Modeling IETF Session Initiation Protocol and its services in SDL

Ken Y. Chan and Gregor v. Bochmann
School of Information Technology and Engineering (SITE),
University of Ottawa
P.O.Box 450, Stn. A, Ottawa, Ontario, Canada. K1N 6N5
{kchan,bochmann}@site.uottawa.ca

Abstract: This paper describes the formal approach to modeling IETF Session Initiation Protocol (SIP) and its services in SDL. The main objective is to discover the advantages and shortcomings of using a formal language such as SDL to model an IETF application signaling protocol like SIP. Evaluating the feasibility of using CASE tools such as Telelogic Tau in modeling a complex protocol like SIP is also the interest of this study. By creating an "Abstract User" interface, we discover the importance of use case analysis in specifying SIP services more precisely. In addition, the object-oriented extension in SDL-96 has been applied to some extent in the modeling process; we create an SDL framework that allow us to reuse and to add SIP services to the core protocol more easily by applying SDL type inheritance in our model. Furthermore, we discuss enhancements that may be made to the SDL language and Tau tools to improve the modeling experience of IETF protocols.

Keywords: SIP, Internet Telephony, UML, Use Case, SDL, MSC, Telelogic, Software Specification, Design Methodology

1 Introduction

With the increasing popularity of voice chats and Internet long distance calls, many companies foresee Internet telephony and voice applications would be a high revenue growth segment of the telecommunication market in the next few years. In the Internet telephony world, many researchers believe the IETF Session Initiation Protocol (SIP) is the emerging Voice over IP (VoIP) signaling protocol that can compete against ITU H.323 protocol suites [1,12]. SIP was originally designed as a simple call setup and handling protocol between user terminals. However, it has also been extended to address VoIP signaling at carrier-grade level and customer-premise telephony applications. Although researchers believe many new types of voice applications can be built with SIP, the flexibility of the SIP header and its unique user agent-proxy architecture would complicate the problem of feature interactions [4,14,15]. Thus, we have developed an SDL model of SIP and its sample services [1,2] to investigate the feature interaction problem. However, the emphasis of this paper is to discuss our experience of modeling a semantic rich application protocol like SIP using the SDL language and the CASE tool called Telelogic Tau [8]. The feature interaction problem is beyond the scope of this paper. Furthermore, we have used version 4.3 and 4.4 of Telelogic Tau in our modeling exercise. The next generation version of Telelogic Tau called "Tau G2" was still under development by Telelogic Inc. and was unavailable to us.

This paper is organized as follows: Section 2 presents an overview of the SIP protocol. Section 3 gives an overview of our design methodology which is further detailed in the subsequent sections. Section 4 explains the steps of applying use case analysis and deriving our use case scenarios. Section 5 describes the process of converting sample service call flows into Message Sequence Charts (MSC) which represent important test case scenarios. Section 6 details the structural design of the model. Section 7 illustrates the behavior specification of the SIP model. Section 8 explains the steps of validating and verifying the SDL model using an SDL CASE tool called Telelogic Tau. Section 9 discusses potential enhancements to and our experience using and SDL and Tau tools to model SIP. Finally, the paper ends with a conclusion and discussion of future work.

2 Overview of SIP

SIP (Session Initiation Protocol) is an application-layer multimedia control protocol standardized under IETF RFC 2543 [1]. Similar to most World Wide Web protocols, SIP has ASCII-based syntax that closely resembles HTTP. This protocol can establish, modify and terminate multimedia sessions including multimedia conferences, Internet telephony calls, and similar applications. SIP can also initiate multi-party calls using a multipoint control unit (MCU) or fully meshed interconnections instead of multicast. SIP offers five facets of establishing and terminating multimedia communications [1] but we focus only in call setup and call handling in this research because they are the central functions of any telephony services and offer interesting behaviors for formal modeling. Call setup is defined as the establishment of call parameters at both called and calling party. Call handling is defined as the ability to manage mid-call and third-party call control such as call transfer and call waiting, after the initial call has been setup and to manage termination of these calls.

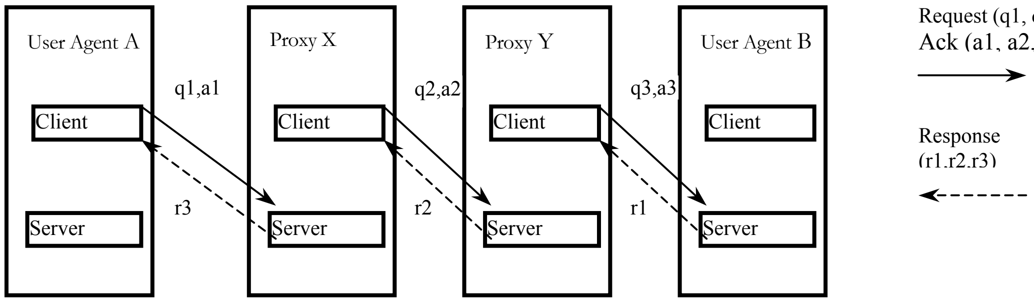


Figure 1: User Agent A calls B via Proxies X and Y

Under the SIP model, signaling parties communicate with each other through asynchronous messaging. Since SIP is a transport-independent signaling protocol, SIP messages can be transferred via UDP, TCP, or other transport protocols. There are three types of messages: request, response, and acknowledgement. In a basic two party call setup as illustrated in Figure 1, the initiator 'A' sends an "INVITE" request to the called party 'B' to establish a unicast multimedia session (e.g. a conference call would require multicast sessions). The "METHOD" field in a SIP request message indicates the action to be performed, and in this case, "INVITE" is the typical call setup and handling SIP request method. Then, the called party would generally reply with one or more appropriate response messages, and the caller would finally acknowledge the final "OK" response by sending an "ACK" message (which is considered as a special request message) to the called party. Upon reception of the "ACK" message, the voice media would be established between the two parties [1].

To cancel a previous request, the request initiator may send a "CANCEL" request message to the called party. To terminate an existing session, the member of the session who desires to terminate the session can send a "BYE" request message to the other party. In an N-party call session, the initiator of a SIP request does not necessarily have to be a member of the session to which it is inviting. Media and participants can be added to or removed from an existing session by sending "INVITE" message(s) with new parameters in the SDP [20] after the initial call setup. To register the location of the session member, the member can send a "REGISTER" message to a SIP registrar that serves as a lookup directory. There are other SIP methods that have been defined, for example, "INFO". Additional SIP methods can also be added to the standard [1]; however, these are beyond the scope of this paper.

In SIP terminology, a call consists of all participants in a conference invited by a common source. A SIP call is identified by a globally unique call-id. Thus, if a user is, for example, invited to the same multicast session by several people, each of these invitations will be a unique call. However, after the multipoint call has been established, the logical connection between two participants is a call leg, which is identified by the combination of "Call-ID", "To", and "From" header fields. A point-to-point Internet telephony conversation maps into a single SIP call. In a call-in conference using a MCU, each participant uses a separate call to invite himself to the MCU. The sender of a request message or the receiver of a response message is known as the client, whereas the receiver of a request message or the sender of a response message is known as a server. A user agent (UA) is a logical entity, which contains both a user agent client and user agent server, and acts on behalf of an end-user for the duration of a call. A proxy is an intermediary program that acts as both a server and a client for the purpose of making requests on behalf of other clients. A proxy may process or interpret requests internally or by passing them on, possibly after translation, to other servers (Figure 1). A reader should not confuse the SIP client and server with the initiator (caller or originator) and callee.

A user agent or a proxy is said to contain both client and server and can act as either a client or a server, but not both simultaneously in the same transaction [1].

