Formal Modeling and Verification of Time-Constrained ARQ Protocols with Event-B

Rajaa Filali[#1], Mohamed Bouhdadi[#2]
[#] LMPHE Laboratory University of Mohammed V, Faculty of sciences
4 Street Ibn Batouta, PB 1014 RP, Rabat, MOROCCO
[1] rajaafilali@gmail.com
[2] bouhdadi@fsr.ac.ma

***Abstract*—Automatic Repeat Request (ARQ) is a control error mechanism based on the retransmissions of lost packet. This mechanism is adequate for an important number of communication protocols where reliability is of prime importance. Formal methods are indispensable for the development of these protocols in order to ensure their correctness. In this paper, we study the practical aspects of applying Event-B and UPPAAL for modeling and verification of time-constrained ARQ protocols. We start by introducing a pattern for the retransmission time-out within Event-B and transform this pattern to pattern in UPPAAL. We have used UPPAAL to augmenting Event-B modeling with real-time verification, since the modeling of timing properties is not directly supported in Event-B. At last, we illustrate our approach with a case study based on the stop-and-wait protocol.**

Keywords - Event-B, UPPAAL, Verification, Time-constraints, Formal modeling, ARQ protocols.

I. Introduction

Automatic-repeat-request (ARQ) protocols [1] are widely used in modern data communications to guarantee reliable transmission between a sender and a receiver over unreliable communication channels. ARQ uses the principle of retransmission upon timeout to recover data considered as lost or damaged. Several formal methods have been applied to ensure the correctness of these protocols, such as Petri Nets [2] and theorem proving [3]. As a formal methods based on the first order classical logic and set theory, Event-B [4] [5] has the advantages in mechanized proving and the possibility to model a system in several levels of abstraction through refinement [6].

In this paper we present an approach to modeling and verification of time-constrained ARQ protocols. This approach consists in exploiting the pattern for modeling of retransmission time-out within Event-B and transforms this pattern to pattern in UPPAAL in order to augmenting Event-B models with time. While Event-B offers a scalable approach to ensuring functional correctness of a system, it is not efficient for verification of timing constraint (non functional properties). UPPAAL [7], on the other hand, is a model checker which has a good support for timing. The use of Event-B with UPPAAL can guarantee both functional and nonfunctional (timed) [8] properties.

We have used the Rodin platform [9] [10], which provides an environment for development, analysis and verification of Event-B models. The ProB [11] was very useful in animating all models and in verifying the absence of error (no counter-examples exist) and deadlock. The model checking tool UPPAAL is called upon to verify the timed automata [12] representing the model.

We illustrate our approach by formal development and verification of the stop-and-wait ARQ protocol [13] which is a basic Automatic Repeat Request (ARQ) protocol that ensures reliable data transfers across noisy channels.

The structure of the paper is as follows. Section 2 gives a short introduction to Event-B, the Rodin platform and UPPAAL. The retransmission time-out pattern is presented in Section 3. Section 4 gives a case study to illustrate the motivation for our approach. Finally, a conclusion is presented to summarize the main outcomes of this research.

II. Overview of Event-B Method and UPPAAL

A. *Event-B and Rodin tool*

Event-B is a modeling method used to formalize and develop transition systems. It is an evolution of the (classical) B-method [14]. Event-B is centered on the notion of events (transitions). It is based on first-order logic [15] and a typed set-theory [16]. The models described with Event-B are built by means of two basic constructs: contexts and machines. Contexts contain the static parts of a model whereas machines contain its dynamic parts. Machines and contexts can be inter-related: a machine can be refined by another one, a context can be extended by another one and a machine can see one or several contexts as shown in Fig.1.

Contexts specify the static part of a model. They may contain carrier sets (similar to types), constants, axioms (containing carrier sets and constants), and theorems (expressing properties derivable from axioms). -Machines specify behavioral properties of the models. They may contain variables defining the state of a machine, invariants constraining that state, and events (describing possible state changes). Each event is composed of a set of guards and a set of actions. Guard state the necessary conditions under which an event may occur, and actions describe how the state variables evolve when the event occurs. Contexts/Machines may be refined from more abstract to more concrete contexts/machines. A key concept in Event-B is proof-obligation (PO) [17] capturing the necessity to prove some internal property of the model such as typing, invariant preservation by events, and correct refinements. Strong tool support is provided in order to support this proof process.

