Hindawi Publishing Corporation
Journal of Computer Networks and Communications
Volume 2014, Article ID 352071, 11 pages
http://dx.doi.org/10.1155/2014/352071

Research Article

A Mechanically Proved and an Incremental Development of the Session Initiation Protocol INVITE Transaction

Rajaa Filali and Mohamed Bouhdadi

LMPHE laboratory, Faculty of sciences, University of Mohammed V, 4 Street Ibn Batouta, PB 1014 RP Rabat, Morocco

Correspondence should be addressed to Rajaa Filali; rajaa.filali@gmail.com

Received 31 May 2014; Revised 25 September 2014; Accepted 19 October 2014; Published 19 November 2014

Academic Editor: Kyandoghere Kyamakya

The Session Initiation Protocol (SIP) is an application layer signaling protocol used to create, manage, and terminate sessions in an IP based network. SIP is considered as a transactional protocol. There are two main SIP transactions, the INVITE transaction and the non-INVITE transaction. The SIP INVITE transaction specification is described in an informal way in Request for Comments (RFC) 3261 and modified in RFC 6026. In this paper we focus on the INVITE transaction of SIP, over reliable and unreliable transport mediums, which is used to initiate a session. In order to ensure the correctness of SIP, the INVITE transaction is modeled and verified using event-B method and its Rodin platform. The Event-B refinement concept allows an incremental development by defining the studied system at different levels of abstraction, and Rodin discharges almost all proof obligations at each level. This interaction between modeling and proving reduces the complexity and helps in assuring that the INVITE transaction SIP specification is correct, unambiguous, and easy to understand.

1. Introduction

Session Initiation Protocol is a network communications protocol commonly employed for Voice over IP (VoIP) signaling. It is based on request/response transaction model. Each transaction consists of a client request that invokes a particular method on the server and at least one response. The two main SIP transactions are the INVITE transaction for setting up a session, and the non-INVITE transaction for maintaining and closing down a session. Their specifications are defined in Request for Comments (RFC) 3261 [1] and they have been modified in RFC 6026 [2].

A large number of the practical protocols have only informal specifications. Several formal methods have been applied to analyze these protocols, such as model checking [3] and theorem proving [4]. However, only a few papers on analyzing SIP using Colored Petri Nets (CPNs) have been published [5–8]. CPNs are based on model checking which verifies the implementation of the system.

Recently a new method Event-B [9] has been developed by Abrial who has developed the B method [10] and the Z language [11]. In this paper, we use Event-B to model and prove the SIP INVITE transaction over reliable and unreliable transport medium. The most important benefit of using Event-B is its capability to use abstraction and refinement [12]. Indeed, in this approach the modeling process starts with an abstraction of the system which specifies the goals of the system. The abstract level of our Event-B model shows these goals in a very general way, and then during refinement levels features of the protocol are modeled and the goals are achieved in a detailed way. Moreover the Rodin tool [13] permits an automated proof of the different models of the system.

In this paper, we use Event-B to model and prove the SNMP protocol. The most important benefit of using Event-B is its capability to use abstraction and refinement [7].

The remainder of the paper is organized as follows. Section 2 gives a brief overview of Event-B. Section 3 provides the requirements which are informally defined. A refinement strategy is proposed in Section 4. Finally, in Section 5, the formal development is presented.

2. Overview of Event-B

Before Event-B is a formal method for specifying, modeling, and reasoning about systems, especially complex systems such as an electronic circuit, an airline seat booking system, a PC operating system, a network routing program, a nuclear plant control system, and a Smartcard electronic purse. Event-B has evolved from classical B.

