Modelling and Analysis of the INVITE Transaction of the Session Initiation Protocol Using Coloured Petri Nets

Lay G. Ding and Lin Liu

School of Computer and Information Science
University of South Australia
Mawson Lakes, SA 5095, Australia
dinlg001@students.unisa.edu.au, lin.liu@unisa.edu.au

Abstract. The Session Initiation Protocol (SIP) is a control protocol developed by the Internet Engineering Task Force for initiating, modifying and terminating multimedia sessions over the Internet. SIP uses an INVITE transaction to initiate a session. In this paper, we create a Coloured Petri Net (CPN) model for the INVITE transaction. Then we verify the general properties of the INVITE transaction by analysing the state space of the CPN model. The analysis results show that in most cases the INVITE transaction behaves as expected. However, in some circumstances, the transaction may terminate in an undesirable state while one communication party is still waiting for a response from its peer. Hence, we propose a set of changes to the INVITE transaction to correct the above problem. The result has shown that this revised INVITE transaction satisfies the properties that we have specified, and the undesirable terminal state has been eliminated.

Keywords: Session Initiation Protocol, Coloured Petri Nets, protocol verification.

1 Introduction

The popularisation of the Internet has been changing the way of communication in our daily life. A common example is the use of Voice over IP (VoIP). Before a conversation can take place between participants, protocols must be employed to establish a session, then to maintain and terminate the session. The Session Initiation Protocol (SIP) [1] is one of the protocols being used for such purposes.

SIP is developed by the Internet Engineering Task Force (IETF) and published as Request for Comments (RFC) 3261 in 2002 [1]. Besides its increasing popularity and importance in VoIP applications, SIP has been recognised by the 3rd Generation Partnership Project as a signalling protocol and permanent element of the IP Multimedia Subsystem architecture [2]. Thus, it is important to assure that the contents of RFC 3261 are correct, unambiguous, and easy to understand. Modelling and analysing the specification using formal methods can help in achieving this goal. Moreover, from the perspective of protocol engineering, verification is also an important step of the life-cycle of protocol development [3,4], as a well-defined and verified specification will reduce the cost for implementation and maintenance.

SIP is a transaction-oriented protocol that carries out tasks through different transactions. Two main SIP transactions are defined [1], the INVITE transaction and the non-INVITE transaction. In this paper, we aim to verify the INVITE transaction, and consider only the operations over a reliable transport medium. Additionally, this paper will focus on functional correctness of the protocol, thus analysis of performance properties, such as session delay, is beyond its scope. Coloured Petri Nets (CPNs), their state space analysis method and supporting tools have been applied widely in verifying communication protocols, software, military systems, business processes, and some other systems [5,6,7]. However, to our best knowledge, very little work has been published on analysing SIP using CPNs, and only the study of [9,10] have been found. Most of the publications related to SIP are in the areas of interworking of SIP and H.323 [8], and SIP services [9,10]. In [9], the authors have modelled a SIP-based discovery protocol for the Multi-Channel Service Oriented Architecture, which uses the non-INVITE transaction of SIP as one of its basic components for web services in a mobile environment. However, no analysis results have been reported on this SIP-based discovery protocol. In [10], the authors have modelled SIP with the purpose of analysing SIP security mechanism, and have verified the CPN model in a typical attack scenario using state space analysis.

The rest of the paper is organised as follows. Section 2 is an overview of SIP layers and the INVITE transaction. Modelling and analysis of the SIP INVITE transaction are described in Section 3. Section 4 proposes and verifies the revised SIP INVITE transaction. Finally, we conclude the work and suggest future research in Section 5.

2 The INVITE Transaction of SIP

2.1 The Layered Structure of SIP

SIP is a layered protocol, comprising the syntax and encoding layer, transport layer, transaction layer, and transaction user (TU) layer, i.e. the four layers within the top box of Fig. 1.

The syntax and encoding layer specifies the format and structure of a SIP message, which can be either a request from a client to a server, or a response from a server to a client. For each request, a method (such as INVITE or ACK) must be carried to invoke a particular operation on a server. For each response, a status code is declared to indicate the acceptance, rejection or redirection of a SIP request, as shown in Table 1.

The SIP transport layer defines the behaviour of SIP entities in sending and receiving messages over the network. All SIP entities must contain this layer to send/receive messages to/from the underlying transport medium.

On top of SIP transport layer is the transaction layer, including the INVITE transaction and the non-INVITE transaction. An INVITE transaction is initi-

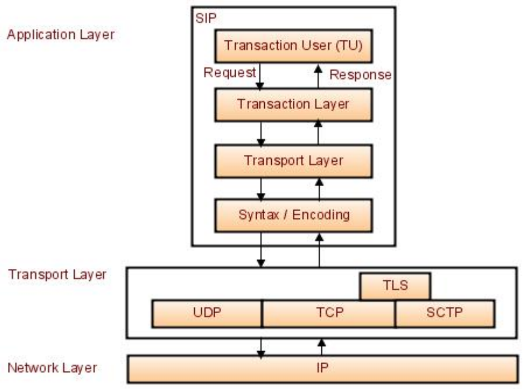


Fig. 1. Layered structure of SIP

Table 1. SIP response messages [1]

Response	**Function**
1xx (100-199)	Provisional - the request was received but not yet accepted
2xx	Success - the request was received successfully and accepted
3xx	Redirection - a further action is required to complete the request
4xx	Client Error - bad syntax found in the request
5xx	Server Error - the server failed to answer the request
6xx	Global Failure - no server can answer the request

ated when an INVITE request is sent; and a non-INVITE transaction is initiated when a request other than INVITE or ACK is sent. Each of the INVITE and non-INVITE transactions consists of a client transaction sending requests and a server transaction responding to requests.

Residing in the top layer of SIP are the TUs, which can be any SIP entity except a stateless proxy [1].

Among the four SIP layers, the transaction layer is the most important layer since it is responsible for request-response matching, retransmission handling with unreliable transport medium, and timeout handling when setting up or tearing down a session.

2.2 The INVITE Transaction

Operations of the client and the server transactions are defined in RFC 3261 by state machines and narrative descriptions. In this section we describe these in detail, where all states, timers, transport errors and responses are shown in *italics*, and all requests are capitalised.