Event-B is not specific to embedded systems design but it is currently being investigated by several industrial from different sectors (automotive, transportation, space) in the context of the DEPLOY project . In Event-B, an event is defined by the syntax: EVENT e WHEN G THEN S END , Where G is the guard, expressed as a first-order logical formula in the state variables, and S is any number of generalized substitutions, defined by the syntax S ::= x := E(v) | x := z : | P(z). The deterministic substitution, x := E(v), assigns to variable x the value of expression E(v), defined over set of state variables v. In a non-deterministic substitution, x := z : | P(z), it is possible to choose non deterministically local variables, z, that will render the predicate P(z) true. If this is the case, then the substitution, x := z, can be applied, otherwise nothing happens. It is also important to indicate that the most important feature provided by Event-B is its ability to stepwise refine specifications. Refinement is a process that transforms an abstract and non-deterministic specification into a concrete and deterministic system that preserves the functionality of the original specification. During the refinement, event descriptions are rewritten to take new variables into account. This is performed by strengthening their guards and adding substitutions on the new variables. New events that only assign the new variables may also be introduced. Proof obligations (POs) are generated to ensure the correctness of the refinement with respect to the abstract model. Event-B is supported by several tools, currently in the form a platform called Rodin. Rodin is an open-source development platform for Event-B. It provides an environment for system modeling and analyses, including support for refinement, i.e. POs are generated automatically between abstraction levels, and support for mathematical proof, i.e. most POs can be discharged automatically or manually.

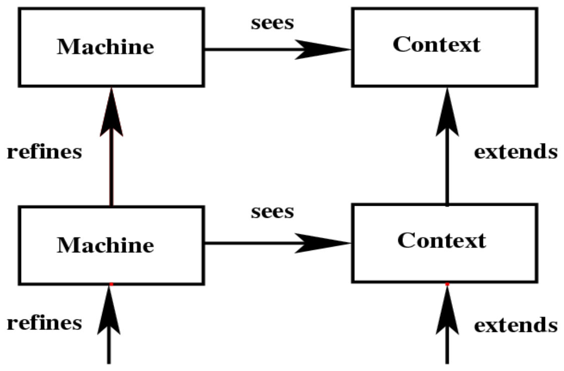

Fig.1. Machine and Context relationship

B. UPPAAL

UPPAAL is a model checker that allows definition of the system behavior by means of a network of timed automata. A timed automaton is a finite state machine extended with clock variables, which are used to measure time progress. The theory of timed automata allows clocks to be evaluated to a real number, but UPPAAL restricts those evaluations to integer numbers. An UPPAAL timed automaton is made up of a number of locations and a number of transitions (also known as edges) between them. A transition usually causes an update of the system's variables. In particular, as a consequence of a transition, integer and boolean variables can be assigned a new value, whereas clock variables can only be restarted. Synchronous actions (i.e. actions that are simultaneously performed by several automata) can be modeled by means of synchronous channels. Such transitions lead to a new state of the system in which the involved automata may all have stepped into a new location. A transition (either synchronous or not) is enabled as long as its corresponding guard condition holds. Guard conditions can use integer, boolean and clock variables, yet with some restrictions. There is a particular kind of locations, the so-called committed locations, which help to model atomic actions. These locations are always left immediately and can interleave only with other commited locations. Additionally, UPPAAL incorporates three mechanisms to model urgency, namely invariants, urgent channels and urgent locations, making it possible to force certain transitions to be fired as soon as they are enabled. Whenever these mechanisms are not used, a transition may take place at an undetermined instant. A fundamental characteristic of a network of timed automata is that clocks progress at the same rate. Nevertheless, in this work some techniques have been developed which allow us to model clocks of different rates as well.

A system in UPPAAL is modeled as a network of timed automata. A subset of CTL (computation tree logic) [18] is the basis for the query language in UPPAAL. The following three kinds of properties can be checked with UPPAAL.