Key features of Event-B are the use of set theory as a modeling notation, the use of refinement to represent systems at different abstraction levels and the use of mathematical proof to verify consistency between refinement levels. From a given model M1, a new model M2 can be built as a refinement of M1. In this case, model M1 is called an abstraction of M2, and model M2 is said to be a concrete version of M1. A concrete model is said to refine its abstraction. Each event of a concrete machine refines an abstract event or refines skip. An event that refines skip is referred to as a new event since it has no counterpart in the abstract model. An Event-B model has two parts, context and machine. Each context specifies the static properties of the system, including sets, axioms, and constants. Each machine specifies the dynamic part of the system, including variables, invariants, and events. Variables represent the current state of the system and invariants specify the global specification of the variables and system behaviors.

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 $S ::= x := E(v) \mid x := z :\mid P(z)$. 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 nondeterministic substitution, $x := z :\mid P(z)$, it is possible to choose nondeterministically 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.

The Rodin is the tool of the Event-B. It allows formal Event-B models to be created with an editor. It generates proof obligations that can be discharged either automatically or interactively. Rodin is modular software and many extensions are available. These include alternative editors, document generators, team support, and extensions (called plugins) some of which include support decomposition and records.

3. Informal Description of SIP INVITE Transaction

The INVITE client and server transactions are defined in RFC 3261 and its modifications are presented in RFC 6026 using two state machines.

3.1. INVITE Client Transaction. When Transaction User (TU) at the client side wants to initiate a session, it creates an INVITE client transaction and passes an INVITE request to the transaction.

(i) An INVITE client transaction has five different states: (1) *calling*, (2) *proceeding*, (3) *accepted*, (4), *completed*, and (5) *terminated*.

(ii) The initial state, *calling*, must be entered when the TU initiates a new client transaction with an INVITE request.

(iii) For any transport (reliable or unreliable), the client transaction must start timer B with a value of 64∗T1 seconds (Timer B controls transaction timeouts).

(iv) If an unreliable transport is being used, the client transaction must start timer A with a value of T1 (Timer A controls request retransmissions).

(v) When timer A fires, the client transaction must retransmit the request by passing it to the transport layer and must reset the timer with a value of 2∗T1. When timer A fires 2∗T1 seconds later, the request must be retransmitted again.

(vi) If the client transaction is still in the *calling* state when timer B fires, the client transaction should inform the TU that a timeout has occurred.

(vii) If the client transaction receives a provisional response while in the *calling* state, it transitions to the *proceeding* state.

(viii) If a Transport Err (Error) occurs or timer B expires, the client transaction moves to the *terminated* state and informs its TU immediately.

(ix) In the *proceeding* state, the client transaction should not retransmit the request any longer. Furthermore, the provisional response must be passed to the TU. Any further provisional responses must be passed up to the TU while in the *proceeding* state.

(x) When in either the *calling* or *proceeding* states, reception of a response with status code from 300–699 must cause the client transaction to transition to *completed*.

(xi) The client transaction should start timer D when it enters the *completed* state, with a value of at least 32 seconds for unreliable transports, and a value of zero seconds for reliable transports. Timer D reflects the amount of time that the server transaction can remain in the *completed* state when unreliable transports are used. This is equal to timer H in the INVITE server transaction, whose default is 64∗T1.

(xii) If timer D fires while the client transaction is in the *completed* state, the client transaction must move to the *terminated* state. When in either the *calling* or *proceeding* states, reception of a 2xx response must cause the client transaction to enter the *accepted* state.

(xiii) The purpose of the *accepted* state, which presents the correction of INVITE client transaction according to RFC 6026, is to allow the client transaction to continue to exist to receive and pass to its TU any retransmissions of the 2xx response. When this state is entered, timer M must be started. This timer reflects the amount of time that the TU will wait for retransmissions of the 2xx responses. When timer M fires, transaction enters the *terminated* state.

3.2. INVITE Server Transaction. The INVITE server transaction is created by TU on the server side when it receives an INVITE request.

(i) The INVITE server transaction can enter five different states: (1) *proceeding*, (2) *accepted*, (3) *completed*, (4) *confirmed*, and (5) *terminated.*

(ii) Initially, the INVITE server transaction enters the *proceeding* state when it is created.