What makes SIP interesting and different from other VoIP protocols are the message header and body. Like HTTP, a SIP message, whether it is a request, response, or acknowledgement message, consists of a header and a body. A sample "INVITE" request message body is shown in Figure 2 :

```
INVITE sip:ken@ee.uottawa.ca SIP/2.0
Via: SIP/2.0/UDP gtwy1.uottawa.ca;branch=8348
;maddr=137.128.16.254;ttl=16
Via: SIP/2.0/UDP gtwy.ee.uottawa.ca
Record-Route: gtwy.ee.uottawa.ca
From: Bill Gate <sip:bill@Microsoft.com>
To: Ken Chan <sip:ken@uottawa.ca>
Contact: Ken Chan <sip:ken@site.uottawa.ca>
Call-ID: 56258002189@site.uottawa.ca
CSeq: 1 INVITE
Subject: SIP will be discussed, too
Content-Type: application/sdp
Content-Length: 187

v=0
o=bill 53655765 2353687637 IN IP4 224.116.3.4
s=RTP Audio
i=Discussion of .Net
c=IN IP4 224.2.0.1/127
t=0 0
m=audio 3456 RTP/AVP 0
```

```
OK 200 SIP/2.0
Via:SIP/2.0/UDP gtwy1.uottawa.ca;branch=8348
;maddr=137.128.16.254;ttl=16
Record-Route: gtwy.ee.uottawa.ca
From:  Bill Gate <sip:bill@Microsoft.com>
To: Ken Chan <sip:ken@uottawa.ca>
Contact: Ken Chan <sip:ken@site.uottawa.ca>
Call-ID: 56258002189@site.uottawa.ca
CSeq: 1 INVITE
Content-Type: application/sdp
Content-Length: 187
```

Figure 2: INVITE request from Bill to Ken (left) and OK response without SDP from Ken to Bill (right) captured at gtwy.ee.uottawa.ca

The top portion of the message is the message header. The first line is the request line, which contains the Method name 'INVITE', the Request-URI (e.g. ken@site.uottawa.ca), and the SIP Version. The Request-URI names the current destination of the request. It generally has the same value as the "To" header field but may be different if the caller is given a cached address that offers a more direct path to the callee through the "Contact" field. The "From" and "To" header fields indicate the respective registration address of the caller and of the callee. They usually remain unchanged for the duration of the call. The "Via" header fields are optional and indicates the path that the request has traveled so far. This prevents request looping and ensures replies take the same path as the requests, which assists in firewall traversal and other unusual routing situations. Only a proxy may append its address as a "Via" header value to a request message. When the corresponding response message arrives at a proxy, the proxy would remove the associated "Via" header from the response message header. The "Record-Route" request and response header fields are optional fields and are added to a request by any proxy that insists on being in the path of subsequent requests for the same call leg. It contains a globally reachable Request-URI that identifies the proxy server. "Call-Id" represents a globally unique identifier for the current call session. The Command Sequence ("CSeq") consists of a unique transaction-id and the associated request method. It allows user agents and proxies to trace the request and the corresponding response messages associated to the transaction. The "Content Type" and "Content-Length" header fields indicate the type of the message body's content and the length of the message body measured in bytes.

The sample response, excluding the associated SDP body in Figure 2, indicates a success 2XX response (2XX is a success response code in the range of 200 and 299) returned by the callee. The first line of a response message is the status line that includes the response string, code, and the version number. It is important to note that "Via" header fields are removed from the response message by the corresponding proxies on the return path. When the calling user agent client receives this success response, it will send an "ACK" message that has a very similar format as the "INVITE" request message, except the Method name would be “ACK” instead of “INVITE” in the request line. Note that SDP is not required in the body of the “ACK” message.

We have discussed only some of the key header fields in SIP. There are many header fields available in SIP and can become very complex. The associated RFC [1] is recommended for further details on SIP. In addition, the SIP community has specified a variety of traditional telephony services for SIP in [2,3,5], for example, Call Forward Busy (CFB), Call Waiting (CW), Originator Call Screening (OCS), Terminating Call Screening (TCS), etc. In the case of CFB, user ‘A’ calls user ‘B’ who is busy and replies with a busy response message. Then, the proxy will forward the call to user ‘C’. Since we have implemented very few advanced Internet telephony services (e.g. Call Forking), we will use CFB as the sample SIP service to describe our modeling approach in the subsequent sections.

3 Design Approach

Although the end result of this formal modeling exercise is to produce an SDL model that serves as a formal specification of SIP and some of the sample services, the modeling exercise fits into a bigger picture, that is, the overall development process to develop reliable SIP implementation. Before we dive right into the details of the SDL model, we present the overall design approach that we use in this project. Not surprisingly, the process is highly iterative by nature because the specifications of SIP services are a collection of informal textual service descriptions and sample call flows that do not offer all the detailed requirements up front. The process is best described in Figure 3, as a typical software development lifecycle, except it has incorporated formal modeling.

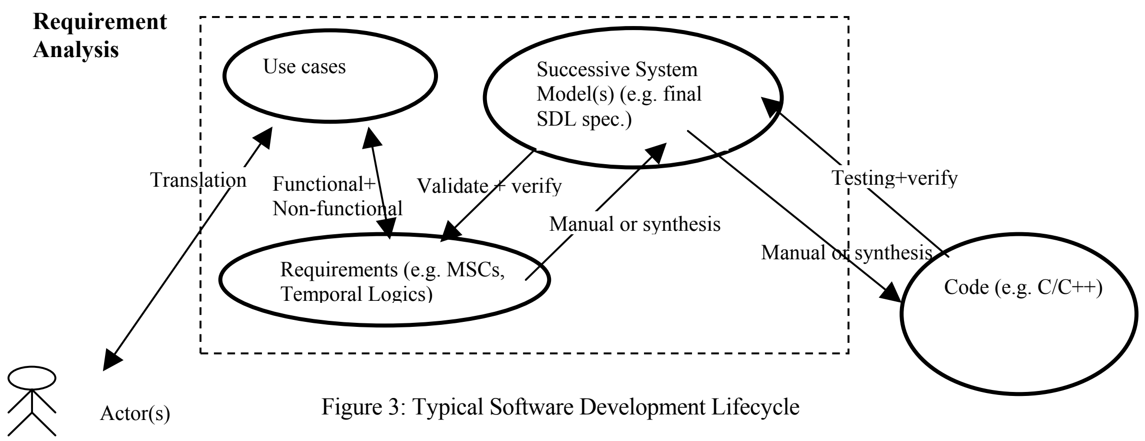


Figure 3: Typical Software Development Lifecycle

Tools that are based on formal methods can be used to ensure the final SDL model would meet these requirements. In a large project, we believe a system modeling team takes the responsibility of building system models and verifying that all requirements, particularly performance requirements, can be met. The system model serves as the final specification resulting from the requirement analysis. The design and implementation teams may use tools to synthesize the implementation or manually translate the specification to code. In this paper, the focus is on the generation of message sequence charts as both our use case scenarios and test cases, and the modeling of SIP and its services as a SDL system model. Synthesis is beyond the scope of this paper.

After the sample service scenarios are translated into message sequence charts, an SDL model of SIP and selected services is created. We found that it is useful to define three sets or versions of the SDL models: the initial model, the version that complies with the published specification (the standard model), and the refined (final) version that includes the preventive measures for feature interactions (the refined model). In Figure 4, we show that the initial models involve validation of each feature against MSC test cases. Then, the standard model is checked for feature interactions. Finally, the refined model is produced.

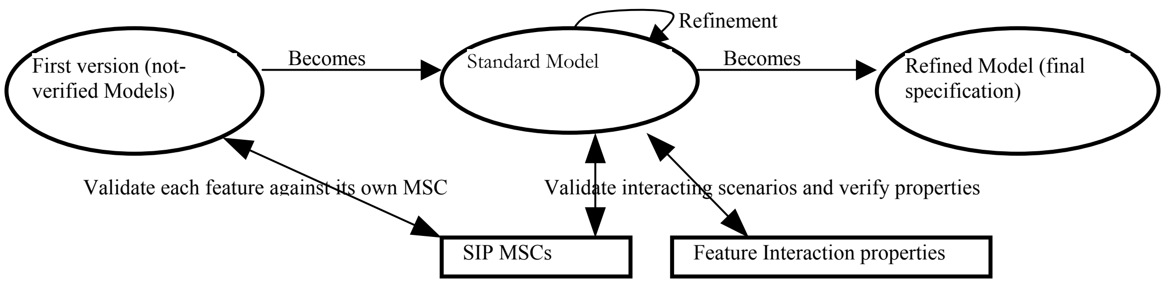

Refinement
First version (not-verified Models)
Becomes
Standard Model
Becomes
Refined Model (final specification)
Validate each feature against its own MSC
Validate interacting scenarios and verify properties
SIP MSCs
Feature Interaction properties

Figure 4: Iterative specification process

The approach to model SIP and its services begins with modeling the core signaling functionality of SIP; we call this the basic (telephony) service of SIP. The basic service includes establishing a two party call, terminating a two party call, suspending the call, presenting dial tone, busy signal, no answer signal, ringing, and alerting signal. The user agent and proxy are the only SIP entities that are modeled. After we complete the basic service, we add additional SIP services such as multi-party call signaling features (call redirection, forwarding, and holding (suspending)) to the model. These protocol features are essential to enabling the development of more complex telephony features such as CFB, CW, OCS, TCS, etc. Advanced Internet telephony features such as Call Forking (CF) and Auto-callback (ACB) will be added at the later stage. The complete SDL model consists of over 60 pages of diagrams, thus only selected diagrams will be presented in this paper.