- **Reachability** i.e. some condition can possibly be satisfied.
- **Safety** i.e. some condition will never occur.
- **Liveliness** i.e. some condition will eventually become true.

III. Retransmission time-out pattern

In this section, we will introduce the pattern of the retransmission over time-out which is modeled firstly in Event-B and then transferred to UPPAAL in order to augmenting event-B models with time.

A. Modeling retransmission time-out in Event-B

In the abstract model, the event *Send* sets the Boolean variable S as one of its actions, so when variable S has the value of TRUE, it shows event *Send* has happened. Also, in event *Confirm* the flag of event *Send* will be checked to see if event Send has already happened, and the variable C will be set to TRUE. SendGrds and ConfirmGrds represent the other possible guards of the event and in the action section. SendActs and ConfirmActs represent the other possible actions of the event.

```
EVENT  Send              EVENT Confirm
 WHERE                    WHERE
 S = FALSE                S = TRUE
 SendGrds                 C = FALSE
 THEN                     ConfirmGrds
 S := TRUE                THEN
 SendActs                 C:=TRUE
 END                      ConfirmActs
                          END
```

In the refinement, two variables t and rcr are declared to represent the current time and the retransmission counter in the machine.

If the event Confirm has not happened when the retransmission timer expires, the event Resend will happen, the retransmission counter (rcr) is incremented by one and the retransmission timer restarted. For the events Send and confirm, the refinement is just a superposition, time constraints are added without changing the existing expressions. If the transmission is successful, the propagation time should be shorter than time. The event tick-tock is added to model the progress of time.

```
EVENT Send        EVENT Confirm       EVENT Resend       EVENT Tick-Tock
 WHERE            WHERE                WHERE             THEN
 S = FALSE        S = TRUE             S = TRUE          t:=t+1
 SendGrds         C = FALSE            t=time            END
 THEN             t<=time              rcr<RCR_MAX
 S := TRUE        ConfirmGrds          reSendGrds
 rcr:=1           THEN                THEN
 SendActs         C:=TRUE             t:=0
END               t:=0                rcr:=rcr+1
                  ConfirmActs         reSendActs
                  END                  END
```

B. Verification of retransmission time-out with UPPAAL

At first, Event-B model is transformed to UPPAAL model as shown in Figure 2. Event-B events are mapped into UPPAAL transitions and abstract clock, event-B guards are mapped into UPPAAL invariants and guards, the invariants and axioms in event-B are modeled to declarations in UPPAAL, and the event-b states are mapped to locations in UPPAAL.

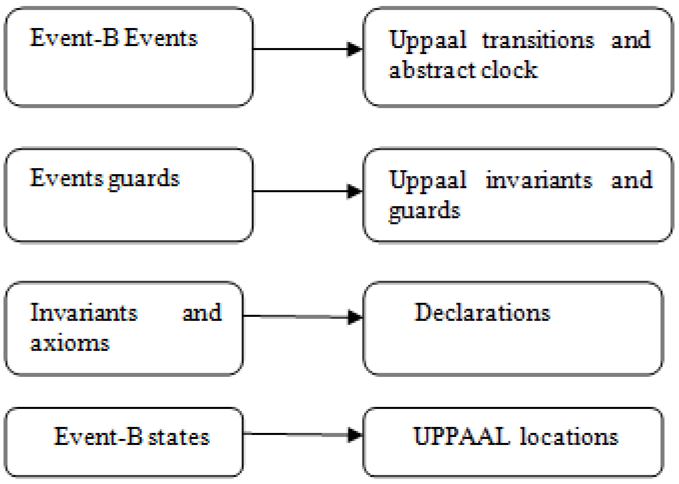

Fig. 2. Event-B to UPPAAL transformation

By applying this mapping on our event-B model, we get the UPPAAL model of retransmission time-out, as is shown in Fig. 3.