(iii) The server transaction must generate a 100 (Trying) response unless it knows that the TU will generate a provisional or final response within 200 ms, in which case it may generate a 100 (Trying) response.

(iv) While in the proceeding state, if the TU passes a response with status code from 300 to 699 to the server transaction, the response must be passed to the transport layer for transmission, and the state machine must enter the *completed* state.

(v) When the TU on the server side forwards a final success response (2xx) to the server transaction, the transaction delivers this response to the transport layer for transmission and enters the accepted state. Retransmissions of the 2xx response are handled by TU, not by the server transaction.

(vi) The purpose of the *accepted* state, which presents the modification of INVITE server transaction according to RFC 6026, is to absorb retransmissions of an accepted INVITE request. Any such retransmissions are absorbed entirely within the server transaction.

(vii) Timer L is started when the *accepted* state is entered. This timer reflects the wait time for retransmissions of 2xx responses. When timer L fires, transaction enters the *terminated* state.

(viii) For unreliable transports, the server transaction must start timer G to control the time for each retransmission.

(ix) If timer G fires, the response is passed to the transport layer once more for retransmission, and timer G is set to fire in min $(2*T1, T2)$ seconds. From then on, when timer G fires, the response is passed to the transport again for transmission, and timer G is reset with a value that doubles, unless that value exceeds T2, in which case it is reset with the value of T2.

(x) When the *completed* state is entered, timer H must be set to fire in $64*T1$ seconds for all transports. Timer H determines when the server transaction abandons retransmitting the response.

(xi) If an ACK is received while the server transaction is in the *completed* state, the server transaction must transition to the *confirmed* state. As timer G is ignored in this state, any retransmissions of the response will cease.

(xii) If timer H fires while in the *completed* state, it implies that the ACK was never received. In this case, the server transaction must transition to the *terminated* state, and must indicate to the TU that a transaction failure has occurred.

(xiii) The purpose of the *confirmed* state is to absorb any additional ACK messages that arrive, triggered from retransmissions of the final response. Once timer I fires, the server must transition to the *terminated* state.

```
SETS
    REQUESTS
    RESPONSES
```

Box 1

```
VARIABLES
    cl_sent
    sr_got,
    sr_sent
    cl_got
```

Box 2

4. Refinement Strategy

The development is made of an initial model followed by some refinements.

The initial model is high level abstraction showing that the transaction has a client side and a server side; the client and the server can both send and receive a message.

The first refinement: in this model we specify the requests and the responses sent by the client and the server, respectively; we introduce also their different states.

The second refinement contains the introduction of the timer constraints for reliable transport.

The third refinement contains the introduction of the timer constraints for unreliable transport.

5. Formal Development

5.1. Initial Model. In this initial model, we just formalize a communication between client and server by means of messages (Algorithm 1).

First, we define two carrier sets REQUESTS and RESPONSES: they describe, respectively, the set of messages which can be sent by the client and the set of messages which can be sent by the server (see Box 1).

Then the variables *cl_sent*, *sr_got*, *sr_sent*, and *cl_got* are introduced to define, respectively, the set of requests sent by the client, the requests received successfully by the server, the set of responses sent by the server, and successfully received responses by the client (see Box 2).

We now define the invariants. In invariants (inv1 and inv2), the set *cl_sent* and *sr_got* are simply typed as subset of REQUESTS. As expected in invariants (inv3 and inv4), the set *sr_sent* and *cl_got* are defined as a subset of RESPONSES.

```
INVARIANTS
inv1: $cl\_sent \subseteq REQUESTS$
inv2: $sr\_got \subseteq REQUESTS$
inv3: $sr\_sent \subseteq RESPONSES$
inv4: $cl\_got \subseteq RESPONSES$
inv5: $\forall\ m \cdot m \in REQUESTS \land m \notin cl\_sent \Rightarrow m \notin sr\_got$
inv6: $\forall\ l \cdot l \in RESPONSES \land l \notin sr\_sent \Rightarrow l \notin cl\_got$
```

