process representing each of the hosts in protocol. First, we consider a client ignoring for the moment the possibility of interference from the intruder, the client is represented by the process $Client$ below. The client first request to write or read a file, and send an appropriate Message 1 or Message 2. He then waits for a corresponding Messages 4 or Message 3; he obtain the $Port_s$ by decompose the message, and carries out sessions using the $Port_s$. A data packet of less than 512 bytes signals the termination of a transfer.

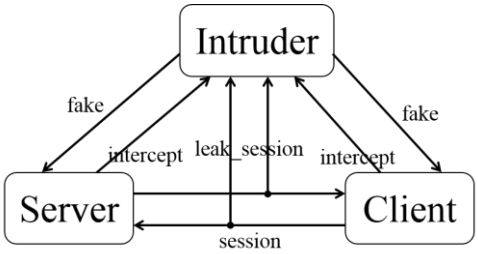

Fig. 2. Channel.

A file will be divided into several data packets depend on the size of it. The data packet will transfer iteratively. When the client send a read request to server to read a file, the server will reply data packet to client. We modelling the behaviors of transfer data packets iteratively.

$ClientIterRecv$ represents the client receive the data packet and reply acknowledgement to server iteratively. It will check the size of the Data, if the size less than 512 byte, which stand for the last packet, the $ClientIterRecv$ behavior stop. Or it will receive the data packet then reply acknowledgement to server, then it turn to $ClientIterRecv$.

Algorithm 1 ClientIterRecv

ClientIterRecv =

If size(Data)<512 **then**
Skip
Else
$session!$
$DATAs.server.client.Port_s.Port_c.Data.ndx \rightarrow$
$session?$
$ACKc.client.server.Port_c.Port_s.ndx \rightarrow$
$ClientIterRecv$
End if

$ServerIterSend$ represents the server send the data packet and receive acknowledgement from client iteratively. It will check the size of the Data, if the size less than 512 byte, which stand for the last packet, the $ServerIterSend$ behavior stop. Or it will send the data packet then reply acknowledgement to server, then it turn to $ServerIterSend$. The model of $ServerIterSend$ is similar to $ClientIterRecv$.

When the client send a write request to server to put a file, the server will reply an acknowledgement to client, and then they will transfer data packet. We modelling the behaviors of transfer data packets iteratively.

$ClientIterSend$ represents the client send the data packet and receive acknowledgement from server iteratively. It will check the size of the Data, if the size less than 512 byte, which stand for the last packet, the $ClientIterSend$ behavior stop. Or it will send the data packet and then receive acknowledgement from server, and then it turn to $ClientIterSend$.

Algorithm 2 ClientIterSend

ClientIterSend =

If size(Data)<512 **then**
Skip
Else
$session?$
$DATAc.client.server.Port_c.Port_s.Data.ndx \rightarrow$
$session!$
$ACKs.server.client.Port_s.Port_c.ndx \rightarrow$
$ClientIterSend$
End if

$ServerIterRecv$ represents the server receive the data packet and send acknowledgement to client iteratively. It will check the size of the Data, if the size less than 512 byte, which stand for the last packet, the $ServerIterRecv$ behavior stop. Or it will receive the data packet and then reply acknowledgement to client, and then it turn to $ServerIterRecv$. The model of $ServerIterRecv$ is similar to $ClientIterSend$.

The client contains two behaviors, one is send read request and then get the file he need, another is send write request and then send file to server. We modelling the behaviors of Client.

$$Client \triangleq$$
$$\left(\begin{array}{l} session?RRQ.client.server.Port_c.Port_s.filename \rightarrow \\ if\ (Error)\{Stop\}else\{ClientIterRecv\} \rightarrow \\ Client \end{array}\right)$$
$$\square$$
$$\left(\begin{array}{l} session?WRQ.client.server.Port_c.Port_s.filename \rightarrow \\ if\ (Error)\{Stop\}else\{ \\ session!ACKs.server.client.Port_s.Port_c.ndx \rightarrow \\ ClientIterSend\} \rightarrow \\ Client \end{array}\right)$$

The Server contains two behaviors, one is receive read request and then send the file out, the other is receive write request and then receive file from client. We modelling the behaviors of Server.

$$Server \triangleq$$
$$\left(\begin{array}{l} session!RRQ.client.server.Port_c.Port_s.filename \rightarrow \\ if\ (Error)\{Stop\}else\{ServerIterSend\} \rightarrow \\ Server \end{array}\right)$$
$$\square$$
$$\left(\begin{array}{l} session!WRQ.client.server.Port_c.Port_s.filename \rightarrow \\ if\ (Error)\{Stop\}else\{ \\ session?ACKs.server.client.Port_s.Port_c.ndx \rightarrow \\ ServerIterRecv\} \rightarrow \\ Server \end{array}\right)$$