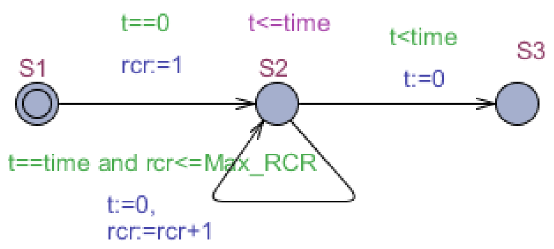

Fig. 3. UPPAAL model of retransmission time-out

The pattern model in UPPAAL is verified using some properties. We discuss each of the properties with the UPPAAL query language representation.

retransmission.S2 --> retransmission.S3 imply t<=time: if the state S2 of the automaton retransmission is reached, it will result in reaching the state S3 just when t<=time.

A<> t>time imply not retransmission.S2: for any t>time, the state S2 cannot be reached.

E[] retransmission.S2 imply ! retransmission.S3: the automaton retransmission may be still in the state S2 until the state S3 will be activated.

IV. Case Study

In this section, we will use Event-B to model the real-time properties of Stop-and-wait Protocol and then applied our approach to verify its correctness.

A. *Informal specification of stop-and-wait protocol*

A Stop-and-Wait Protocol can be considered to be any data transfer protocol in which the sending entity stops after transmitting a message and waits until it receives an acknowledgement indicating that the receiver is ready to receive the next message.

The stop-and-wait is a basic Automatic Repeat Request (ARQ) protocol that ensures reliable data transfers across noisy channels and combine flow control with error recovery using a timeout and retransmission. When a message is sent, a timer is started, which will expire after some finite timeout period.

We consider that the sender and receiver entities can each be in one of two states. For the sender, this is one state in which the sender is ready to send a new message, and another in which the sender is waiting for an acknowledgement of the currently outstanding message. For the receiver, this is one state in which it is ready to receive a message, and another in which it is processing a message and generating the appropriate acknowledgement with which to reply. Both the sender and receiver will alternate between their two respective states as protocol execution proceeds.

Both the sender and the receiver maintain a sequence number. In the case of the sender, this sequence number (the sender sequence number) records the sequence number of the message to send next, or if a message is currently outstanding, the sequence number of the message that is currently outstanding. In the case of the receiver, this sequence number (the receiver sequence number) records the sequence number of the next message expected by the receiver.

B. Formal modeling of SAW protocol in Event-B

We have made our model by several refinements, but we will just present the final model because of lack of space.

Sets, constants and axioms: We first introduce the basic sets we will use in our model: a set “Messages” of possible messages among the sender, a set “ACK” of acknowledgment which can be sent by the receiver and the two sets “state_send” and “state_rec” which represent the possible states of the sender and the receiver, respectively.

We define also the two constants that we assigned default values: the constant “time” represents the timeout period and the constant “RCR_MAX” which represents the maximum number of retransmission.

SETS

Messages

ACK

state_send

state_rec

CONSTANTS

time

RCR_MAX

s_ready

wait_ack

r_ready

processing

AXIOMS

Axm1	:	partition(state_send, {s_ready}, {wait_ack})
Axm2	:	partition(state_rec, {r_ready}, {processing})

Variables and invariants: In order to manipulate states we introduce two new variables:

st_snd: denotes the current state of sender.

st_rcv: denotes the current state of receiver.

Then the variables Packet_msg_snd, Packet_msg_rcv, Packet_ack_snd, and Packet_ack_rcv are introduced to define respectively, the set of messages sent by the sender, the messages received successfully by the receiver, the set of acknowledges sent by the receiver and successfully received acknowledges by the sender.

The variables t and rcr denote the current time and the number of retransmission respectively.

sn and rn represent respectively the current sender sequence number and the current receiver sequence number.

oldSn and oldRn represent respectively the old sender sequence number and the old receiver sequence number.