Box 3

```
      client_Send $\hat{=}$
ANY
      msg
WHERE
      grd4: $msg \in REQUESTS$
      grd3: $msg \notin cl\_sent$
THEN
      act6: $cl\_sent := cl\_sent \cup \{msg\}$
END
      server_Receive $\hat{=}$
ANY
      msg
WHERE
      grd5: $msg \in REQUESTS$
      grd6: $msg \in cl\_sent$
THEN
      act5: $sr\_got := sr\_got \cup \{msg\}$
END
      server_send $\hat{=}$
ANY
      msg
WHERE
      grd1: $msg \in RESPONSES$
      grd2: $msg \notin sr\_sent$
      grd3: $sr\_got \neq \emptyset$
THEN
      act2: $sr\_sent := sr\_sent \cup \{msg\}$
END
      client_Receive $\hat{=}$
ANY
      msg
WHERE
      grd5: $msg \in RESPONSES$
      grd6: $msg \in sr\_sent$
THEN
      act5: $cl\_got := cl\_got \cup \{msg\}$
END
```

Algorithm 1: Events of initial model.

The invariant (inv5) denotes that if any message is not in the set *cl_sent*, it must not be in the set *sr_got*. The same for (inv6) (see Box 3).

Finally, we define four events in our abstract model. An event *client_send* represents the sending request from the client to the server. An event *server_receive* represents the request received successful by the server, guards of this event state that a message should be in the set *cl_sent*. An event *server_send* represents the response sent from the server to the client, the guard of this event state that the set *sr_got* should not be empty. The last event *client_got* represents the response received successful by the client.

```
CONSTANTS
      Ready
      Calling
      Proceeding
      Accepted
      Completed
      Confirmed
      Terminated
      INVITE
      ACK
      r2xx
      r3xx_r699
      r1xx
```

Box 4

Proofs. In this initial model, there are 6 invariant preservation proofs, with 2 of them proved interactively.

5.2. First Refinement. In this first refinement, we introduce specific requests (INVITE, ACK) and responses (1xx, 2xx, 3xx-699). We introduce also the different states of the client and the server. First we define the context which contains the carrier set STATES. It represents the different states of client and server. We define also the constants in the context and their associated axioms by the agent (see Box 4).

In order to manipulate states we introduce two new variables (See Box 5):

c_st denotes the current state of client,

s_st denotes the current state of server.

We are now ready to define our events (Algorithm 2).

(i) *Client_send_INVITE* refining the abstract event client_send: the client sends an INVITE and the state calling must be entered.

(ii) *Server_receive_INVITE* refining the abstract event Server_receive: the server receives the INVITE and it enters in proceeding state.

(iii) *Server_send_1xx* refining the abstract event server_send: after receiving the request INVITE, the server sends a provisional response (1xx) to a client and it remains in proceeding state

(iv) *Client_receive_1xx* refining the abstract event client_receive: the client receives the response 1xx and the state proceeding must be entered.

(v) *Server_send_2xx* refining the abstract event server send: while the server is in proceeding it can send the final success response to a client and it enters in accepted state.

```
VARIABLES
    s_st
    c_st
INVARIANTS
inv1: s_st ∈ STATES
inv2: c_st ∈ STATES
inv3: c_st = Calling ⇒ INVITE ∈ cl_sent
inv4: s_st = Proceeding ⇒ INVITE ∈ sr_got ∨ r1xx ∈ sr_sent
inv5: c_st = Proceeding ⇒ r1xx ∈ cl_got
inv6: s_st = Accepted ⇒ r2xx ∈ sr_sent
inv7: c_st = Accepted ⇒ r2xx ∈ cl_got
inv8: s_st = Completed ⇒ r3xx_r699 ∈ sr_sent
inv9: c_st = Completed ⇒ r3xx_r699 ∈ cl_got ∨ ACK ∈ cl_sent
inv10: s_st = Confirmed ⇒ ACK ∈ sr_got
```

Box 5

(vi) *Client_receive_2xx* refining the abstract event client_receive: the client receives the final success response 2xx and the state proceeding must be entered.