4 From use case diagrams to use case scenarios as MSC

We begin our modeling exercise with use case analysis. The following subsections describe how use case diagrams and use case scenarios are derived from existing informal service specifications [2,3,5].

4.1 Defining Use Case

Our use cases are defined in forms of use case diagrams [21]. They are based on the informal call flow diagrams and textual service specification described in various IETF documents [2,3,5]. Each actor in our use cases has a specific role in a service; Figure 5 shows the use case diagram of Call Forward Busy (CFB).

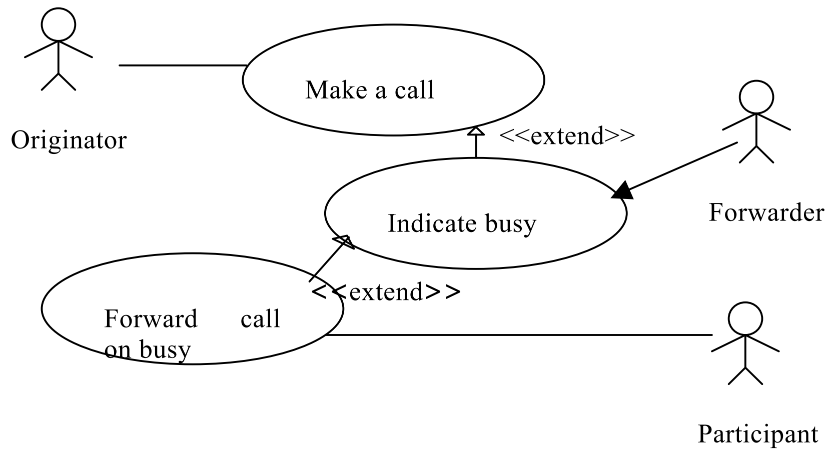


Figure 5: Use Case Diagram of Call Forward Busy

In this example, there are three distinct actors (originator, forwarder, and participant/forwardee). In section 5, we show an originator corresponds to the “SipUserAgent1_1” user agent process. The forwarder corresponds to the “SipUserAgent1_2” user agent process. The forwardee corresponds to the “SipUserAgent1_3” user agent process. The “indicate busy” use case extends the “make a call” use case. The "extend" relationship is used to describe additional functionality that the extended use case has. We will show how use case scenarios can be derived from such user case diagram in the next section.

4.2 Defining use case scenarios as MSC

Since the IETF drafts have provided a call flow diagram or success scenario for each sample service in graphical notation [3], we translate these scenarios into message sequence charts. However, we note that these call flow diagrams only include the sequence of exchanged SIP messages at the protocol level. They do not represent service scenarios in the sense of use cases. In the previous section, we show how we apply use case analysis and come up with use case diagrams.

Following standard practice of software engineering, we think that it is important to define service usage scenarios at the interface between the user and the system providing the communication service. We have therefore associated each use case (e.g. "make a call", or "indicate busy", or "forward call on busy") to a partial trace of a certain call flow scenario. Then, a use case diagram which contains the relationships between use cases and actors represents a full trace of a service. Thus, we can translate a use case diagram to at least one service usage scenario. As an example, Figure 6 shows a use case scenario (service usage scenario) of CFB, which corresponds to the use case diagram in Figure 5 and also to the call flow diagram of CFB found in [3]. We have also defined an abstract user interface (see Section 6.3) which represents the interactions at the service level. These interactions between the users and the SIP system describe use case scenarios of SIP services. The users are the actors of the use case scenarios and are represented by the environment "env_0" in SDL.

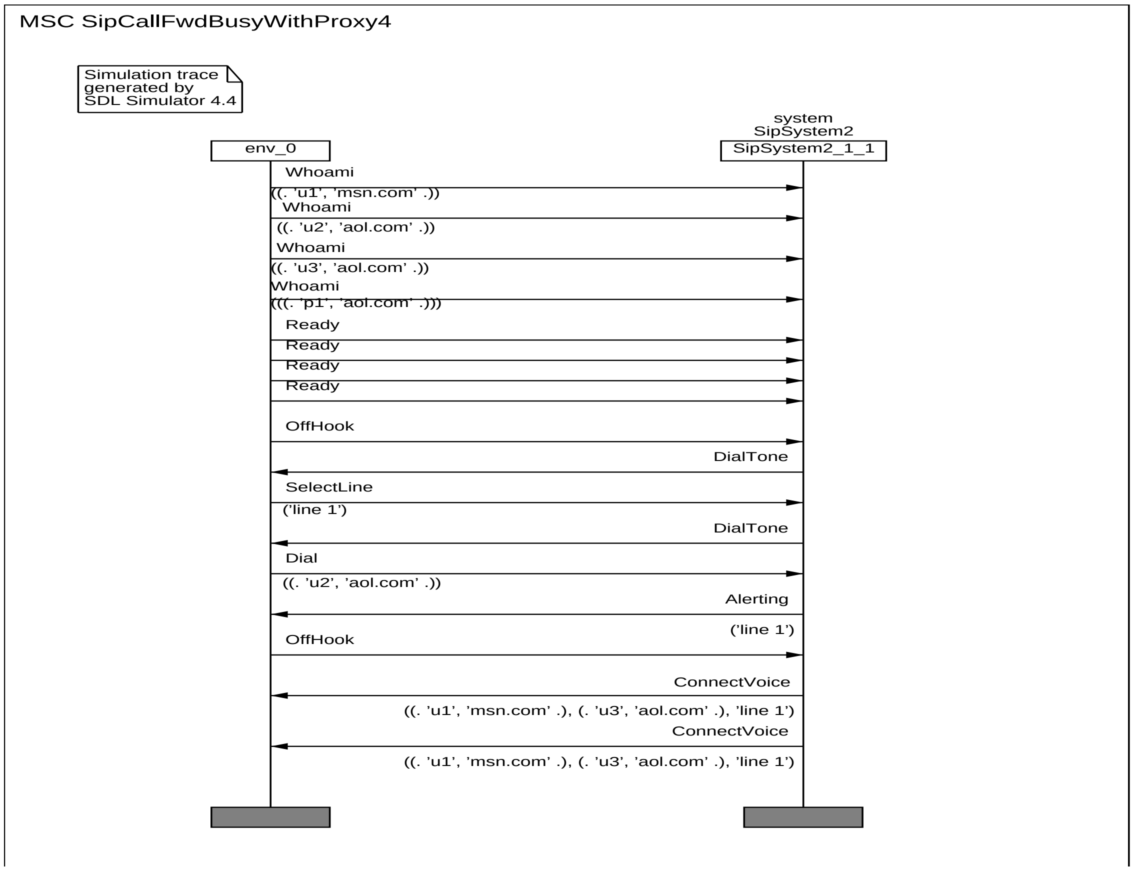


Figure 6: Call Forward Busy Use Case Scenario

5 Converting call flows to MSC as test case scenarios

The combination of the use case scenario with the corresponding scenario of exchanged SIP messages from [3] is represented in Figure 7. We may call such a combined scenario a service and protocol scenario. It can be used as a test case for validating the SDL specification of the SIP protocol, as explained in Section 8.

After having defined our use case scenarios, which correspond largely to the call flow diagrams that we found in [3], we can combine the use case scenario with the corresponding scenarios of exchanged SIP messages from [3]. Figure 7 shows such an MSC corresponding to the use case of Figure 6. We may call such a sequence chart a "combined service and protocol scenario". Such a combined scenario may be used as a test case for validating the SDL specification of the SIP protocol, as explained in Section 8.

The test cases are written as message sequence charts because we use the Telelogic Tau's Validator to verify our SDL model against the combined scenario.