We allow the possibility of intruder action: we must allow instances of Message to be intercepted, instances of Message to be faked, and sessions to be either faked or overheard. We do this via a renaming:

$$Client \triangleq Client[session \leftarrow session,$$
$$session \leftarrow client\_fake\_session,$$
$$session \leftarrow leak\_session]$$
$$Server \triangleq Server[session \leftarrow session,$$
$$session \leftarrow server\_fake\_session,$$
$$session \leftarrow leak\_session]$$

IV. Attacks Upon the Protocol

We will analyze the security of the protocol by putting it in parallel with an intruder. We want to model the intruder as a process that can perform any attack that we would expect a real-world intruder to be able to perform. Thus our model will allow the intruder to send message, or replay message. More precisely, we model an intruder who can:

- Overhear messages so as to learn the contents, possibly intercepting these messages;
- Drive new messages from ones he already knows;
- Fake new messages using messages he knows;
- Fake new messages onto $Server$ or $Client$.

We consider an intruder who initially knows TID of $Server$. We assume that the intruder is a user of the protocol in his own right, so can use the protocol to establish sessions with other clients, and other clients might try to establish sessions with him. Now we consider the CSP model of intruder. We begin by defining the set of facts that the intruder might learn; this consists of the atomic datatypes,

$$Facts \triangleq Server \cup Client \cup Port$$

We now define the submessages of a message. This are

the fact that an intruder will learn by seeing a message (without doing any further deductions); they are also the facts that the intruder needs to know in order to send a fake message (this definition could be simplified by assuming that the intruder always knows all the hosts' identities):

$$submessages(RRQ.client.server.Port_c.Port_s.filename) \triangleq \{client, server, Port_c, Port_s, filename\}$$
$$submessages(WRQ.client.server.Port_c.Port_s.filename) \triangleq \{client, server, Port_c, Port_s, filename\}$$
$$submessages(ACKc.client.server.Port_c.Port_s.ndx) \triangleq \{client, server, Port_c, Port_s, ndx\}$$
$$submessages(ACKs.server.client.Port_s.Port_c.ndx) \triangleq \{server, client, Port_s, Port_c, ndx\}$$
$$submessages(DATAc.client.server.Port_c.Port_s.Data.ndx) \triangleq \{client, server, Port_c, Port_s, Data, ndx\}$$
$$submessages(DATAs.server.client.Port_s.Port_c.Data.ndx) \triangleq \{server, client, Port_s, Port_c, Data, ndx\}$$
$$submessages(ERR.server.client.Port_s.Port_c.ErrType) \triangleq \{server, client, Port_s, Port_c, ErrType\}$$

We declare a channel $fake$ to represent messages introduced by the intruder: the receiver of these messages should not be aware that they are fakes; we declare a channel $intercept$ to represent messages sent by an honest agent that are intercepted by the intruder: the sender should not be aware that the message was intercepted.

$$channel \quad fake, intercept : MSG$$

We declare a channel $deduce$, which will be used for deducing new facts:

$$channel \quad deduce : Facts$$

The definition of the intruder is parameterized by the set of fact that he knows. The intruder can overhear or intercept a message so as to learn all its submessages; he can fake a message when he knows all the submessages; he can deduce a new fact from ones he already knows.

$$Intruder(S) \triangleq$$
$$\Box_{m \in MSG} leak\_session.m \rightarrow Intruder(S \cup submessages(m))$$
$$\Box$$
$$\Box_{m \in MSG} intercept.m \rightarrow Intruder(S \cup submessages(m))$$
$$\Box$$
$$\Box_{m \in MSG} session.m \rightarrow Intruder(S \cup submessages(m))$$
$$\Box$$
$$\Box_{m \in MSG, submessages(m) \subseteq S} fake.m \rightarrow Intruder(S)$$
$$\Box$$
$$\Box_{f \in Facts, f \notin S} deduce.f \rightarrow Intruder(S \cup \{f\})$$

We now consider a system with an intruder. First we form the system without the intruder:

$$SYSTEM \triangleq Client() [| INTRUDER\_CONT |] Server()$$

where:

$$INTRUDER\_CONT \triangleq \{session, fake, intercept, client\_fake\_session, server\_fake\_session, leak\_session\}$$

Note that in the above definition:

- $session$ events are shared between the client and server, so the intruder takes no part in these events;
- $client\_fake\_session$ events are shared between the client and intruder, so the server takes no part in these events;
- $server\_fake\_session$ events are shared between the server and intruder, so the client takes no part in these events;
- $leak\_session$ events are shared by all three agents, so the intruder over hears the session between the client and the server.