(vii) *Server_send_3xx-699* refining the abstract event server_receive: when the server sends the final nonsuccess response, it enters in completed state.

(viii) *Client_receive_3xx-699* refining the abstract event server_receive: while the client is in calling or proceeding state and it receives the nonsuccess response 3xx-6xx, the state completed must be entered.

(ix) *Client_send_ACK* refining the abstract event client_send: the client sends an acknowledgment (ACK) to the server after receiving the nonsuccess response.

(x) *Server_receive_ACK* refining the abstract event server_receive: the server receives the ACK and it enters in confirmed state.

(xi) We add two new events *client_final_state* and *server_final_state* as anticipated events.

Proofs. The proof obligation generator of the Rodin Platform produces 49 proof obligations, with 7 of them proved interactively.

5.3. Second Refinement. In this refinement, we introduce the time constraints for reliable transport. We add six timers B, H, D, I, L, and M: timer B controls transaction timeouts, timer H determines when the server transaction abandons retransmitting the response, timer D reflects the amount of time that the server transaction can remain in the “completed” state, timer I determines the server state from confirmed to terminated, timer M reflects the amount of time that the TU will wait for retransmissions of the 2xx responses, and timer L reflects the wait time for retransmissions of 2xx responses. For modeling these timers, we define six variables that represent each of them, and others six Boolean variables for whether the timer is held or not (see Box 6).

```
VARIABLES
    H
    B
    I
    D
    Temp_B
    Temp_H
    Temp_D
    Temp_I
    Temp_M
    Temp_L
```

Box 6

Concerning events, we add six new events Expire B, Expire_D, Expire_H, Expire_I, Expire_M, and Expire_L; they represent when each timer will expire. We add also six time progression events: Tick_Tock_B, Tick_Tock_D, Tick_Tock_H, Tick_Tock_I, Tick_Tock_L, and Tick_Tock_ M (Algorithm 3).

We refine some events:

Client_send_INVITE, client_receive_1xx, server_send_3xx-699, client_receive_3xx-699, server_receive_ACK, client_receive_2xx, server_send_2xx.

When the client transaction sends an invite, the client transaction must start timer B (we refine the abstract event client send INVITE by adding the action temp B := TRUE). If the client transaction is still in the *calling* state when timer B fires, the client transaction should pass to the final state *terminated*. If the client transaction receives a provisional response while in the *calling* state, it transitions to the *proceeding* state; then the timer B must turn off (we refine the abstract event client_receive_1xx by adding the action temp B := FALSE). The client transaction should start timer D when it enters the *completed* state, so we add temp D := TRUE as action in the event client_receive_3xx-699. If timer D fires while the client transaction is in the *completed* state, the client transaction must move to the *terminated* state.

When the *completed* state is entered, timer H must be set to fire (we refine the abstract event server send 3xx 699 by

```
  INITIALISATION $\hat{=}$
BEGIN
    $act7$: $cl\_sent := \emptyset$
    $act8$: $cl\_got := \emptyset$
    $act9$: $sr\_sent := \emptyset$
    $act10$: $sr\_got := \emptyset$
    $act11$: c_st := Ready
    $act12$: s_st := Ready
END
  client_Send_INVITE $\hat{=}$
REFINES
    client_Send
ANY
    $msg$
WHERE
    $grd4$: $msg \in REQUESTS$
    $grd3$: $msg \notin cl\_sent$
    grd5: msg = INVITE
    grd6: c_st = Ready
THEN
    $act6$: $cl\_sent := cl\_sent \cup \{msg\}$
    act7: c_st := Calling
END
  server_receive_INVITE $\hat{=}$
REFINES
    server_Receive
ANY
    $msg$
WHERE
    $grd5$: $msg \in REQUESTS$
    $grd6$: $msg \in cl\_sent$
    grd7: msg = INVITE
    grd8: s_st = Ready
THEN
    $act5$: $sr\_got := sr\_got \cup \{msg\}$
    act6: s_st := Proceeding
END
  server_send_1xx $\hat{=}$
REFINES
    server_send
ANY
    $msg$
WHERE
    $grd1$: $msg \in RESPONSES$
    $grd2$: $msg \notin sr\_sent$
    $grd3$: $sr\_got \neq \emptyset$
    grd4: msg = r1xx
    grd5: s_st = Proceeding
THEN
    $act2$: $sr\_sent := sr\_sent \cup \{msg\}$
    $act3$: s_st := Proceeding
END
  client_Receive_1xx $\hat{=}$
REFINES
    client_Receive
ANY
    $msg$
```