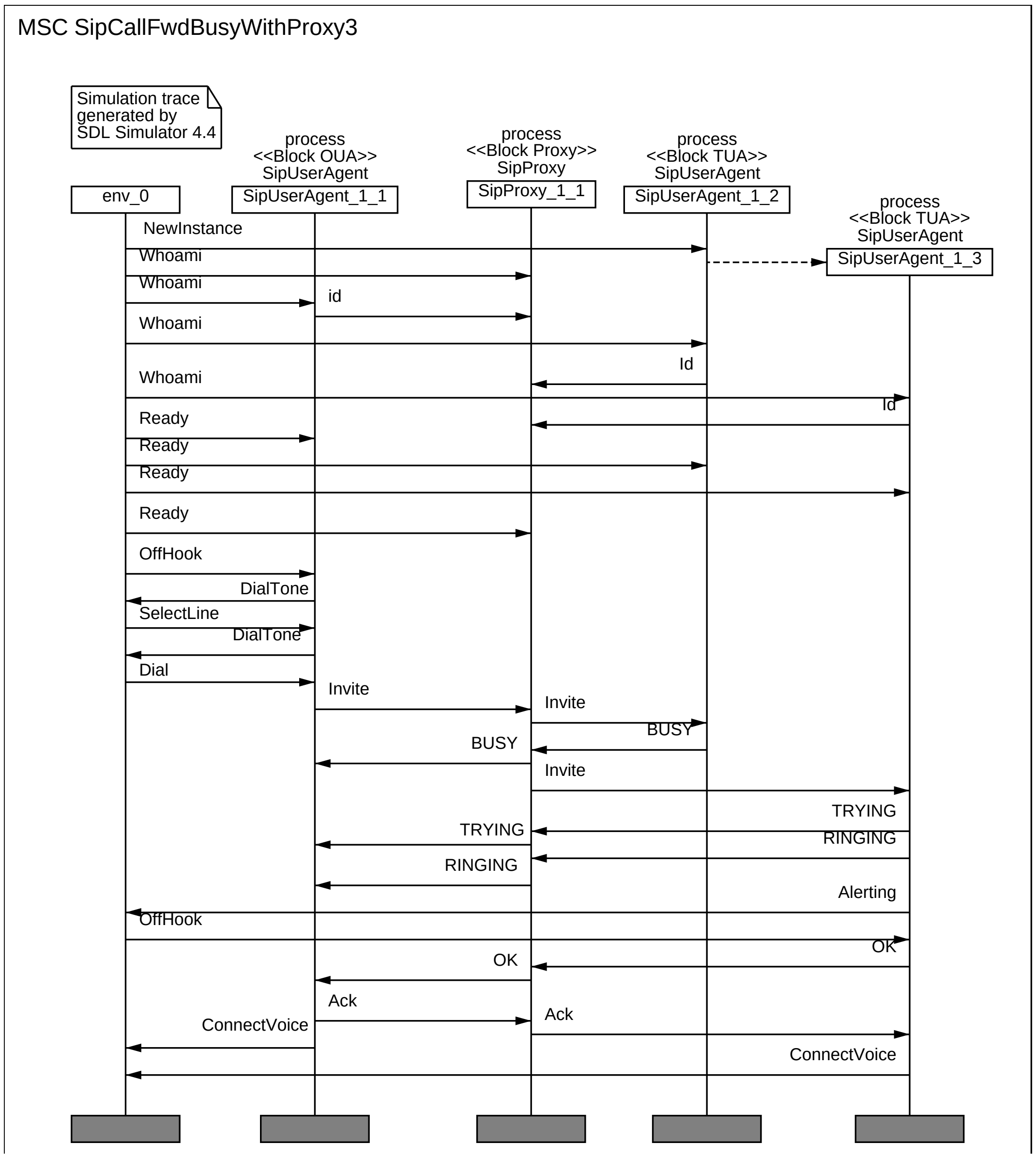


Figure 7: Call Forward Busy Test Scenario in IETF SIP Service Examples draft

The sample call forward busy described in the IETF SIP Service Example draft [2], which is shown in Figure 7, is used to demonstrate our process of converting informal requirements to message sequence charts. In Figure 7, user 'A' calls user 'B' who is busy and replies with a busy response message. Then, the proxy forwards the call to user 'C'. Since Figure 7 can be used as a test scenario MSC against which the model is verified, such test scenario must be a complete trace which includes the initialization phase of the simulation. However, the initialization phase (signal "NewInstance", "Whoami", "Id", and "Ready" are all operational management signals, see Section 6.1.2) is not part of the service specification; it is our invention for facilitating simulation and verification of the model. The call flow diagram and the service requirements of call forwarding described in [5] can be used in conjunction for creating a formal call forward busy specification. Before we begin developing the SDL model, we need to articulate the requirements in formal notations. The first step is to convert the call flow diagram to a syntactically correct message sequence chart. In Figure 7, the method name of the request message (e.g. 'INVITE') is converted to an SDL signal (or MSC message). The SIP message parameters would become the signal parameters. Also, we can use co-regions to express the general ordering of messages because the SIP RFC indicates that the arrival of a call setup response message such as 'TRYING' or 'RINGING' can be out of order. However, we decided that we could not use co-regions in our message sequence charts because Tau does not support co-regions in the validation process. The symbol of instance end is used to end all instances in our message sequence charts because it defines the end of the description of an instance with a message sequence charts; it does not define the termination of the instance. An action that describes the interaction between a SIP entity and a non-SIP entity is written as a signal/message from a SIP entity to the environment in our message sequence charts. For example, ConnectVoice is a signal that is sent from the user agent to the media controller to establish RTP voice streams. SelectLine signal is used to select the line on the phone. Evidently, our use case scenarios written as message sequence charts together with the SDL model give a more precise formal service specification of SIP services than the IETF drafts.

After we have developed the SDL system model for SIP, we would verify the model against these message sequence charts which serve as the basic SIP compliant test cases for the system model. The final system model that includes all the preventive measures for feature interactions is the specification of the SIP protocol entities. The SDL model may also be used to synthesize the code base, but it is beyond the scope of this paper.

6 Defining the structural model

In this section, the structural definitions of the SIP entities are discussed. The relationship between modeled entities, their interfaces, and attributes are considered parts of the structural definition. A SDL system represents static interactions between SIP entities. The channels connected between various block instances specify the signals or SIP messages that are sent between user agents and/or proxies. Block and process types (e.g. SipUserAgentType, SipProxyType) are used to represent SIP entity types such as user agent and proxy. In addition, the Tau tool Organizer component [8] allows the service designer to partition the specification into a number of modules, called SDL Packages. SDL Packages are used to package SDL entities for reuse.

6.1 Core SIP entities

The following subsections describe the two main SIP entity types in the model: User Agent and Proxy.

6.1.1 UserAgent

A SIP User Agent contains both, what is called in SIP, a user agent client (UAC) and a user agent server (UAS). Since a user agent can only behave as either a UAC or UAS in a SIP transaction, the user agent is best represented by the inheritance of UAC and UAS interfaces. The inheritance relationship is modeled using separate gates (C2Sgate and S2Cgate) to partition the user agent process and block into two sections: client and server. The "Envgate" gate manages the sending and receiving of "Abstract User" signals between the user agent and the environment (see Figure 8). An instantiation of a block type represents an instance of a SIP entity such as user agent or proxy, and contains a process instance that describes the actual behavior of the entity. The process definition file contains the description of all the state transitions or behaviors of the features to which the SIP entity has subscribed. In addition, each SIP entity must have a set of permanent and temporary variables for its operations. In the case of a user agent, the permanent variables store the current call processing state values of the call session (e.g. To, From). The temporary variables store the values of the consumed messages for further processing.

6.1.2 Proxy

Similar to a user agent, a proxy consists of client and server portion. It tunnels messages between user agents but also intercepts incoming messages, injects new messages, or modifies forwarding messages on behalf of the user agent(s). A proxy is also a favorite entity to which features are deployed. If a proxy needs to keep track of the states of a session for a feature, it is considered a stateful proxy, and vice versa. A SIP Proxy has four gates that interact with user agents or proxies: client-to-proxy (C2Pgate), proxy-to-client (P2Cgate), server-to-proxy (S2Pgate), and proxy-to-server (P2Sgate). The "Envgate" gate manages the sending and receiving of "Abstract User" signals between the user agent and the environment (see Figure 8). In our SDL model, we use different SDL systems to represent different structural bindings between SIP entities and to simulate a particular set of call scenarios. The most complex system in our telephony model (see Figure 8) realizes the concept of originating and terminating user endpoints. It contains an originating user agent block, a proxy block, and a terminating user agent block. The originating block contains all the user agent process instances that originate SIP requests while the terminating block contains all the user agent process instances that receive these requests. Upon receiving a request, a terminating user agent would reply with the corresponding response messages. It is important to note that only the originating user agent and proxy instances can send SIP requests (including acknowledgements).