VARIABLES

st_snd

st_rcv

t

rcr

sn

rn

r

oldSn

oldRn

Packet_msg_snd

Packet_msg_rcv

Packet_ack_snd

Packet_ack_rcv

INVARIANTS

inv1	:	$st\_snd \in state\_send$
inv2	:	$st\_rcv \in state\_rec$
inv3	:	$t \in \mathbb{N}$
inv4	:	$rcr \in \mathbb{N}$
inv5	:	$sn \in \mathbb{N}$
inv6	:	$rn \in \mathbb{N}$
inv7	:	$Packet\_msg\_rcv \in \mathbb{P}(Messages \times \mathbb{N})$
inv8	:	$Packet\_msg\_snd \in \mathbb{P}(Messages \times \mathbb{N})$
inv9	:	$Packet\_ack\_snd \in \mathbb{P}(ACK \times \mathbb{N})$
inv10	:	$Packet\_ack\_rcv \in \mathbb{P}(ACK \times \mathbb{N})$
inv11	:	$r \in BOOL$
inv12	:	$oldSn \in \mathbb{N}$
inv13	:	$oldRn \in \mathbb{N}$

Events: Events of the model are enumerated as follows:

S_send_req : the sender moves from 's_ready' to 'wait_ack' state and sends the message with the sender sequence number sn.

R_receive_req : the receiver receives the message sent by the sender and moves to processing state. The receiver increments his sequence number by one.

R_send_ack : after receiving the request, the receiver sends an acknowledgment to the sender..

S_receive_ack : represents the acknowledgment received successfully by the sender.

Re_send_req : When timer t fires, the sender must retransmit the request and must reset the timer with a value 0 and increments the retransmission counter rcr by one.

receive_old_req: when the receiver receives an old message, it remains in r_ready state.

re_send_ack : represents the resending acknowledgment from the receiver to the sender.

receive_old_ack : represents the duplicate acknowledgment received by the sender.

tick_tock : represents the time progressing event.

S_send_req ≙

ANY

msg

WHERE

grd1 : st_snd=s_ready

grd2 : t=0

grd3 : (msg↦sn) ∉ Packet_msg_snd

THEN

act1 : st_snd≔wait_ack

act2 : rcr≔1

act3 : Packet_msg_snd≔Packet_msg_snd ∪ {msg↦sn}

END

R_receive_req ≙

ANY

msg

WHERE

grd1 : st_rcv=r_ready

grd2 : sn=rn

grd3 : (msg↦sn) ∈ Packet_msg_snd

THEN

act1 : st_rcv≔processing

act2 : rn≔rn+1

act3 : Packet_msg_rcv≔Packet_msg_rcv ∪ {msg↦sn}

act4 : Packet_msg_snd≔Packet_msg_snd \ {msg↦sn}

END

R_send_ack ≙

ANY

msg

ack

WHERE

grd1 : st_rcv=processing

grd2 : (msg↦sn) ∈ Packet_msg_rcv

grd3 : ack ∈ ACK

THEN

act1 : st_rcv≔r_ready

act2 : Packet_ack_snd≔Packet_ack_snd ∪ {ack↦rn}
END
S_receive_ack ≙
ANY
ack
WHERE
grd1 : st_snd=wait_ack
grd2 : $t \leq time$
grd3 : rn > sn
grd4 : (ack↦rn) ∈ Packet_ack_snd
THEN
act1 : st_snd≔s_ready
act2 : t≔0
act3 : sn≔rn
act4 : Packet_ack_rcv≔Packet_ack_rcv ∪ {ack↦rn}
act5 : Packet_ack_snd≔Packet_ack_snd \ {ack↦rn}
END
Re_send_req ≙
ANY
msg
WHERE
grd1 : st_snd=wait_ack
grd2 : t=time
grd3 : rcr<RCR_MAX
grd4 : msg ∈ Messages
THEN
act1 : st_snd≔wait_ack
act2 : rcr≔rcr+1
act3 : t≔0
act4 : Packet_msg_snd≔Packet_msg_snd∪{msg↦sn}
act5 : oldSn≔sn
act6 : oldRn≔rn
END
receive_old_req ≙
ANY
msg
WHERE
grd1 : sn≠rn
grd2 : st_rcv=r_ready
grd3 : (msg↦sn) ∈ Packet_msg_rcv
grd4 : (msg↦sn) ∈ Packet_msg_snd
THEN
act1 : st_rcv≔r_ready
act2 : Packet_msg_snd≔Packet_msg_snd \ {msg↦sn}
act3 : flag_old≔TRUE
END
re_send_ack ≙