INVITE Client Transaction. Referring to Fig. 2(a), an INVITE client transaction is initiated by the TU with an INVITE request. Meanwhile the client

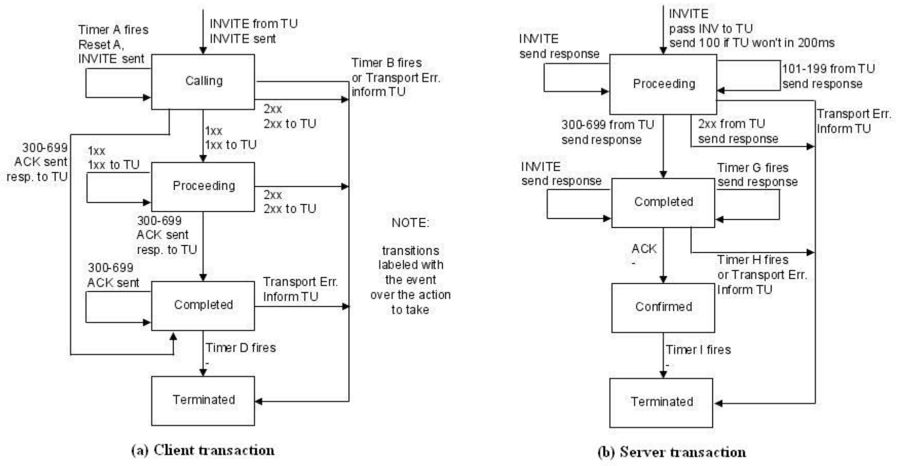


Fig. 2. INVITE client transaction (a) and INVITE server transaction (b)[1]

transaction enters its initial state, *Calling*[1]. The INVITE request must be passed by the client transaction to SIP transport layer to be sent to the server side.

Once the *Calling* state is entered, *Timer B* is started for all transports (reliable or unreliable). For an unreliable transport medium, *Timer A* is also started, to control retransmissions of INVITE requests. While *Calling*, if the client transaction receives a *1xx* response (Table 1), it enters the *Proceeding* state. If a *Transport Err* (Error) occurs or *Timer B* expires, the client transaction moves to the *Terminated* state and informs its TU immediately. A *Transport Err* is indicated by SIP transport layer when a request cannot be sent to the underlying transport medium, which is generally caused by fatal ICMP errors in UDP or connection failures in TCP.

When in its *Proceeding* state, the client transaction may receive any number of provisional responses (*1xx*) before receiving a final response (*2xx* or *300-699*). While *Calling* or *Proceeding*, the reception of a final response by the client transaction will change its state to *Completed* or *Terminated*, depending on the type of the final response. If a *2xx* is received (indicating that the INVITE request is accepted by the server), the client transaction must enter its *Terminated* state, without sending any ACKs. If a *300-699* response is received (the call establishment was not successful), an ACK is generated by the client transaction and passed to SIP transport layer to be sent to the server side, and the client transaction moves to the *Completed* state. The reason for sending an ACK is to cease the retransmission of *300-699* responses by the server. All responses received by the client transaction when it is *Calling* or *Proceeding* must be passed to the TU except for retransmitted responses.

[1] Fig. 2(a) shows that an INVITE is sent before the *Calling* state is entered. However, based on the text description in RFC 3261, the *Calling* state is entered before the INVITE is sent to SIP transport layer. Additionally, the client transaction has to be in a state when sending a message. So we follow the text description in the RFC.

When the *Completed* state is entered, *Timer D* is started with a value of at least 32 seconds for an unreliable transport medium and zero seconds for a reliable medium. This timer is used to allow client transaction to absorb retransmitted *300-699* responses, and to re-pass ACKs to SIP transport layer. When the client transaction is in its *Completed* state, if a *Transport Err* occurs, it changes to the *Terminated* state and informs the TU of the failure of sending an ACK. If *Timer D* fires, the client transaction must also move to the *Terminated* state. Once the *Terminated* state is entered, the client transaction must be destroyed by its TU immediately.

INVITE Server Transaction. The server transaction is established by its TU at the server side when the TU receives a new INVITE request from the client side (Fig. 2 (b)). Once the server transaction is constructed, it enters the *Proceeding* state and sends a *100 Trying* response if the TU does not generate a response within 200 milliseconds[2], to cease retransmissions of INVITE requests by the client transaction.

While *Proceeding*, the server transaction can pass any provisional responses from its TU to SIP transport layer without changing state. If the response from its TU is a *2xx* response, or if it is informed of a transport error by SIP transport layer, the server transaction moves to its *Terminated* state. Otherwise, it will enter the *Completed* state, waiting for an acknowledgement from the client for the *300-699* response that was sent while in the *Proceeding* state. A transport error in the server transaction is indicated by SIP transport layer when a response cannot be sent to the underlying transport medium.

If a retransmitted INVITE is received in the *Proceeding* state, the most recent provisional response from TU must be passed to SIP transport layer for retransmission. If a retransmitted INVITE is received in the *Completed* state, the server transaction should pass a *300-699* response to SIP transport layer.

Once the *Completed* state is entered, *Timer H* is started. It sets the maximum time during which the server transaction can retransmit *300-699* responses. If the transport is unreliable, *Timer G* should also be started to control the time for each retransmission. If *Timer H* fires or a *Transport Err* occurs before an ACK is received by the server transaction, the transaction moves to its *Terminated* state. If an ACK is received before *Timer H* fires, the server transaction moves to its *Confirmed* state and *Timer I* is started with a delay of 5 seconds for an unreliable transport, and zero seconds for a reliable transport. *Timer I* is used to absorb additional ACKs triggered by the retransmission of *300-699* responses. When *Timer I* fires, *Terminated* state is entered. The server transaction is destroyed once it enters the *Terminated* state.

[2] Fig. 2(b) shows that if TU does not generate a response within 200ms, the server transaction must send a *100 Trying* response before the *Proceeding* state is entered. In fact, the INVITE server transaction cannot send any message before it is created (i.e. before entering the *Proceeding* state). This is inconsistent with the text description in the RFC 3261. So we follow the text description in the RFC.

3 Modelling and Analysis of the SIP INVITE Transaction

3.1 Modelling Assumptions

According to [1], SIP INVITE transaction can operate over a reliable (e.g. TCP) or an unreliable (e.g. UDP) transport medium. In this paper we assume a reliable transport medium is used, because firstly TCP is made mandatory in [1] for larger messages; secondly we use an incremental approach, checking whether the protocol works correctly over a perfect medium before checking its operations over an imperfect one. Furthermore, a lossy medium may mask some problems that will only be detected with a perfect medium.

Referring to Fig. 2(b), once the INVITE server transaction is created by its TU, if the transaction knows that the TU will not generate a response within 200 ms, it must generate and send a 100 Trying response. We assume that the INVITE server transaction does not know that the TU will generate a response within 200 ms after the server transaction is created, i.e. the server transaction must generate and send a 100 Trying response after it is created.

We also assume that a request message carries only a method (such as INVITE or ACK) and a response message carries only a status code (such as 200 OK), without including any header fields (such as Call-ID) or message bodies, as they are not related to the functional properties to be investigated.

3.2 INVITE Transaction State Machines with Reliable Medium

In Section 2 we have found some inconsistencies between the INVITE transaction state machines (Fig. 2) and the narrative descriptions given in Sections 17.1.1 and 17.2.1 of [1]. In this section, we present a revised version of the two state machines obtained by considering the modelling assumptions stated in the previous section and by eliminating the inconsistencies found in Section 2. We call these state machines "the INVITE transaction state machines with reliable transport medium" (Fig. 3), and the state machines provided in [1] (Fig. 2) "the original state machines".