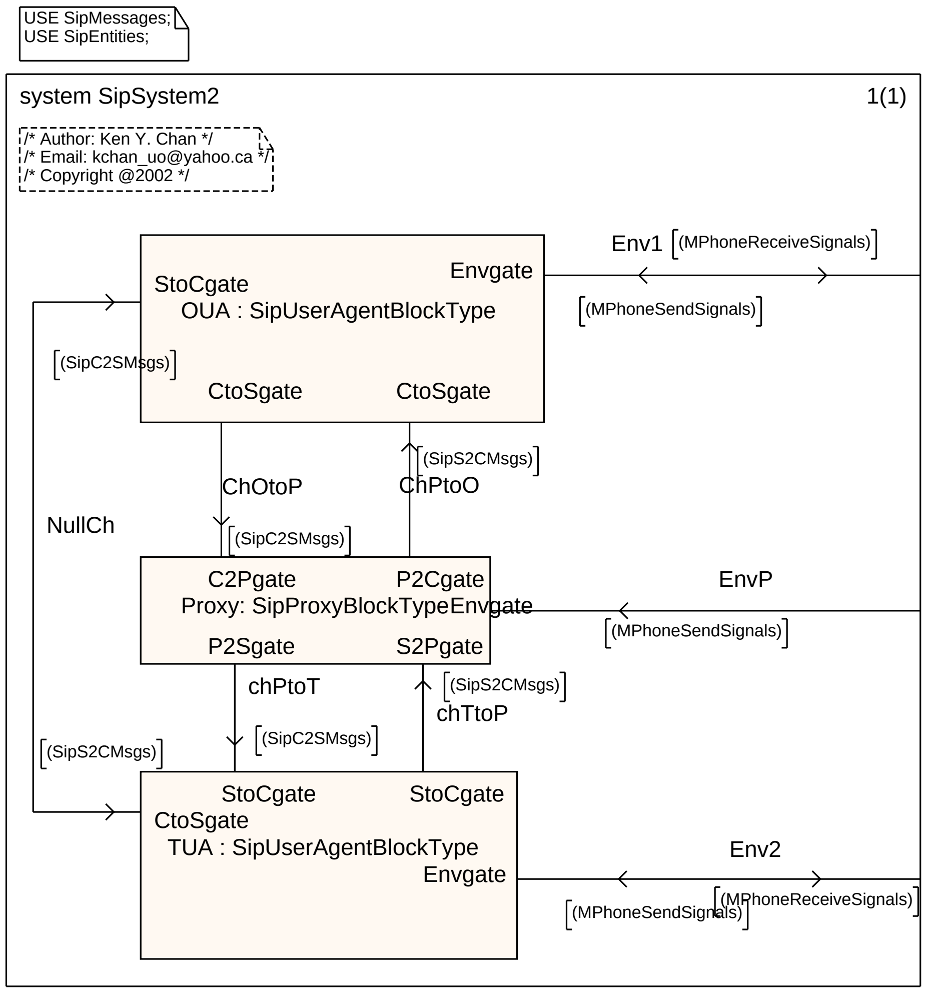


Figure 8: System Diagram of Proxy and User Agents

All blocks are initialized with one process instance. During the simulation, a 'NewInstance' "Misc User" signal can be sent to a process instance to create a new process instance. Signals such as "NewInstance", "Whoami", "Id", and "Ready" are not "Abstract User" signals. They are created for the purpose of operational management, configuration and administration. We grouped all non-SIP signals to the "Envgate" gate. We believe it was an oversight. We should have created different gates for different types of environmental signals. This would be a part of our future work.

Before the first "INVITE" message is sent, the environment must initialize each user agent and proxy instance with a unique Internet address by sending them a message called 'Whoami'. The user agent instances would in turn send an 'Id' message along with its Internet address and process id (PId) to the proxy. Thus, the proxy can establish a routing table for routing signals to the appropriate destinations during simulation. Similar to the user agent, a proxy has a set of permanent and temporary variables for its operations. In Figure 9, we have a block interaction diagram that describes the relationship between the process set and its block.

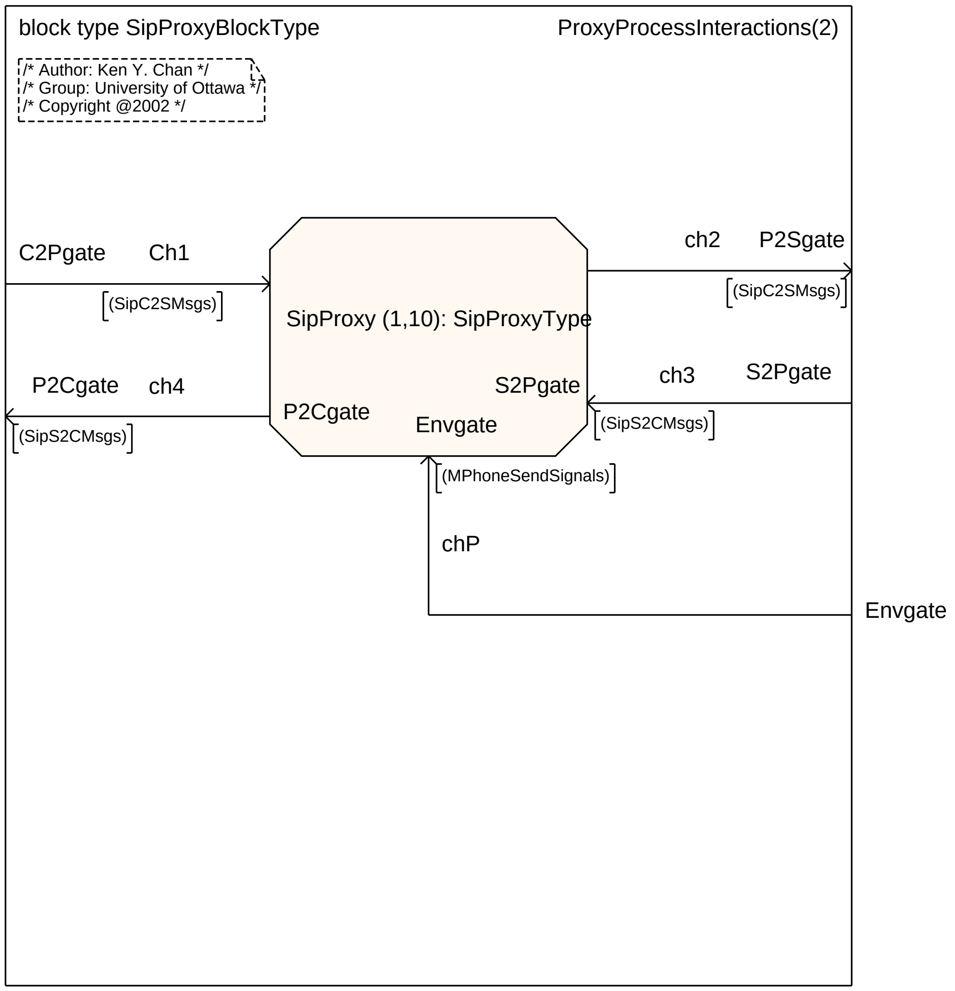


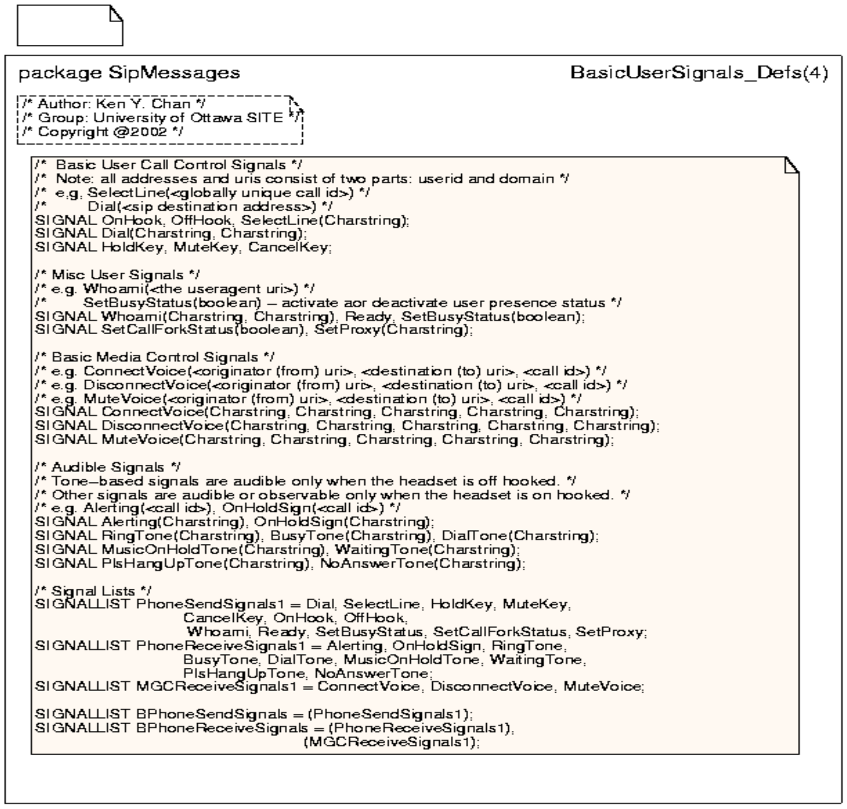
Figure 9: Block-Interaction Diagram of Proxy Type