WHEN
grd1 : st_rcv=r_ready
grd2 : r=TRUE
THEN
act1 : st_rcv≔r_ready
act2 : r≔FALSE
END
receive_old_ack ≙
ANY
ack
WHERE
grd1 : st_snd=wait_ack
grd2 : (ack↦oldRn) ∈ Packet_ack_rcv
grd3 : oldSn <sn
grd4 : oldSn<oldRn
THEN
act1 : st_snd≔wait_ack
act2 : oldSn ≔ oldSn+1
END
tick_tock ≙
BEGIN
act1 : t≔t+1
END

Proofs: The Proof Obligation Generator of the Rodin Platform produces 28 proof obligations, with 9 of them proved interactively.

The synchronization between events is verified by using ProB plugin.

C. Verification of SAW properties with UPPAAL

We transfer our Event-B model to UPPAAL model by applying the mapping described above. The transferred UPPAAL model is composed of 4 automatons, as shown in Fig.4, Fig.5 and Fig.6.

Fig.4 represents the sender automaton: Going from state s_ready to wait_ack , the message is transmitted with the corresponding information and rcr is reset. In state wait_ack there are three possibilities: in case the maximum number of transmissions has not been reached and the timer t expires, the sender retransmits the message and remains in this state. If the ack is received within time (t<time), the sender moves to the initial state s_ready. When the sender receives duplicate acknowledgment, it remains in wait_ack state.

In state r_ready, the receiver (see fig.5) is waiting for the first message to arrive, once received, the variable rn is incremented by one and the receiver enters the state processing, then it sends the ack to the sender and moves to r_ready state. when the receiver receives an old message, it remains in r_ready state.

These automatons communicate between them via synchronization channels:

chan send_req, rec_req, send_ack, rec_ack;

send_req! means sending a message in the channel send_req.

send_req? means waiting for a message from the channel send_req.

The variable t is used in UPPAAL as a clock to model the timer.

After using the simulator to ensure that the model behaves as the system we wanted to model (and sometimes also to detect some errors in the original design), the next phase is to check that the model verifies the properties. These properties are written as logic CTL formulas:

A<> t>time imply not Sender.wait_ack: for any t>time, the state wait_ack cannot be reached.

E[] Sender.wait_ack imply ! Sender.s_ready : the sender may be still in the state wait_ack until the state s_ready will be activated.

E[] Receiver.processing imply Sender.wait_ack: when the receiver is in processing state, the sender state wait_ack is already reached.