Referring to Fig. 2, a number of timers are defined in the original state machines to deal with message loss or processing/transmission delays. When the transport medium is reliable, the requests (INVITE and ACK) and the final responses (non-1xx) in the original INVITE server transaction are sent only once [1]. As a result, Timer A of the INVITE client transaction and Timer G of the INVITE server transaction are not applied (see Fig 3). Additionally, as Timer I of the original INVITE server transaction is set to fire in zero seconds for a reliable transport medium, the Confirmed state and Timer I are not considered. Hence, after an ACK for 300-699 response is received, the INVITE server transaction is terminated immediately (Fig. 3(b)).

Because we have assumed that the server transaction must generate and send a 100 Trying response after it is created, we remove the if clause "if TU won't in 200ms" from the top of the original INVITE server transaction state machine. Additionally as noted in Section 2 (footnote 2), the INVITE server transaction

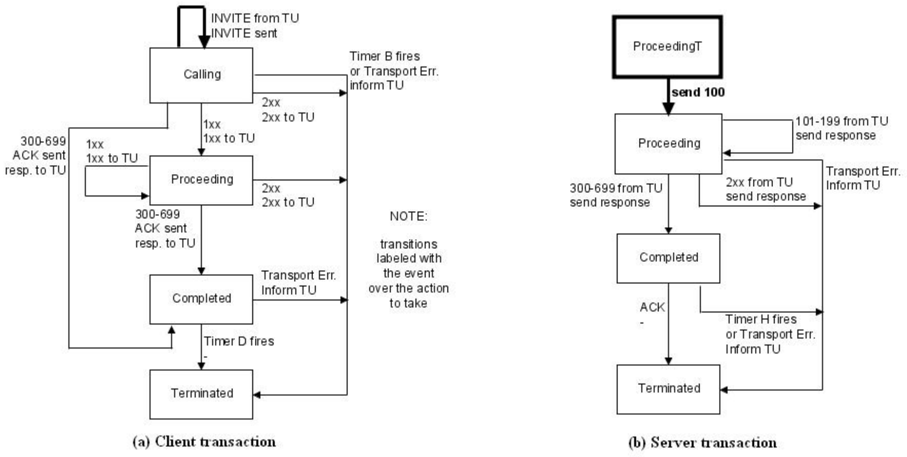


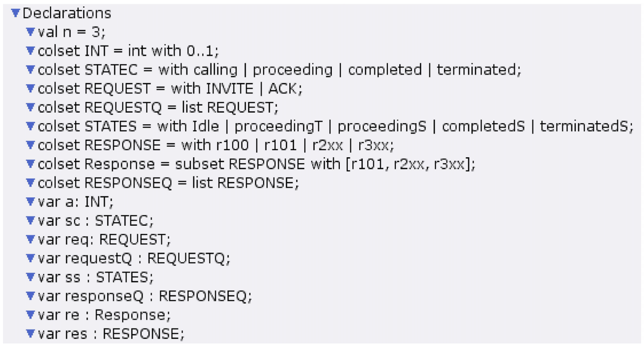
Fig. 3. INVITE client transaction (a) and server transaction (b) with reliable medium

can not receive or send any messages before it is created (i.e. before the Proceeding state is entered). So we firstly remove the action "pass INV to TU" from the top of the original state machine. Then to specify the state of the server transaction when it has just been created by its TU, we add a new state, *ProceedingT*. In the ProceedingT state, the only action to be carried out by the server transaction is to generate and send a 100 Trying response (see Fig. 3(b)).

A further modification is made to the original state machine for the INVITE client transaction based on the inconsistency mentioned in Section 2 (footnote 1). According to the narrative description provided in Section 17.1.1 of [1], the INVITE client transaction must firstly enter its Calling state to pass an INVITE received from its TU to SIP transport layer. However, the original state machine (Fig. 2(a)) shows that sending an INVITE by the client transaction can occur before it enters the Calling state (i.e. before the transaction is created), which is impossible. Therefore, we modified the input arc at the top of the original client state machine (refer to Fig. 2(a)) so that an INVITE request can be received from its TU and passed to the SIP transport layer by the INVITE client transaction only when the transaction is in its Calling state (see Fig. 3). Note that the event and action that label this input arc (to the Calling state) can not occur more than once due to the reason that the TU only passes one INVITE request to an INVITE client transaction [1].

3.3 CPN Model of the INVITE Transaction

The CPN model for the INVITE transaction is shown in Fig. 4 (declarations) and Fig. 5 (the CPN). This model is based on the state machines shown in Fig. 3. In the following, names of places, transitions, and variables of the CPN model are written in `typewriter` style. To distinguish a server transaction's state from a client transaction's state with the same name, a capitalised S is



```
▼Declarations
  ▼val n = 3;
  ▼colset INT = int with 0..1;
  ▼colset STATEC = with calling | proceeding | completed | terminated;
  ▼colset REQUEST = with INVITE | ACK;
  ▼colset REQUESTQ = list REQUEST;
  ▼colset STATES = with Idle | proceedingT | proceedingS | completedS | terminatedS;
  ▼colset RESPONSE = with r100 | r101 | r2xx | r3xx;
  ▼colset Response = subset RESPONSE with [r101, r2xx, r3xx];
  ▼colset RESPONSEQ = list RESPONSE;
  ▼var a: INT;
  ▼var sc : STATEC;
  ▼var req: REQUEST;
  ▼var requestQ : REQUESTQ;
  ▼var ss : STATES;
  ▼var responseQ : RESPONSEQ;
  ▼var re : Response;
  ▼var res : RESPONSE;
```

Fig. 4. Declarations of the CPN model of the INVITE transaction

appended to the name of the state of the server transaction (except for the proceedingT state). For example, `proceedingS` represents the Proceeding state of the server transaction while `proceeding` represents the Proceeding state of the client transaction. SIP response messages (Table 1) are named as follows: `r100` represents a 100 Trying response; `r101` is for a provisional response with a status code between 101 and 199; `r2xx` for a response with a status code between 200 and 299; and `r3xx` for a response with a status code between 300 and 699.

Declarations. Referring to Fig. 4, a constant, `n`, is defined to represent the maximum length of the queue in place `Responses` (Fig. 5). Four colour sets, `INT`, `STATEC`, `REQUEST`, and `REQUESTQ`, are declared for modelling the client transaction. `INT` is the set of integers between 0 and 1, typing place `INVITE Sent` where the number of INVITE requests that have been sent is counted. `STATEC` (typing place `Client`) models all the possible states of the INVITE client transaction. `REQUEST` models the two SIP methods specified for the INVITE transaction, INVITE and ACK. `REQUESTQ` is a list of `REQUEST` (typing place `Requests`). To model the server transaction, colour sets `STATES`, `RESPONSE`, `Response`, and `RESPONSEQ` are declared. `STATES` is used to type place `Server`. It defines all the possible states of the server transaction, and a temporary state, `Idle` (modelling the existence of the INVITE server transaction). `RESPONSE` models the four different categories of responses from the server side, and `Response` is a subset of `RESPONSE`, used in the inscriptions of the arcs associated with transition `Send Response` (see Fig. 5). This subset is used for modelling that any response except `r100` can be sent when the server is in its `proceedingS` state, which can be implemented using the variable `re` that runs over the subset. `RESPONSEQ` is a list of responses sent by the server transaction, and it is used to type place `Responses`, to model a First-In-First-Out queue. Variable `a` is of type `INT`, and `sc` and `ss` can take values from the colour sets `STATEC` and `STATES` respectively. Variables `req` and `res` are of types `REQUEST` and `RESPONSE` respectively. For dealing with the lists that store requests and responses, we declare variables `requestQ` of type `REQUESTQ` and `responseQ` of type `RESPONSEQ`.