We want to know whether the intruder can ever spy upon sessions or cause fake sessions to be set up between client and server; i.e., we want to know whether the system with an intruder will ever perform $leak\_session$, $client\_fake\_session$, $server\_fake\_session$. Thus we will test our system against the specifications:

$$SPEC_l \triangleq CHAOS(\Sigma - \{leak\_session\})$$
$$SPEC_c \triangleq CHAOS(\Sigma - \{client\_fake\_session\})$$
$$SPEC_s \triangleq CHAOS(\Sigma - \{server\_fake\_session\})$$

If the system with the intruder refined these specifications, then it would indeed be secure. However, PAT can be used to discover that $SYSTEM$ does not refine any of the above specifications; It discovers the following attacks upon the protocol.

Attack 4.1. $SYSTEM$ does not refine $SPEC_s$. It can perform the trace:

$$\langle fake.Msg1, server\_fake\_session.Msg6, fake.Msg5 \rangle$$

We can rewrite the attack in more conventional style; we write, for example, I_c to represent the intruder I imitating $client$.

Step 1. $I_c \rightarrow S : fake.Msg1$

Step 2. $S \rightarrow I_c : server\_fake\_session.Msg6$

Step 3. $I_c \rightarrow S : fake.Msg3$

The intruder sends a RRQ to server (Step 1), pretending to be client, to read a file. No attempt is made to authenticate the identity of the client. The server send the data of file to intruder with TID of intruder and server (Step 2), and then the intruder send acknowledgement to server (Step 3). The intruder get the file of the server after the session above.

Attack 4.2. $SYSTEM$ does not refine $SPEC_c$. It can

perform the trace:

$$\langle client\_fake\_session.Msg1,\ fake.Msg6,$$
$$client\_fake\_session.Msg3\rangle$$

We can rewrite the attack as follow:

Step 1. $C \to I_s : client\_fake\_session.Msg1$

Step 2. $I_s \to C : fake.Msg6$

Step 3. $C \to I_s : client\_fake\_session.Msg3$

The client sends a RRQ to server which is pretended by intruder (Step 1), to read a file. The client was not aware the message was intercept, the intruder deduce the message received from client and fake the message to client. The client get the file that was not supposed to be after the session above.

Attack 4.3. $SYSTEM$ does not refine $SPEC_l$. It can perform the trace:

$$\langle leak\_session.Msg1, leak\_session.Msg6,$$
$$leak\_session.Msg3\rangle$$

We can rewrite the attack as follows:

Step 1. $C \to S : leak\_session.Msg1$

Step 2. $S \to C : leak\_session.Msg6$

Step 3. $C \to S : leak\_session.Msg3$

The intruder can get the message between client and server, then it get the submessage, such as the TID of server and client and the port they used. The intruder than overhear the communications between server and client.

The reason these attacks succeeded was that the messages are not authenticated, so the intruder can get or send messages without any identification.

V. Conclusion

In this paper we have shown how to model the TFTP in CSP, and how the session can be attacked by the intruder which have been described attacks on the protocol.

In order to overcome the problems discovered above, we think the TFTP protocol allows additional data options at the end of RRQ and WRQ packets. These data options are mainly used to negotiate the size of the data block and wait for time and other information. Therefore username or password can be stored in this data option field.

When we take usernames and passwords stores in the data segment, the security of protocol can be improved. First of all, before the establishment of communication, we can design the algorithm that was related to time the clients' TID and servers' TID, which requires both ends of the moment with an Internet time synchronous. During the communication, client and server can obtain the submessages of the message, and both of them calculate a determined value with the algorithm to compare with a preset value. Therefore it prevent intruder counterfeit the server and the client, at the same time, and provides a different configuration of algorithms to encrypt the real data, so that can prevent intruder monitoring the session.

Appendix A

In this appendix we give a brief introduction to TFTP, We present the order of Headers in TABLE II. The order of the contents of a packet will be: local medium header, if used, Internet header, Datagram header, TFTP header, followed by the remainder of the TFTP packet.

TABLE I: Order of Headers

			2 bytes
Local Medium	Internet	Datagram	TFTP Opcode

In Table III we present TFTP Formats [7], TFTP support five type of packets, all of which have been mentioned above, RRQ and WRQ have the format shown in below. The file name is a sequence of bytes in hetscii terminated by a zero byte. The mode field contains the string “netascii”, “octet”, or “mail”(or any combination of upper and lower case, such as “NETASCII”, “NetAscii”, etc.) in netascii indicating the three modes defined in the protocol.

Data is actually transferred in DATA packets depicted below. DATA packets have a block number and data field. The block numbers on data packets begin with one and increase by one for each new block of data.