6.2 SIP Messages

In this case, SIP messages are defined as SDL signals in the SIPMessage package. We have defined only the main header fields of the SIP header because we are interested in only the operation (method) and the endpoints of the call session. These fields in a SIP message are represented by the corresponding signal parameters. Since we could not find any SDL package that supports *linked list* data structure in Tau, the number of variable fields such as ‘Via’ and ‘Contact’ are fixed in our model. Complex data and array types of SDL have been tried for this purpose, but were dropped from the model because they may cause the Tau validation engine to crash. Instead, we used a fix number of parameters to simulate these variable SIP header fields.

6.3 Abstract User Interface

As mentioned in Section 4, a set of user signals (see Figure 10), that is not part of the SIP specifications [1,2,3], has also been defined here to facilitate simulation and verification. They represent the interface between the user and the local IP-telephony equipment in an abstract manner; we call this interface the “Abstract User interface”. The modeling of these user-observable behaviors is essential to describe feature interactions; however, the SIP protocol messages described in the SIP standard do not describe these user interactions properly. For example, an INVITE message in SIP may play different roles. It is more than just a call setup message; for instance, it may be used for putting the call on hold in the middle of a call (mid-call features) [2]. To make the simulation as realistic to real phone calls as possible, the “Abstract User Interface” includes signals such as Offhook, Onhook, Dial, SelectLine, CancelKey, RingTone, AlertTone, TransferKey which relate to the actions that are available on most telephones units on the market (see Section 4 and Figure 6).



```
package SipMessages                                   BasicUserSignals_Defs(4)

/* Author: Ken Y. Chan */
/* Group: University of Ottawa SITE */
/* Copyright @2002 */

/*  Basic User Call Control Signals */
/*  Note: all addresses and uris consist of two parts: userid and domain */
/*  e.g. SelectLine(<globally unique call id>) */
/*        Dial(<sip destination address>) */
SIGNAL OnHook, OffHook, SelectLine(Charstring);
SIGNAL Dial(Charstring, Charstring);
SIGNAL HoldKey, MuteKey, CancelKey;

/* Misc User Signals */
/* e.g. Whoami(<the useragent uri>) */
/*       SetBusyStatus(boolean) -- activate aor deactivate user presence status */
SIGNAL Whoami(Charstring, Charstring), Ready, SetBusyStatus(boolean);
SIGNAL SetCallForkStatus(boolean), SetProxy(Charstring);

/* Basic Media Control Signals */
/* e.g. ConnectVoice(<originator (from) uri>, <destination (to) uri>, <call id>) */
/* e.g. DisconnectVoice(<originator (from) uri>, <destination (to) uri>, <call id>) */
/* e.g. MuteVoice(<originator (from) uri>, <destination (to) uri>, <call id>) */
SIGNAL ConnectVoice(Charstring, Charstring, Charstring, Charstring, Charstring);
SIGNAL DisconnectVoice(Charstring, Charstring, Charstring, Charstring, Charstring);
SIGNAL MuteVoice(Charstring, Charstring, Charstring, Charstring, Charstring);

/* Audible Signals */
/* Tone--based signals are audible only when the headset is off hooked. */
/* Other signals are audible or observable only when the headset is on hooked. */
/* e.g. Alerting(<call id>), OnHoldSign(<call id>) */
SIGNAL Alerting(Charstring), OnHoldSign(Charstring);
SIGNAL RingTone(Charstring), BusyTone(Charstring), DialTone(Charstring);
SIGNAL MusicOnHoldTone(Charstring), WaitingTone(Charstring);
SIGNAL PlsHangUpTone(Charstring), NoAnswerTone(Charstring);

/* Signal Lists */
SIGNALLIST PhoneSendSignals1 = Dial, SelectLine, HoldKey, MuteKey,
                     CancelKey, OnHook, OffHook,
                     Whoami, Ready, SetBusyStatus, SetCallForkStatus, SetProxy;
SIGNALLIST PhoneReceiveSignals1 = Alerting, OnHoldSign, RingTone,
                     BusyTone, DialTone, MusicOnHoldTone, WaitingTone,
                     PlsHangUpTone, NoAnswerTone;
SIGNALLIST MGCReceiveSignals1 = ConnectVoice, DisconnectVoice, MuteVoice;

SIGNALLIST BPhoneSendSignals = (PhoneSendSignals1);
SIGNALLIST BPhoneReceiveSignals = (PhoneReceiveSignals1),
                                  (MGCReceiveSignals1);
```

Figure 10: “Abstract User” Interface (Signals)

The parameters of each “Abstract User” signal are designed to give an unambiguous semantics to a user action. For example, The ‘ConnectVoice’ signal has five parameters: From’s user and domain, To’s user and domain, and call-id. These parameters mark the logical endpoints of a call segment, which is used by a media controller to establish a voice stream between the two hops. All the output signals have the call-id as their parameter that represents the line number of the call.

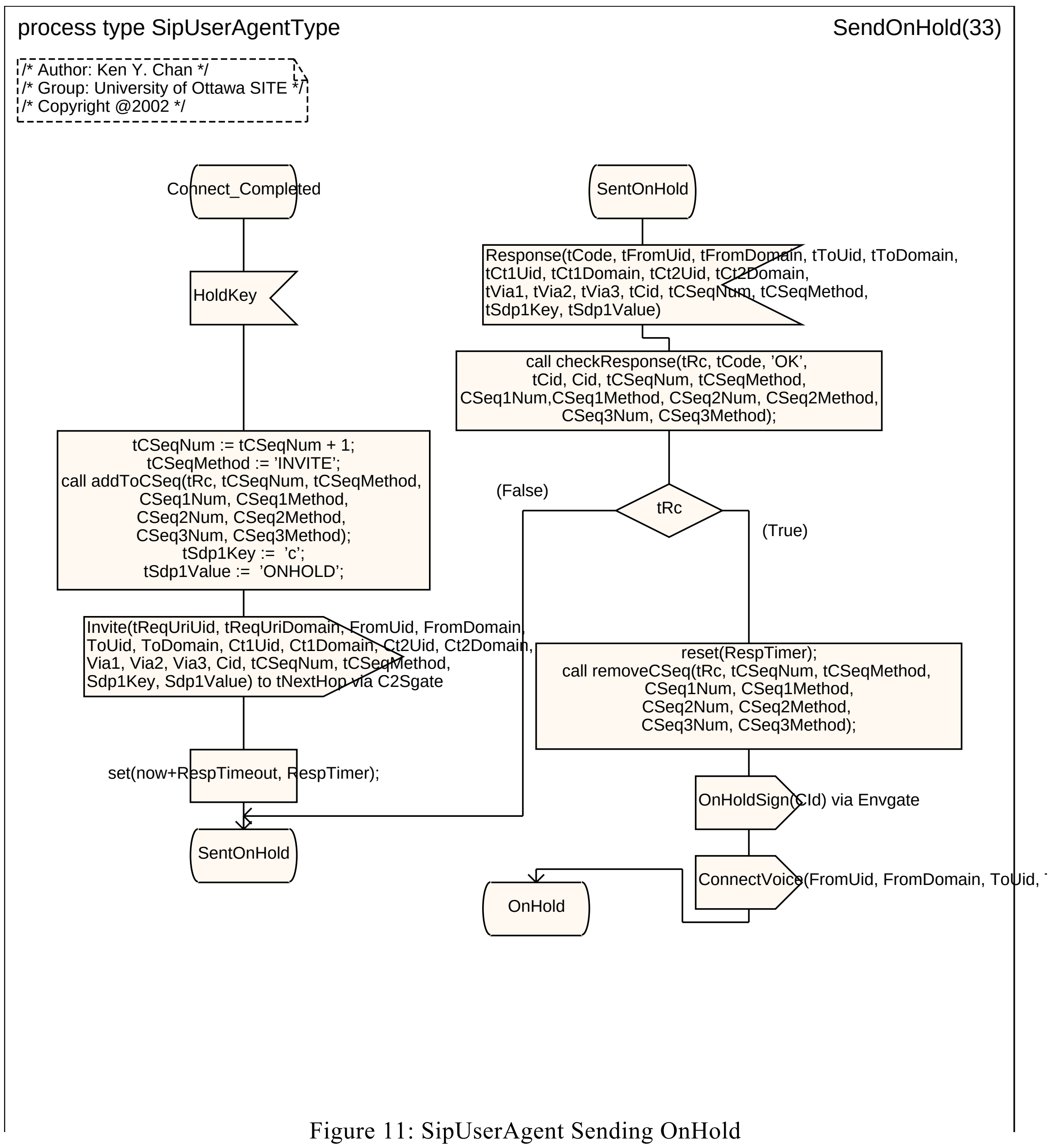
Finally, one may ask how these high-level “Abstract User” signals are connected to the protocol’s low-level primitives (SIP messages). We initially considered having a dedicated process in each SIP entity block (SipUserAgentBlockType and SipProxyBlockType) to map these user signals to SIP signals which are managed by another dedicated process (SipUserAgent and SipProxy). However, we found it would introduce unnecessary complexity to the design so we abandoned such approach. Instead, each block type has one process which contains all the behavior of the block type. The mapping between “Abstract User” signals and SIP signals is simple. In Section 7.1, Figure 11 shows when the user agent receives a “HoldKey” “Abstract User” signal at the “Connect_Completed” state, it updates its state variables, sets the timer, and sends the corresponding “Invite” message to the destination. Then, the user agent would make a state transition from the “Connect_Completed” state to “SentOnHold” state.