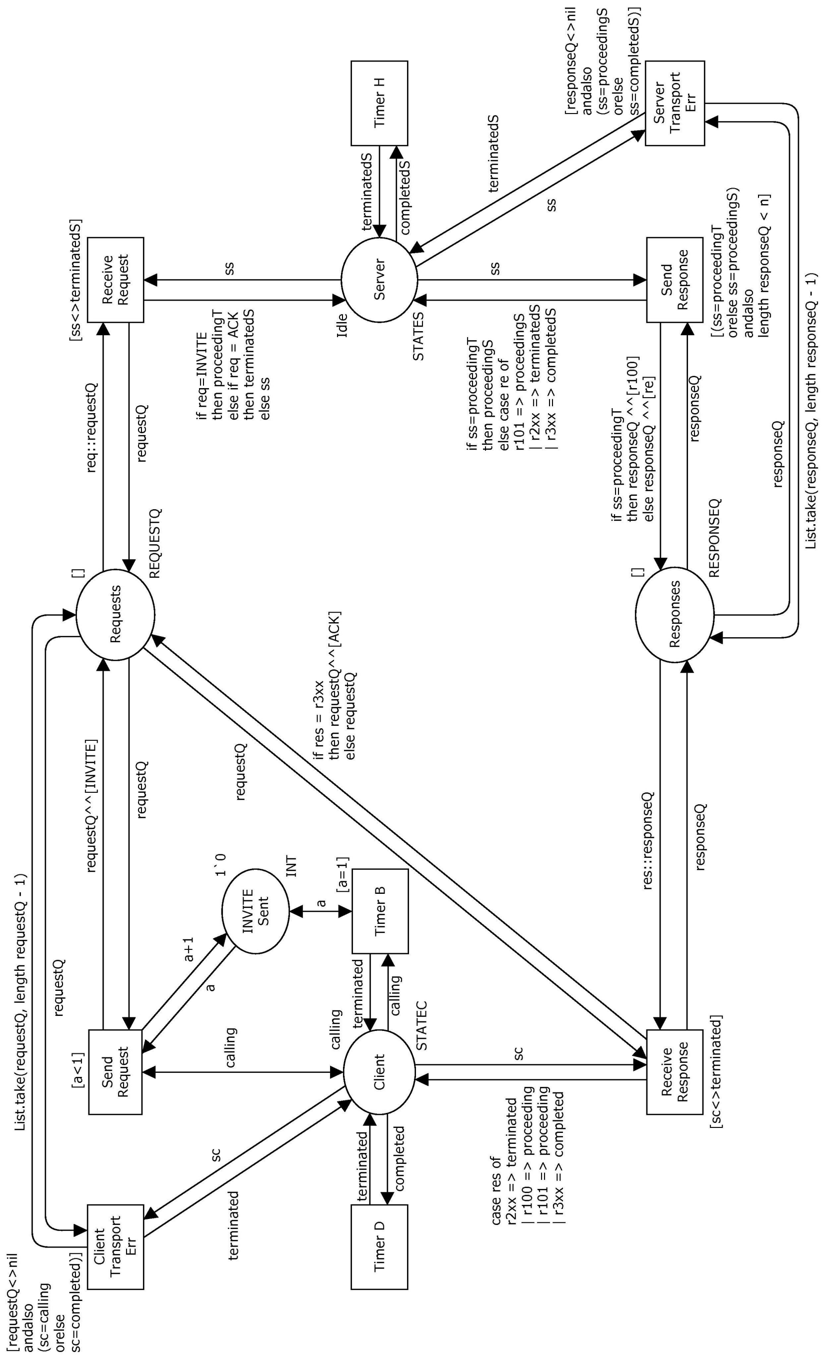

Timer H
Server Transport Err
[responseQ<>nil andalso (ss=proceedingS orelse ss=completedS)]
terminatedS
completedS
terminatedS
ss
Receive Request
[ss<>terminatedS]
Server
Idle
STATES
ss
ss
Send Response
[(ss=proceedingT orelse ss=proceedingS) andalso length responseQ < n]
if req=INVITE then proceedingT else if req = ACK then terminatedS else ss
if ss=proceedingT then proceedingS else case re of r101 => proceedingS | r2xx => terminatedS | r3xx => completedS
req::requestQ
requestQ
if ss=proceedingT then responseQ ^^[r100] else responseQ ^^[re]
responseQ
responseQ
List.take(responseQ, length responseQ - 1)
[]
Requests
REQUESTQ
[]
Responses
RESPONSEQ
if res = r3xx then requestQ^^[ACK] else requestQ
requestQ
List.take(requestQ, length requestQ - 1)
requestQ
requestQ^^[INVITE]
requestQ
res::responseQ
responseQ
1`0
INVITE Sent
INT
a+1
a
a
[a=1]
Timer B
Send Request
[a<1]
calling
calling
terminated
calling
STATEC
Client
sc
Receive Response
[sc<>terminated]
case res of r2xx => terminated | r100 => proceeding | r101 => proceeding | r3xx => completed
sc
terminated
terminated
completed
Client Transport Err
[requestQ<>nil andalso (sc=calling orelse sc=completed)]
Timer D

Fig. 5. CPN model of the INVITE transaction

INVITE Client Transaction. The left part of the CPN model (Fig. 5), including places `Client` and `INVITE Sent`, and the transitions connected to them, models the client transaction state machine with reliable medium (Fig. 3(a)).

States of the client transaction are represented by place `Client` (typed with colour set `STATEC`). The initial state of `Client` is `calling`, indicating that the INVITE client transaction has been initiated by its TU, and an INVITE request has been passed from its TU to the client transaction for transmission.

Five transitions are associated with `Client`, to model operations of the INVITE client transaction. `Send Request` models how the transaction passes an INVITE request received from its TU to SIP transport layer. It is enabled only if there is a `calling` in `Client` and no INVITE request has been passed to SIP transport layer (i.e. `INVITE Sent` contains an integer with value 0). `Receive Response` models how the client transaction receives responses and sends ACKs. It is enabled when a response is received (i.e. removed from the head of the queue in place `Responses`) and the `Client` is not `terminated`. If the client transaction receives a 300-699 response (`r3xx`), an ACK is passed to SIP transport layer, and `Client` moves to its `completed` state. If the received response is `r100`, `r101` or `r2xx`, no ACK is sent; when the response is `r100` or `r101`, `Client` moves to `proceeding`; and when the response is `r2xx`, the `Client` moves to `terminated`.