Algorithm 2: Continued.

```
WHERE
        grd5: $msg \in RESPONSES$
        grd6: $msg \in sr\_sent$
        grd7: msg = r1xx
        grd8: c_st = Calling
THEN
        act5: $cl\_got := cl\_got \cup \{msg\}$
        act6: c_st := Proceeding
END
    server_send_2xx $\hat{=}$
REFINES
        server_send
ANY
        msg
WHERE
        grd1: $msg \in RESPONSES$
        grd2: $msg \notin sr\_sent$
        grd3: $sr\_got \neq \emptyset$
        grd4: msg = r2xx
        grd5: s_st = Proceeding
THEN
        act2: $sr\_sent := sr\_sent \cup \{msg\}$
        act3: s_st := Accepted
END
    client_Receive_2xx $\hat{=}$
REFINES
        client_Receive
ANY
        msg
WHERE
        grd5: $msg \in RESPONSES$
        grd6: $msg \in sr\_sent$
        grd7: msg = r2xx
        grd8: c_st = Calling $\vee$ c_st = Proceeding
THEN
        act5: $cl\_got := cl\_got \cup \{msg\}$
        act6: c_st := Accepted
END
    server_send_3xx-699 $\hat{=}$
REFINES
        server_send
ANY
        msg
WHERE
        grd1: $msg \in RESPONSES$
        grd2: $msg \notin sr\_sent$
        grd3: $sr\_got \neq \emptyset$
        grd4: msg = r3xx_r699
        grd5: s_st = Proceeding
THEN
        act2: $sr\_sent := sr\_sent \cup \{msg\}$
        act3: s_st := Completed
END
    client_Receive_3xx-699 $\hat{=}$
REFINES
        client_Receive
ANY
        msg
```

Algorithm 2: Continued.

```
**WHERE**
    *grd5*: *msg* ∈ *RESPONSES*
    *grd6*: *msg* ∈ *sr_sent*
    grd7: msg = r3xx_r699
    grd8: c_st = Calling ∨ c_st = Proceeding
**THEN**
    *act5*: *cl_got* := *cl_got* ∪ {*msg*}
    *act6*: c_st := Completed
**END**
  **client_send_ACK** ≙
**REFINES**
    client_Send
**ANY**
    *msg*
**WHERE**
    *grd4*: *msg* ∈ *REQUESTS*
    *grd3*: *msg* ∉ *cl_sent*
    grd5: msg = ACK
    grd6: c_st = Completed
**THEN**
    *act6*: *cl_sent* := *cl_sent* ∪ {*msg*}
    *act7*: c_st := Completed
**END**
  **server_Receive_ACK** ≙
**REFINES**
    server_Receive
**ANY**
    *msg*
**WHERE**
    *grd5*: *msg* ∈ *REQUESTS*
    *grd6*: *msg* ∈ *cl_sent*
    grd7: msg = ACK
    grd8: s_st = Completed
**THEN**
    *act5*: *sr_got* := *sr_got* ∪ {*msg*}
    *act6*: s_st := Confirmed
**END**
    **Client_final_state** ≙
      STATUS
    **anticipated**
**BEGIN**
    *act1*: c_st := Terminated
**END**
  **Server_final_state** ≙
      ***STATUS***
    **anticipated**
**BEGIN**
    act1: s_st := Terminated
**END**
```

Algorithm 2: Events of first refinement.

adding the action temp H := TRUE). If timer H fires while in the *completed* state, it implies that the ACK was never received. In this case, the server transaction must transition to the *terminated* state and must indicate to the TU that a transaction failure has occurred.