Similarly, when the originating user agent receives a success response from the destination, the user agent would cancel the timer and send the corresponding “OnHoldSignal” signal to the environment (e.g. the caller phone alerts the user with an On-hold tone). Finally, the user agent would make a state transition from the “SentOnHold” state to the “OnHold” state. In general, the mapping between “Abstract User” signals and SIP signals is based on a relationship of trigger events and actions. For example, an “Abstract User” signal can trigger the sending of a SIP signal, and vice versa. We will explain this in more detail in Section 7.

7 Behavior specification

7.1 User Agent Process Specifications

In general, a SIP feature or service is represented by a set of interactions between users and the user agent processes, and possibly the proxy processes. Each process instance plays a role in a feature instance. A process instance contains state transitions which represent the behaviors that the process instance plays in a feature instance. In our model, we capture the behavior of a SIP entity in an SDL process type (e.g. UserAgentType). The first feature we model is the basic SIP signaling functionality, also known as the basic service. Each process type has state transitions that describe the basic service. In the case of a user agent, the process includes the UAC and UAS behavior. We define a feature role as the behavior of a SIP entity that makes up a feature in a distributed system. A feature role is invoked or triggered by trigger events. The entry states of a feature role in a SIP entity are the states for which the trigger events are defined as inputs. For example, the on-hold feature can be triggered in the ‘Connect_Completed’ (entry) state by an INVITE message with ‘ONHOLD’ as one of its parameters (Figure 11). After the invite message is sent, a response timer is immediately set. Then, the user agent is waiting for the on-hold request to be accepted. When the other user agent responds with an OK message, the requesting agent would cancel or reset the response timer and inform the device to display the on-hold signal on the screen.



process type SipUserAgentType
SendOnHold(33)
/* Author: Ken Y. Chan */
/* Group: University of Ottawa SITE */
/* Copyright @2002 */
Connect_Completed
HoldKey
tCSeqNum := tCSeqNum + 1; tCSeqMethod := 'INVITE'; call addToCSeq(tRc, tCSeqNum, tCSeqMethod, CSeq1Num, CSeq1Method, CSeq2Num, CSeq2Method, CSeq3Num, CSeq3Method); tSdp1Key := 'c'; tSdp1Value := 'ONHOLD';
Invite(tReqUriUid, tReqUriDomain, FromUid, FromDomain, ToUid, ToDomain, Ct1Uid, Ct1Domain, Ct2Uid, Ct2Domain, Via1, Via2, Via3, Cid, tCSeqNum, tCSeqMethod, Sdp1Key, Sdp1Value) to tNextHop via C2Sgate
set(now+RespTimeout, RespTimer);
SentOnHold
SentOnHold
Response(tCode, tFromUid, tFromDomain, tToUid, tToDomain, tCt1Uid, tCt1Domain, tCt2Uid, tCt2Domain, tVia1, tVia2, tVia3, tCid, tCSeqNum, tCSeqMethod, tSdp1Key, tSdp1Value)
call checkResponse(tRc, tCode, 'OK', tCid, Cid, tCSeqNum, tCSeqMethod, CSeq1Num,CSeq1Method, CSeq2Num, CSeq2Method, CSeq3Num, CSeq3Method);
(False)
tRc
(True)
reset(RespTimer); call removeCSeq(tRc, tCSeqNum, tCSeqMethod, CSeq1Num, CSeq1Method, CSeq2Num, CSeq2Method, CSeq3Num, CSeq3Method);
OnHoldSign(CId) via Envgate
ConnectVoice(FromUid, FromDomain, ToUid,
OnHold

Figure 11: SipUserAgent Sending OnHold

In general, trigger events are expressed as incoming signals; pre-condition, post-conditions, constraints are expressed as enabling conditions or decision. Actions are tasks, procedure calls, or output signals. As a triggering event being consumed by the user agent process, the parameters of the event may be examined along with the pre-conditions of the feature. Then, actions such as sending out a message and modifying the internal variables may be executed. Post-conditions and constraints on the action may also be checked. Finally, the process progresses to the next state.

Although we have described the skeleton of the model (systems, blocks, data variables in processes), we have not defined the essential behavior of the model, which are the state transitions of the processes. The question is how to come up with the state transitions for the "initial" models. Since we have a message sequence chart describing the success scenario of a two party call, we assign a unique state to each input and output of the instance type (user agent or proxy type). For example, if

‘Connect_Request’ state is the current state and we have received a ‘HoldKey’ signal, we would send an ‘Invite’ request to the other user agent. We do not put any timers in the ‘initial’ model because we would like to produce a simple rapid prototype model that we can experiment with. However, we add timers (e.g. response timer) to the ‘standard’ and ‘refined’ models.

A state transition occurs when 1) an “Abstract User” signal is received from the environment, 2) a request or response message is received, or 3) a continuous signal is enabled. The so-called Continuous Signal in SDL is used to model the situation in which a transition is initiated when a certain condition is fulfilled. For example if the UAS is busy, the boolean ‘isBusy’ would be true in the ‘Server_Ring’ state. The UAS would immediately send the BUSY response to the caller. This way, we would not have to worry about the timer expiration because we do not need to send a busy toggling signal to simulate busy during a simulation. An asterisk ‘*’ can be used in a state and signal symbol to denote any state or any signal; its semantics is equivalent to a wildcard. A state symbol with a dash ‘-‘ means the current state. Error handling, such as response timer expiration, can easily be modeled with SDL timers and a combination of ‘*’ and ‘-‘ state symbols. For example, when the response timer expires, a timeout message would be automatically sent to the input queue of the user agent process. The expiration of the response message is generalized as the situation in which the receiving end does not answer the request in time. Thus, a ‘NoAnswer’ signal is sent to the environment. Finally, the process can either return to the previous state or go directly to the idle state through the ‘Jump_Idle’ connector in this case.

Moreover, we can add additional features or services such as CFB, OCS, and other services, to the system. To add behaviors of additional features to a process type, we can subtype a “basic” process type such as UserAgentType. The derived type has the same interfaces and also additional state transitions [11]. We do not want to add new interfaces (signal routes) to the process types because we do not want to change the interfaces of the block types. If a feature requires new SIP methods and response codes, we would not need to change the interfaces because method names and response codes are simply signals parameters in our model. Thus, we avoid the need to add new interfaces whenever a new feature is defined.