Timer B is modelled by transition `Timer B`. Since our focus is on the functional correctness of SIP rather than its performance properties such as session delay, values of timers are not modelled. To model that Timer B can not be started before an INVITE request is sent, place `INVITE Sent` is set to an input place of `Timer B`. `Timer B` is enabled only when an integer with value 1 is in `INVITE Sent` (see the guard `[a=1]`) and the `Client` is `calling`. The initial marking of `INVITE Sent` is 0 (no INVITE request has been sent to SIP transport layer), when `Send Request` is fired (i.e. an INVITE request has been sent), the integer value is incremented by 1.

Timer D (Fig. 3(a)) sets the maximum time for which the client transaction can stay in its Completed state to wait for retransmitted 300-699 responses from the server transaction. Since there are no retransmissions when the transport medium is reliable, according to [1], Timer D is set to fire in zero seconds in this case. Once it fires, the client transaction enters its Terminated state. This seems to indicate that the client transaction would enter the Terminated state immediately after it is in the Completed state, and nothing else can happen in between the two states. Thus we might be able to fold the two states into one and not to consider Timer D when the transport is reliable. However, from Fig. 3, a Transport Err can occur when the client transaction is in its Completed state, and the Completed state is entered after the client transaction has passed an ACK to SIP transport layer. The transport layer may report an error immediately when it receives the ACK, thus a Transport Err occurs at the transaction layer. According to Fig. 3, the transaction layer needs to inform its TU of this error when it is in the Completed state. From this perspective, we can not fold the Completed and Terminated states. Therefore, we create a transition, `Timer`

D. It is enabled once there is a completed in Client, and its occurrence brings Client to terminated.

Transition Client Transport Err (modelling a Transport Err at the transaction layer) is enabled when the Client is completed, and its occurrence also brings Client to terminated. When an error is reported by SIP transport layer, the ACK that has been passed to it from the transaction layer will not be sent to the server side, so when Client Transport Err occurs, the ACK that has just been put in the queue in Requests is destroyed (see the inscription of the arc from Client Transport Err to Requests). From Fig. 3(a), a Transport Err can occur when the client transaction is Calling, so Client Transport Err can be enabled as well when a calling is in Client (see the guard of Client Transport Err).

INVITE Server Transaction. Referring to the right part of Fig. 5, place Server and the four transitions connected to it model the INVITE server transaction specified in Fig. 3(b). Place Server models the states of the transaction, proceedingT, proceedingS, completedS, terminatedS. Idle (see Fig. 4) is not a state of the server transaction, it is the initial marking of Server, indicating that it is ready for the TU to create a server transaction once the TU receives an INVITE request. Transition Receive Request models the reception of a request (INVITE or ACK). While there is an Idle in Server, if the request received is an INVITE, a proceedingT is created in Server, modelling that the server transaction is created (by the TU) and it is now in its proceedingT state (in this case and only in this case, transition Receive Request models the operation of the TU instead of the server transaction of receiving an INVITE request from the client side). Once the Server enters its proceedingT state, Send Response is enabled, thus r100 can be placed into the queue in Responses, changing the state of the Server to proceedingS. In the proceedingS state, Send Response is again enabled. When it occurs, either a provisional response r101 or a final response (r2xx or r3xx) is appended to the queue in Responses, and a proceedingS (if a r101 is put in the queue), completedS (for r3xx) or terminatedS (for r2xx) is put in Server (refer to the else clauses of the inscriptions of the outgoing arcs of Send Response). While the Server is completedS, if an ACK is received, the occurrence of Receive Request will generate a terminatedS in the Server. The guard of Receive Request models that the server transaction can not receive any requests after it enters the Terminated state because a server transaction is destroyed immediately after the Terminated state [1].

Only one timer, Timer H, is used by the INVITE server transaction (Fig. 3(b)) when the transport medium is reliable, to control the transmission and retransmission of 300-699 responses by the INVITE server transaction. It is modelled by transition Timer H, which is enabled when the Server is completedS. When it fires, Server moves to its terminatedS state, indicating that an ACK corresponding to a 300-699 response is never received by the server transaction before Timer H has expired. Server Transport Err models a transport error occurring at the server side. It is enabled after a response has been sent to SIP transport layer (i.e. when the server transaction is proceedingS or completedS). When it fires, the response that has just been put into the queue in `Responses` is removed (see the inscription of the arcs from `Server Transport Err` to `Responses`, the `List.take` function returns the remained responses of list `responseQ`) and a `terminatedS` is put into `Server`.

The Underlying Transport Medium. The middle part of the CPN model (i.e. places `Requests` and `Responses` of Fig. 5) models SIP transport layer and the underlying transport medium (see Fig. 1). `Requests` models the transmission of requests from the client side to the server side; whereas `Responses` models the transmission of responses in the reverse direction. The maximum number of provisional responses (i.e. `r101`) that can be sent from the server transaction is not given in [1], so there may be an arbitrarily large number of `r101` to be put into the queue in `Responses`. We use a positive integer parameter, `n` (Fig. 4) to represent the maximum length of the queue in `Responses` and a guard for transition `Send Response` (`[length responseQ < n]`) to limit the maximum length of the queue in `Responses`.

3.4 State Space Analysis of the INVITE Transaction CPN Model

In this section, we firstly define the desired properties for the INVITE transaction, then we analyse the state space of the CPN model described in the previous section. In order to avoid state explosion problem with state space analysis, we use 3 as the maximum length of the queue in place `Responses` (i.e. `n=3`).

The functional properties that we are interested in include *absence of deadlocks* and *absence of livelocks*. According to [3] a protocol can fail if any of the two properties are not satisfied. We also expect that the INVITE transaction has *no dead code* (and action that is specified but never executed). A deadlock is an undesired dead marking in the state space of a CPN model, and a marking is dead if no transitions are enabled in it [11]. For the INVITE transaction, there is only one desired dead marking, representing the desirable terminal state of the INVITE transaction. In this state both the client and the server transactions are in their Terminated state (see Fig. 3), and no messages remain in the communication channel. Any other dead marking is thus undesirable, i.e. a deadlock. A livelock is a cycle of the state space that once entered, can never be left, and within which no progress is made with respect to the purpose of the system.

To analyse the desired properties of the INVITE transaction, we firstly check the state space report generated by CPN Tools [12]. The report shows that a full state space with 52 nodes and 103 arcs is generated. There are 3 nodes fewer in the Strongly Connected Components (SCC) graph than in the state space, which means that the state space contains cycles (which needs to be further investigated to see if they are livelocks). We also found ten dead markings in the state space, which are nodes 5, 19, 25, 28, 29, 30, 36, 45, 46 and 50 (Table 2). Additionally, there are no dead transitions in the state space, so the INVITE transaction has no dead code.