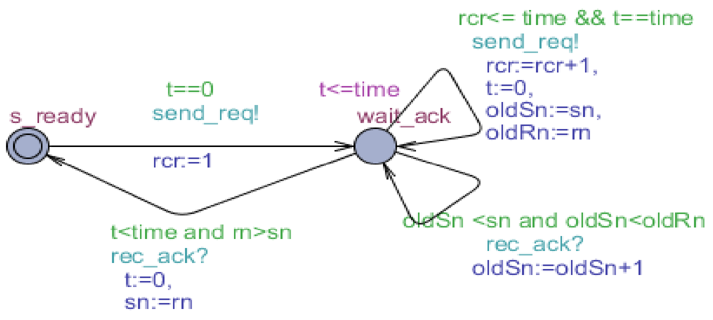

Fig. 4. The Sender in UPPAAL

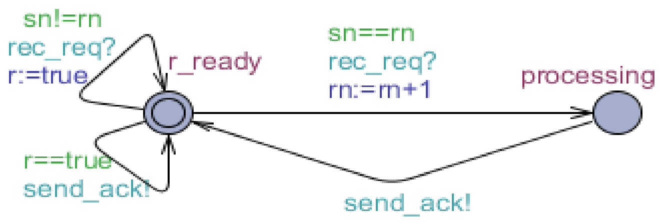

Fig. 5. The Receiver in UPPAAL

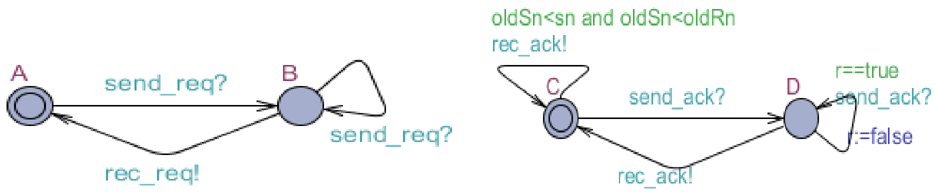

Fig. 6. The two Channels in UPPAAL

V. CONCLUSION

In this paper we proposed an approach to modeling and verification of time-constrained ARQ protocols. The main principle of these protocols is the retransmissions of lost packets upon time-out. We used Event-B formal method with its Rodin tool and the model checker UPPAAL. This approach consists in exploiting the pattern for modeling of retransmission time-out within Event-B and transform this pattern to pattern in UPPAAL to verify the time constraints properties. For this purpose, a formal link between the semantics of Event-b model and UPPAAL was established.

We applied our approach on the stop-and-wait ARQ protocol which is a basic Automatic Repeat Request protocol and we have refined our Event-b model to add more specifications and we have also verified the model with ProB to ensuring the synchronization between events, after that we mapped the model into UPPAAL to verify the time constraints of this protocol.

REFERENCES

[1] N. Hoang Anh, L. Hanzo, "Hybrid automatic-repeat-reQuest systems for cooperative wireless communications". Communications Surveys & Tutorials, IEEE 16.1, pp. 25-45, 2014.
[2] B. Eike, R. Devillers, M. Koutny, Petri net algebra. Springer Science & Business Media, 2013.
[3] C, Chin-Liang, R Char-Tung Lee, Symbolic logic and mechanical theorem proving. Academic press, 2014.
[4] J.R. Abrial, Modeling in Event-B: system and software engineering, Cambridge University Press, 2010.
[5] D. Cansell, D. Mery, "The event-B Modelling Method: Concepts and Case Studies", Springer, Heidelberg, pp. 33-140, 2007.
[6] R.J. Back, On the correctness of refinement steps in program development, Department of Computer Science, University of Helsinki, 1978.
[7] B. Gerd, A. David, K.G. Larsen, "A tutorial on uppaal .Formal methods for the design of real-time systems". Springer Berlin Heidelberg, pp. 200-236, 2004.
[8] C. Lawrence, et al., Non-functional requirements in software engineering. Vol. 5. Springer Science & Business Media, 2012.
[9] J.R. Abrial, J.B. Butler, S. Hallerstede, T.S. Hoang, F. Mehta, L. Voisin, Rodin: "an open toolset for modelling and reasoning in Event-B". , Vol 12, No.6, pp.447-466, 2010.
[10] C. Jones, I. Oliver, A. Romanovsky, and E. Troubitsyna, RODIN (rigorous open development environment for complex systems) University of Newcastle upon Tyne, Computing Science, 2005.
[11] O. Ligot, J. Bendisposto M. Leuschel, "Debugging event-b models using the prob disprover plug-in". Proceedings AFADL, 7, 2007.
[12] B, Johan, W. Yi. Timed automata: "Semantics, algorithms and tools". Lectures on concurrency and petri nets. Springer Berlin Heidelberg, pp. 87-124, 2004.
[13] G. E. Gallasch, Parametric verification of the class of stop-and-wait protocols (Doctoral dissertation, University of South Australia). 2007.
[14] Abrial, J.R., The B-book: assigning programs to meanings. Cambridge University Press, 2005.

[15] F. Melvin, First-order logic and automated theorem proving. Springer Science & Business Media, 2012.
[16] J. Thomas, Set theory. Springer Science & Business Media, 2013.
[17] S. Hallerstede, "On the purpose of Event-B proof obligations, In Abstract state machines", B and Z, Springer Berlin Heidelberg, pp. 125-138, 2008.
[18] M. Reynolds, "An axiomatization of full computation tree logic". The Journal of Symbolic Logic 66.03, pp.1011-1057, 2011