When *confirmed* state is entered, timer I is set to fire in zero seconds for reliable transports. We add temp (I := TRUE) as action in the refined event server_receive_ACK. Once timer I fires, the server MUST transition to the *terminated* state.

Timer L is started when the *accepted* state is entered. When timer L fires, transaction enters the *terminated* state.

When *accepted* state is entered, timer M must be started. When timer M fires, transaction enters the *terminated* state.

Proofs. There are 10 invariant preservation proofs. They are all straightforward and easily proved automatically by the Rodin platform prover.

```
    Expipre_B $\hat{=}$
REFINES
        client_final_state
WHEN
        grd3: r1xx $\notin$ cl_got
        grd1: B = 64 $*$ cst
        grd4: c_st = Calling
THEN
        act1: c_st := Terminated
        act2: B := 0
        act3: Temp_B := FALSE
END
    Expire_H $\hat{=}$
REFINES
        server_final_state
WHEN
        grd3: ACK $\notin$ sr_got
        grd1: H = 64 $*$ cst
        grd2: s_st = Completed
THEN
        act1: s_st := Terminated
        act2: H := 0
        act3: Temp_H := FALSE
END
    Expire_D $\hat{=}$
REFINES
        client_final_state
WHEN
        grd1: Temp_D = TRUE
        grd2: c_st = Completed
THEN
        act1: c_st := Terminated
        act2: Temp_D := FALSE
END
    Expire_I $\hat{=}$
REFINES
        Server_final_state
WHEN
        grd2: Temp_I = TRUE
        grd1: s_st = Confirmed
THEN
        act1: s_st := Terminated
        act2: Temp_I := FALSE
END
    Expire_M $\hat{=}$
REFINES
        Client_final_state
WHEN
        grd2: Temp_M = TRUE
        grd1: c_st = Accepted
THEN
        act1: c_st := Terminated
        act2: Temp_M := FALSE
END
    Expire_L $\hat{=}$
REFINES
        Server_final_state
```

Algorithm 3: Continued.

```
WHEN
        grd2: Temp_L = TRUE
        grd1: s_st = Accepted
THEN
        act1: s_st := Terminated
        act2: Temp_L := FALSE
END
```

Algorithm 3: Events of second refinement.

```
    Resend_INVITE  ≙
WHEN
    grd1: c_st = Calling
    grd2: A = T1
THEN
    act1: T1 := 2 ∗ T1
    act2: cl_sent := cl_sent ∪ {INVITE}
    act3: c_st := Calling
    act4: A := 0
END
    Resend_3xx-699  ≙
ANY
    data
WHERE
    grd1: data = T1
    grd2: G = min ({data, T2})
    grd3: s_st = Completed
THEN
        act1: sr_sent := sr_sent ∪ {r3xx_r699}
        act2: s_st := Completed
        act3: T1 := 2 ∗ T1
        act4: G := 0
END
```

Algorithm 4: Events of third refienement.

5.4. Third Refinement. In this last refinement, we introduce the time constraints for unreliable transport. New variables are declared, "A" as the timer that controls request retransmissions, "G" controls the time for each retransmission, and two Boolean variables temp A and temp G for whether the timer is held or not. We add also two new events.

Resend_INVITE. When timer A fires, the client transaction must retransmit the request INVITE and must reset the timer with a value of 2∗T1. When timer A fires 2∗T1 seconds later, the request must be retransmitted again.