From Table 2, we can see that node 25 is a desirable dead marking. The requests and responses have been sent and received. Both the client and server

Table 2. List of dead markings

Node	Client	Requests	Server	Responses	INVITE Sent
5	`terminated`	`[]`	`Idle`	`[]`	`1‘1`
19	`terminated`	`[]`	`terminatedS`	`[r100, r2xx]`	`1‘1`
25	`terminated`	`[]`	`terminatedS`	`[]`	`1‘1`
28	`proceeding`	`[]`	`terminatedS`	`[]`	`1‘1`
29	`terminated`	`[]`	`terminatedS`	`[r100, r3xx]`	`1‘1`
30	`terminated`	`[]`	`terminatedS`	`[r100]`	`1‘1`
36	`terminated`	`[]`	`terminatedS`	`[r100, r101, r2xx]`	`1‘1`
45	`terminated`	`[]`	`terminatedS`	`[r100, r101, r3xx]`	`1‘1`
46	`terminated`	`[]`	`terminatedS`	`[r100, r101]`	`1‘1`
50	`terminated`	`[ACK]`	`terminatedS`	`[]`	`1‘1`

transactions are in their Terminated states. Moreover, no messages remain in the channel, i.e. places `Requests` and `Responses` each has an empty list. At node 5 the server's state is `Idle`, which is different from other dead markings (i.e `terminatedS`). To find out whether this node is an undesirable dead marking, we use CPN Tools' query **ArcsInPath (1, 5)** to generate the path to this dead marking (Fig. 6). At the initial state, node 1, the client transaction sends an INVITE request to SIP transport layer. However, a transport error occurs at node 2 and no request is sent to the server side. Hence, the client transaction is terminated and no corresponding INVITE server transaction is constructed at the server side (node 5). This behaviour is expected [1]. Furthermore, for each of the nodes 19, 29, 30, 36, 45, 46 and 50 in Table 2, there are still messages remaining in the communication channel after both the client and server transactions have been terminated. These dead markings are caused by either transport error or timeout. Once a transport error or timeout occurs, the transaction will be destroyed. Therefore a message left in the channel will find no matching transaction. However, these dead markings are acceptable, because if the message is a response, according to [1], the response must be silently discarded by SIP transport layer of the client. If the message is a request (i.e. node 50), according to [1], the request (i.e. ACK) will be passed to the TU of the destroyed server transaction to be handled.

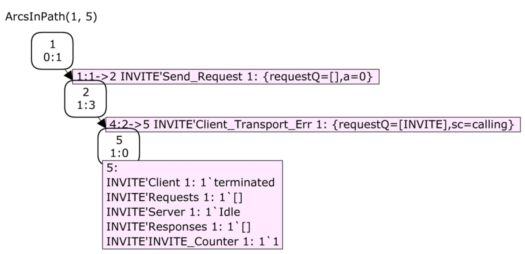

ArcsInPath(1, 5)
1
0:1
1:1->2 INVITE'Send_Request 1: {requestQ=[],a=0}
2
1:3
4:2->5 INVITE'Client_Transport_Err 1: {requestQ=[INVITE],sc=calling}
5
1:0
5:
INVITE'Client 1: 1`terminated
INVITE'Requests 1: 1`[]
INVITE'Server 1: 1`Idle
INVITE'Responses 1: 1`[]
INVITE'INVITE_Counter 1: 1`1

Fig. 6. The path from Node 1 to Node 5

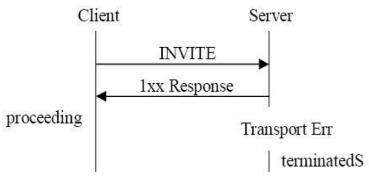


Fig. 7. Scenario of undesired behaviour

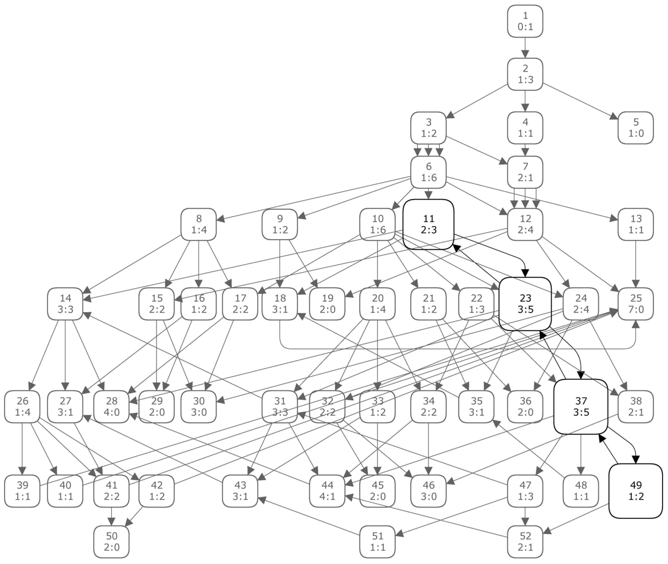


Fig. 8. State Space of the INVITE transaction CPN model

However, node 28 is an undesired dead marking (deadlock). Referring to Table 2, with node 28, the state of the client transaction is `proceeding`, while the server transaction is `terminatedS` due to a transport error (Fig. 7). This behaviour is not expected since in this case the server transaction has been destroyed (i.e. no responses can be received by the client transaction). No timer is given in [1] to specify the maximum time the client transaction can wait for a final response when it is in the `proceeding` state. Thus, when a transport error occurs at the server side, the client transaction will have no way to come out from the `proceeding` state.

To check if there are livelocks, we draw the state space of the INVITE transaction (Fig. 8). As indicated by the difference between the size of the state space and the size of the SCC graph, there are cycles in the state space: the cycles between nodes 11, 23, 37 and 49. However, because from any of the four nodes, the transaction can move to a state which is not within a cycle, these cycles are not livelocks. Therefore, the INVITE transaction is free of livelocks.

4 The Revised INVITE Transaction and Its Verification

4.1 Changes to the INVITE Transaction

In the previous section, we have found that the INVITE transaction can reach a deadlock where the server transaction has terminated but the client transaction is still in its Proceeding state. After discovering this using state space analysis of CPNs, we noticed that this deadlock had been discussed and reported by SIP implementers [13] . However, so far there is no solution published for fixing this bug. In this section, we propose a modification to the INVITE transaction to eliminate the deadlock.

In [1], it is recommended that, at the TU level, the TU at the client side should destroy the original INVITE client transaction, if the TU does not receive a final response within 64xT1 seconds (T1 = 500ms) after an INVITE request is passed to the transaction layer. So for an application that implements this recommendation, the INVITE client transaction will be destroyed eventually if the transaction has reached the deadlock. However, with applications that do not implement this recommendation, a caller will not receive any indications from the system, thus may wait for a long time (listening to the ringing tone) before giving up (i.e. hanging up the phone). Therefore we believe that it is necessary to solve this problem at the transaction layer, i.e. to force the client transaction to reach its Terminated state, instead of staying in the Proceeding state, so that the TU can destroy the transaction and avoid an unnecessarily long wait.

From Fig. 3(a) we see that the client transaction can go from the Proceeding state to the Completed state then the Terminated state only if it receives a final response (300-699 or 2xx) from the server transaction. However, after having reached the deadlock, no responses can be received by the client transaction since the server transaction has been terminated. Therefore, in this case, to bring the client transaction to its Terminated state, we need to use an event that occurs at the client side to trigger this state change, i.e. a timer for the Proceeding state. Referring to Fig. 9, before the client transaction enters the Proceeding state, it now needs to reset Timer B (i.e. restarts it with value 64xT1 ms). Then in the Proceeding state, once Timer B expires, the INVITE client transaction notifies the TU about the timeout and moves to the Terminated state.