TABLE II: TFTP Formats

Type	Op #	Format without header			
	2 bytes	String	1	string	1 byte
RRQ	01	Filename	0	Mode	0
	2 bytes	String	1	String	1 byte
WRQ	02	Filename	0	Mode	0
	2 bytes	2 bytes	n bytes		
DATA	03	Block #	Data		
	2 bytes	2 bytes			
ACK	04	Block #			
	2 bytes	2 bytes	string	1 byte	
ERROR	05	ErrorCode	ErrMsg	0	

TABLE III: Error Codes

Value	Meaning
0	Not defined, see error message (if any).
1	File not found.
2	Access violation.
3	Disk full or allocation exceeded.
4	Illegal TFTP operation.
5	Unknown transfer ID.
6	File already exists.
7	No such user.

We show the error codes of TFTP in TABLE IV, the error code is an integer indicating the nature of the error. The error message is intended for human consumption, and should be in netascii.

Appendix B

In this section we give a brief overview of CSP. More details can be obtained from [5], [7].

An event represents an atomic communication; this might either be between two processes or between a process and the environment. Channels carry sets of events; for example, $client\_fake\_session.Msg1$ is an event of channel $client\_fake\_session$. Σ represents the set of all events. The notation $\{|a,b|\}$ represents the set of all events over channels a and b.

In this paper we use processes defined using the following syntax:

TABLE IV：CSP SYNTAX

$STOP$	process that can perform no events.
$a \to P$	process that can perform the event a, and then act like P.
$P \Box Q$	external choice; the process can act like either P or Q; the choice is made by the environment.
$P[[a \leftarrow b]]$	process that acts like P, except the event a is renamed b; this operator can also be used for multiple renamings, and has a comprehension form.
$P \setminus A$	process that acts like P, except all events from the set A are hidden, i.e., made internal.
$CHAOS(A)$	the most nondeterministic, nondivergent process with alphabet A; the process can perform any sequence of events from A.
$P[\|A\|]Q$	parallel composition of P and Q, synchronizing on events from A.

The trace model of CSP represents a process by the set of traces it can perform, where a trace is a sequence of events. We say that process P is (trace) refined by process Q if the traces of P are a superset of the traces of Q. PAT can be used for testing refinement between two finite state processes.

PAT is a tool based on CSP and designed to apply model checking techniques for system analysis. It is self-contained framework for simulating and reasoning of concurrent, real-time systems [8] and other domains. Above all, PAT implements various model checking techniques catering for different properties such as deadlock-freeness, reachability, LTL properties with fairness assumption in distributed systems [9]. Here we list some notations.

- $\#define\ V\ 0$

  It defines a global constant V with the initial value 0.
- $var\ Dstate[V] = [0,1,2,\cdots,V-1]$

  This statement defines an array named $Dstate$. The size of the array is V, which is a global constant. And the initial value of the array is specified as the sequence $[0,1,2,\cdots,V-1]$.
- $channel\ c\ 1$

  This statement declares a channel, c is the channel name and 1 is the buffer size. Channel buffer size must be greater than or equal to 0. Notice that a channel with buffer size 0 sends/receives messages synchronously.
- $P = \{v = v+1\} \to Skip$

  It denotes a global variable can be updated by an action.

$\#define\ goal\ v > 0; \#assert\ P\ reaches\ goal;$ It defines an assertion to check whether process P can reach a state which a condition goal is satisfied or not.

REFERENCES

[1] K. R. Sollins, The TFTP Protocol[J]. Rfc, 1981.
[2] K. Sollins, The TFTP Protocol (Revision 2)[M]. RFC Editor, 1992.
[3] A. Harkin, TFTP Option Extension[J]. Information on Rfc, 1995.
[4] A. Harkin, TFTP Option Extension[J]. Information on Rfc, 1998.
[5] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the Acm*, vol. 21, no. 1, pp. 666-677, 1985.
[6] J. Sun, Y. Liu, J. S. Dong, *Model Checking CSP Revisited: Introducing a Process Analysis Toolkit[M]// Leveraging Applications of Formal Methods, Verification and Validation*, Berlin Heidelberg: Springer, 2008.
[7] A. W. Roscoe, "The theory and practice of concurrency," *Prentice Hall PTR*, 1997.
[8] J. Sun, Y. Liu, and J. S. Dong, "Modeling and verifying hierarchical real-time systems using stateful timed CSP[J]," *Acm Transactions on Software Engineering & Methodology*, vol. 22, no. 1, pp. 1-29, 2013.
[9] Y. Si, J. Sun, and Y. Liu, "Model checking with fairness assumptions using PAT[J]," *Frontiers of Computer Science*, vol. 8, no. 1, pp. 1-16, 2014.

Lou Chen was born on Jan. 7th in Anhui province, China. He obtained a bachelor's degree in mathematics and applied mathematics in Hefei, Anhui province in 2011. He is currently studying for master's degree at East China Normal University, his major is software engineering.