Resend_3xx-699. If timer G fires, the response 3xx-699 is passed to the transport layer once more for retransmission, and timer G is set to fire in min (2∗T1, T2) seconds. From then on, when timer G fires, the response is passed to the transport again for transmission, and timer G is reset with a value that doubles, unless that value exceeds T2, in which case it is reset with the value of T2 (Algorithm 4).

Table 1: Proof statistics.

Model	Total number of POs	Automatic Proof	Interactive Proof
Initial model	6	4	2
First refinement	49	42	7
Second refinement	10	10	0
Third refinement	14	13	1
Total	79	69 (87%)	10 (13%)

We refine the events expire_I, expire_M, expire_L, and expire_D by adding, respectively, the actions d = 32, I = 10∗cst, M = 64∗cst, and L = 64∗cst, where cst = 500 ms.

Proofs. This refinement requires 14 proofs, all proved automatically except one.

5.5. Proof Statistics. Table 1 describes the proof statistics of the formal development of SIP INVITE transaction in the Rodin tool. These statistics measure the size of the model, the proof obligations generated and discharged by the Rodin platform, and those that are interactively proved.

6. Conclusion

In this paper, we have modeled and proved SIP INVITE transaction over reliable and unreliable transport medium using Event-B

We have explained our approach using refinement, which allows us to achieve a very high degree of automatic proof. The powerful support is provided by the Rodin tool. Rodin proof is used to generate the proof obligations and to discharge those obligations automatically and interactively. Modeling and analyzing SIP specification using formal methods can help in assuring correctness, unambiguity, and clarity of the SIP protocol. Since a well-defined and verified protocol specification can reduce the cost for its implementation and maintenance, modeling and analysis are important steps of the protocol development life-cycle from the point view of protocol engineering.

In the future work, we would model and prove the non-INVITE transaction over reliable and unreliable mediums.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] J. Rosenberg, H. Schulzrinne, G. Camarillo et al., "Sip: session initiation protocol," Tech. Rep. RFC 3261, Internet Engineering Task Force, 2002.

[2] R. Sparks and T. Zourzouvillys, "Correct transaction handling for 2xx responses to session initiation protocol (sip) invite requests," RFC 6026, 2010, http://www.ietf.org/rfc/rfc6026.txt.

[3] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, Cambridge, Mass, USA, 1999.

[4] C.-L. Chang, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1973.

[5] L. G. Ding and L. Liu, *Modeling and Analysis of the INVITE Transaction of the Session Initiation Protocol Using Colored Petri Nets*, Springer, 2008.

[6] L. Liu, "Verification of the sip transaction using colored Petri nets," in *Proceedings of the 23nd Australasian Conference on Computer Science*, vol. 91, pp. 75–84, Australian Computer Society, 2009.

[7] S. Kızmaz and M. Kırcı, "Verification of session initiation protocol using timed colored petri net," *International Journal of Communications, Network & System Sciences*, vol. 4, no. 3, pp. 170–179, 2011.

[8] S. Barakovic, D. Jevtic, and J. Barakovic Husic, "Modeling of session initiation protocol invite transaction using colored Petri nets," in *Proceedings of the 8th International Conference on Modeling and Simulation (ICMS '12)*, 2012.

[9] J. R. Abrial, *Modeling in Event-B: System and Software Engineering*, Cambridge University Press, 2010.

[10] J. R. Abrial, *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, Cambridge, UK, 2005.

[11] J. R. Abrial, "$B^{\#}$: toward a synthesis between Z and B," in *ZB 2003: Formal Specification and Development in Z and B*, D. Bert, J. P. Bowen, S. King, and M. Waldén, Eds., vol. 2651 of *Lecture Notes in Computer Science*, pp. 168–177, Springer, Berlin, Germany, 2003.

[12] R. J. Back, *On the Correctness of Refinement Steps in Program Development*, Department of Computer Science, University of Helsinki, Helsinki, Finland, 1978.

[13] 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.