4.2 The Revised INVITE Transaction CPN and Its Analysis

Fig. 10 shows the CPN model for the revised INVITE transaction. It is obtained from the CPN in Fig. 5 by modifying the arc inscription and the guard of transition `Timer B`. The arc inscription from place `Client` to `Timer B` has been changed from `calling` to a variable, `sc`, of colour set `STATEC`. This variable can be bound to any value of `STATEC`, but `Timer B` should not be enabled in states other than `calling` or `proceeding`. Therefore, the guard of `Timer B`

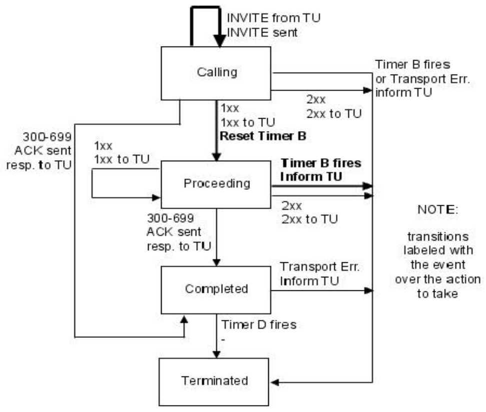


Fig. 9. Revised INVITE client transaction state machine

has also been changed. In Fig. 5, the guard `Timer B ([a=1])`, is used to set the condition that `Timer B` can not be enabled before an INVITE request has been sent in the `calling` state. Since `proceeding` is not associated with sending the INVITE request, the guard for the revised CPN model is modified to `[(sc=calling andalso a=1) orelse sc=proceeding]`.

The same as with analysing the original CPN model shown in Fig. 5, we use 3 as the maximum length of the queue in place `Responses` (i.e. `n=3`) to avoid state explosion. The state space of the CPN model is then generated. The state

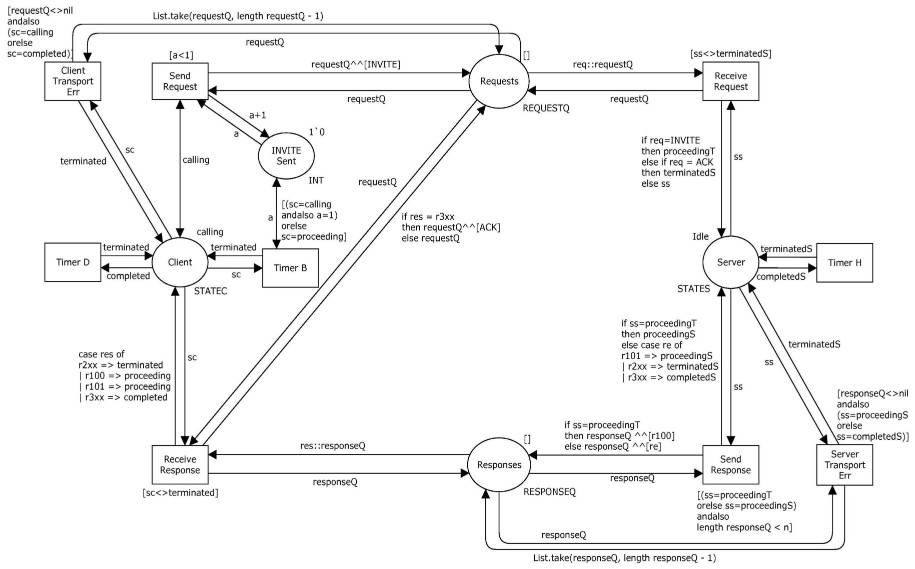

[requestQ<>nil andalso (sc=calling orelse sc=completed)]
List.take(requestQ, length requestQ - 1)
requestQ
[a<1]
Client Transport Err
Send Request
requestQ^^[INVITE]
Requests
[]
req::requestQ
[ss<>terminatedS]
Receive Request
requestQ
REQUESTQ
requestQ
a+1
a
1`0
INVITE Sent
INT
terminated
sc
calling
if req=INVITE then proceedingT else if req = ACK then terminatedS else ss
ss
requestQ
[(sc=calling andalso a=1) orelse sc=proceeding]
a
if res = r3xx then requestQ^^[ACK] else requestQ
calling
Idle
Timer D
terminated
completed
Client
terminated
sc
Timer B
STATEC
Server
terminatedS
completedS
Timer H
STATES
case res of r2xx => terminated | r100 => proceeding | r101 => proceeding | r3xx => completed
sc
if ss=proceedingT then proceedingS else case re of r101 => proceedingS | r2xx => terminatedS | r3xx => completedS
terminatedS
ss
ss
[responseQ<>nil andalso (ss=proceedingS orelse ss=completedS)]
if ss=proceedingT then responseQ ^^[r100] else responseQ ^^[re]
res::responseQ
[]
Receive Response
Responses
Send Response
Server Transport Err
responseQ
responseQ
[sc<>terminated]
RESPONSEQ
[(ss=proceedingT orelse ss=proceedingS) andalso length responseQ < n]
responseQ
List.take(responseQ, length responseQ - 1)

Fig. 10. CPN model of the revised INVITE transaction

space report shows that there are more nodes and arcs generated for both the state space and SCC graph. However, the differences between the state space and SCC graph, i.e. 3 nodes and 6 arcs, have remained the same. Additionally, the report shows that there are no dead transitions in the state space of the INVITE transaction, so it has no dead code.

All the dead markings are shown in Table 3. We see that the deadlock in the state space of the original model (node 28 of Table 2) has been removed. However, the state space of the revised CPN model has more dead markings (i.e. nodes 59, 60, 64, 66 and 67). This is because in the Proceeding state of the INVITE client transaction (refer to Fig. 10), when `Timer B` occurs, the `Client` moves to `terminated` before the responses (i.e. `r101`, `r2xx` or `r3xx`) have been received (queuing in `Responses`). Previously, we only had these responses queuing in the `Responses` when `Client` was marked by `proceeding`. Since these dead markings (nodes 59, 60, 64, 66 and 67) have similar behaviour to nodes 19, 26, 31, 32, 33, 39, 47, 51, 52, 53, 58 (see Table 3), i.e. messages are left in the channel, and client and server transactions have each entered its Terminated state, they are all acceptable. Furthermore, from Table 3, nodes 5 and 26 were discovered in the state space of the original model, and they have already been identified as desirable dead markings of the INVITE transaction.

To check if there are livelocks in the revised INVITE transaction, we draw the state space (Fig. 11). There are cycles between nodes 11, 23, 40 and 56. However, none of them are livelocks because from each of the four nodes the transaction can move to a state that is not within a cycle. Therefore, the revised INVITE transaction has no livelock.

Table 3. List of dead markings of the revised CPN model

Node	Client	Requests	Server	Responses	INVITE Sent
5	terminated	[]	Idle	[]	1‘1
19	terminated	[]	terminatedS	[r100, r2xx]	1‘1
26	terminated	[]	terminatedS	[]	1‘1
31	terminated	[]	terminatedS	[r100, r3xx]	1‘1
32	terminated	[]	terminatedS	[r100]	1‘1
33	terminated	[]	terminatedS	[r2xx]	1‘1
39	terminated	[]	terminatedS	[r100, r101, r2xx]	1‘1
47	terminated	[]	terminatedS	[r3xx]	1‘1
51	terminated	[]	terminatedS	[r100, r101, r3xx]	1‘1
52	terminated	[]	terminatedS	[r100, r101]	1‘1
53	terminated	[]	terminatedS	[r100, r2xx]	1‘1
58	terminated	[ACK]	terminatedS	[]	1‘1
59	terminated	[]	terminatedS	[r101, r3xx]	1‘1
60	terminated	[]	terminatedS	[r101]	1‘1
64	terminated	[]	terminatedS	[r101, r101, r2xx]	1‘1
66	terminated	[]	terminatedS	[r101, r101, r3xx]	1‘1
67	terminated	[]	terminatedS	[r101, r101]	1‘1

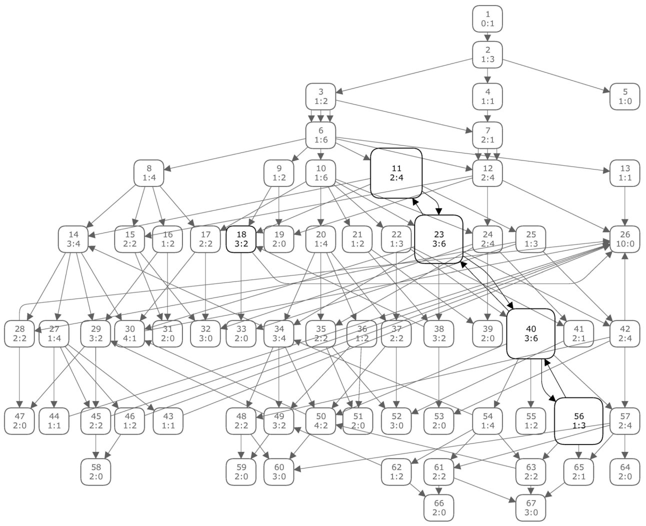


Fig. 11. Cycles in the state space of the revised INVITE transaction CPN model

5 Conclusions and Future Work

The INVITE transaction is one of the essential transactions of SIP. It has been used in conjunction with other protocols to establish sessions and provide communication services. Based on the state machines and narrative descriptions that are provided in [1], we have modelled and analysed the SIP INVITE transaction with reliable transport medium using CPNs. The contributions of the paper are summarised below.

– Refinement to the definition of the INVITE transaction. We have found some inconsistencies between the state machines and the narrative descriptions in [1]. Modifications have been proposed to the state machines to remove the inconsistencies. After omitting the states and actions which are defined for SIP over an unreliable transport medium only, we have obtained the state machines for the INVITE transaction over a reliable transport medium, and have created a CPN model for it, which provides a formal specification for the INVITE transaction.
– Verification of the INVITE transaction. By examining the state space of the CPN model, we have found that the INVITE transaction has no livelock or dead code. We have also found in the state space of the INVITE transaction a sequence of events that lead to the desirable terminal state, however, the INVITE transaction may terminate in an undesirable state, in which the INVITE client transaction is still in its Proceeding state.
– Revision to the definition of the INVITE transaction and its verification. To eliminate the undesirable behaviour, we have proposed a set of changes to the INVITE client transaction. Using state space analysis, we have found that the revised INVITE transaction has satisfied the desired properties.

In the future, we shall model and analyse the INVITE transaction with unreliable transport medium. We have noticed that, very recently, an Internet draft (work in progress) has been published by IETF, to propose updates to the INVITE transaction state machines [14]. The proposed updates have no impacts on the behaviour of the INVITE transaction when the transport medium is reliable, which means IETF may have not been aware of the incompleteness of [1] of the specification of the INVITE transaction. On the other hand, the proposed updates may have influence on the INVITE transaction when the transport medium is unreliable. Therefore, the other possible future work can include modelling and analysing the updated version of INVITE transaction proposed in the Internet Draft [14]. In this way, the correctness of the proposed updates given in the Internet Draft [14] can be checked and confirmed.

Acknowledgements. We would like to express our appreciation to the members of Computer Systems Engineering Centre, University of South Australia, for attending our presentation and providing useful comments on our research. Especially, we would like to thank Professor Jonathan Billington, Dr Guy Gallasch, and Dr Somsak Vanit-Anunchai for all their helpful suggestions and invaluable comments.

References

1. Rosenberg, J., et al.: RFC 3261: SIP: Session Initiation Protocol. Internet Engineering Task Force (2002), `http://www.faqs.org/rfcs/rfc3261.html`
2. Sparks, R.: SIP: basics and beyond. Queue 5(2), 22–33 (2007)
3. Holzmann, G.J.: Design and validation of computer protocols. Prentice Hall, Englewood Cliffs, New Jersey (1991)
4. Sidhu, D., Chung, A., Blumer, T.P.: Experience with formal methods in protocol development. In: ACM SIGCOMM Computer Communication Review, vol. 21(2), pp. 81–101. ACM, New York (1991)
5. Examples of Industrial Use of CP-nets, `http://www.daimi.au.dk/CPnets/intro/example_indu.html`
6. Billington, J., Gallasch, G.E., Han, B.: Lectures on Concurrency and Petri Nets: A Coloured Petri Net Approach to Protocol Verification. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 210–290. Springer, Heidelberg (2004)
7. Kristensen, L.M., Jørgensen, J.B., Jensen, K.: Application of Coloured Petri Nets in System Development. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 626–685. Springer, Heidelberg (2004)
8. Turner, K.J.: Modelling SIP Services Using CRESS. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529, pp. 162–177. Springer, Heidelberg (2002)
9. Gehlot, V., Hayrapetyan, A.: A CPN Model of a SIP-Based Dynamic Discovery Protocol for Webservices in a Mobile Environment. In: the 7th Workshop and Tutorial on Practical Use of CPNs and the CPN Tools, University of Aarhus, Denmark (2006)
10. Wan, H., Su, G., Ma, H.: SIP for Mobile Networks and Security Model. In: Wireless Communications, Networking and Mobile Computing, pp. 1809–1812. IEEE, Los Alamitos (2007)

11. Jensen, K., Kristensen, L., Wells, L.: Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. Int. J. on Software Tools for Technology Transfer (STTT) 9(3), 213–254 (2007)
12. Home Page of the CPN Tools, `http://wiki.daimi.au.dk/cpntools/cpntools.wiki`
13. Rosenberg, J.: Bug 706 - Clarify lack of a timer for exiting proceeding state, Bugzilla (2003), `http://bugs.sipit.net/show_bug.cgi?id=706`
14. Sparks, R.: draft-sparks-sip-invfix-00: Correct transaction handling for 200 responses to Session Initiation Protocol INVITE requests. Internet Engineering Task Force (2007), `http://tools.ietf.org/id/draft-sparks-sip-invfix-00.txt`