7.2 Proxy Process Specification

A proxy is different from a user agent because a proxy processes messages exchanged between user agents or proxies. We model the proxy also as an entity to which telephony features may be deployed. The proxy performs a predefined set of actions based on the header information (e.g. originator and destination addresses) of the incoming message(s). A proxy listens for response messages from the terminating user agent and forwards the response message to the originating agent based on the routing information that was set during the initiation phase of the simulation.

In summary, a state model of SIP and its sample services can be derived from message sequence charts. Many protocol features such as timer expiration, parameter checking, sending and receiving messages can be mapped to equivalent structures in SDL. The object-oriented extension to SDL allows specialization of telephony services. Thus, SDL is suitable in modeling telephony services. In the next section, we will examine the validation and verification process.

8 Verification and Validation

The SDL specification that has been discussed in the previous subsection was constructed using the Telelogic Tau tool version 4.3 and 4.4 [8]. Tau offers many verification or reachability analysis features: bit-state, exhaustive, random walk bit state exploration and verification against a given MSC. Bit-state exploration is particularly useful and efficient [9] in checking for deadlocking interactions because it checks for various reactive system properties without constructing the whole state space like the exhaustive approach does.

We verified our SDL model of SIP mainly by checking whether the model would be able to realize specific interaction scenarios which were described in the form of Message Sequence Charts (MSCs) [7,8] (see Section 5). In fact, we used the scenarios described informally in [1,2], and rewrote them in the form of MSCs. Then we used the Tau tool to check that our SDL model was able to generate the given MSC. An MSC is verified if there exists an execution path in the SDL model such that the scenario described by the MSC can be satisfied. Thus, an MSC in Tau is considered as an existential quantification of the scenario. When “Verify MSC” is selected, Tau may report three types of results: verification of MSC, violation of MSC, and deadlock. Unless Tau is set to perform 100% state space coverage, partial state space coverage is generally performed and reported in most cases. Verification of the model against an MSC in Tau is apparently achieved by using the MSC to guide the simulation of the SDL model. As a result, the state space of the verification has become manageable [10].

Although we could use MSCs as the test cases of our SDL model, MSCs have limitations in terms of expressing quantification of instances and their behaviors. For example, we could not find a way to write an MSC to verify this condition: For all user agents that receive an ‘OK’ response, they must reply the sender of the response with an ‘ACK’ message. We note that such properties could also be checked during simulation using Observer Process Assertions [8], as discussed in Section 9. Also, Live Sequence Chart (LSC) [18], which has not been discussed much in this paper, appear to be a promising extension to MSCs in this context. However, Tau does not support LSC at the moment. Further research in the application of LSC would be a part of our future works.

9 Advantages of using SDL and Telelogic Tau and possible enhancements

In the course of modeling SIP services, we discovered not only the advantages but also some shortcomings of using SDL and CASE tools such as Telelogic Tau to model an IETF application protocol such as SIP. SDL is a good language to model SIP because SIP and its services can be easily expressed as interactions between extended communicating finite state machines. We can simulate a distributed SIP system a transition at a time using Tau. Other advantages of using Tau include 1) the ability to express test scenarios as MSCs and to verify these MSCs against the model, 2) the ability to verify the model does not violate system properties by writing these test properties (e.g. livelock) as Observer Process Assertions.

However, we find the lack of SDL packages that support a variety of abstract data types in Tau has been a problem in modeling SIP. Unlike bit-oriented protocols such as Ethernet and Token Ring, SIP is an ASCII-based, attribute-rich signaling protocol with mandatory, optional and variable size parameters. If we were to model SIP messages as SDL signals, we cannot easily insert, remove, search, and modify values from the optional and/or variable size header fields. Although it is not impossible to model variable header fields with fixed size arrays in Tau, it is a very inconvenient task. Many programming languages offer the language feature of pointer (C++ [17]) and/or reference (Java [16], C++) so that a programmer could build his/her customized data structures. Pointer is a dangerous programming feature that is normally not included in formal languages. The SDL language could be extended with additional built-in ADTs. We believe supports for common abstract data types such as *linked list* and *hash table* should be made mandatory in SDL. This would be sufficient to enhance our modeling experience. A *linked list* can be used to describe a variable–length parameter list whereas a *hash table* can be used to store all the key-value pairs in a SDP body. These two abstract data types eliminate the needs to re-index the content. Although SDL has the *Vector* type which is basically a subtype of *Array*, this *Vector* type does not have the properties of a Java *Vector*; it does not have insert, remove, and update operators.

Secondly, we believe the SDL language should be extended with string processing facilities. In programming languages such as Java and C++, the language definition comes with string processing libraries. For example, the *int indexOf(String substring)* operator of the *String* class in Java returns the index position of the first occurrence of the *substring* in that string object instance [16]. Similar operations can be added to the SDL language. Furthermore, regular expression operators may be added to the SDL language. These string processing operations would facilitate the checking and comparisons of SIP header values. Without these operations, it is difficult to develop a complete model of an IETF application protocol like SIP.

Furthermore, we find that the Tau validation tool needs to offer more flexible verification features. For example, the ability to incorporate model checking of the SDL system using temporal logic formula would be a bonus. We may use temporal logic formula to verify more complex distributed system properties such as live-locks. In addition, we could not easily specify actors in our use case scenarios; we group all the actors together as the environment. We could create additional actor/user processes to simulate actors. However, if we were to show the interactions between actors in our use case scenarios, the message sequence charts would also display the exchange of internal SIP messages between user agent and proxy processes. We could probably develop an actor-layer to partition user signals from internal signals but it would not be a trivial task. Ideally, Tau could offer different environment instances for different channels to a system. This way, we would not need to develop our actor-layer in the model.

Last but not least, we find that the SDL editor lacks GUI context sensitive features, such as allowing a designer to select one of the existing state names when he/she creates a new state label, that are normally found in other CASE tools such as Microsoft Visual Studio [19]. Cutting and pasting objects between pages is also not always allowed; these some the minor improvements that can be made to the tools.

10 Conclusion and future works

In this paper, we have described our approach to the formal modeling of the IETF Session Initiation Protocol (SIP) and its services in SDL. We have discussed the advantages and shortcomings of using a formal language such as SDL to model an IETF application signaling protocol like SIP. Moreover, we have shown that modeling a complex IETF protocol like SIP using CASE tools such as Telelogic Tau is feasible. Also, we have explained that the “Abstract User” interface that we developed has allowed us to develop a more user-centric and precise formal service specification for SIP. We believe SIP or any IETF application protocols should be specified from a user-centric perspective. Currently, some of the IETF RFCs and drafts for the application protocols are not written with use case analysis in mind. Many researchers have found IETF documents difficult to read. In addition, we have shown that our SDL framework allows us to reuse and to add SIP services to the core protocol quite easily. Furthermore, we have discussed the enhancements that may be made to the SDL language and Tau tools to improve the modeling experience of IETF protocols.

As part of our future works, we would like to modify our model to support the new SIP standard which is specified under [13]. We intended to start the model with the latest standard but unfortunately the new RFC came to our attention only in the later stage of our project. We have some ideas on how to extend our model to support the new standard but it is beyond the scope of this paper. In addition, we would like to use the Target Expert tool included in Tau to generate target executables in the future.

Acknowledgements

This project would not be possible without the support of Communications and Information Technology Ontario (CITO) and Natural Science and Engineering Research Council (NSERC), Canada. The tools and the support we have received from Telelogic and Klocwork Inc. are appreciated. Furthermore, we would like to thank our colleagues at SITE, in particular Dr. Luigi Logrippo from Université du Québec en Outaouais and Dr. Daniel Amyot from University of Ottawa, for their insights to the domain of formal telephony specification and SDL.

References

1. M. Handley, H. Shulzrinne, E. Schooler, and J. Rosenberg, “SIP: Session Initiation Protocol”, Request For Comments (Proposed Standard) 2543, Internet Engineering Task Force, Mar. 1999.
2. A. Johnston, R. Sparks, C. Cunningham, S. Donovan, and K. Summers, “SIP Service Examples”, Internet Draft, Internet Engineering Task Force, June 2001, Work in progress.
3. A. Johnston, S. Donovan, R. Sparks, C. Cunningham, D. Willis, J. Rosenberg, K. Summers, and H, Schulzrinne, “SIP Call Flow Examples”, Internet Draft, Version 5, Internet Engineering Task Force, June 2001. Work in progress.
4. K. Chan, and G. v. Bochmann, "Methods for Designing IP Telephony Services with Fewer Feature Interactions", Feature Interactions in Telecommunications and Software Systems VII, IOS Press, to be published, June 2003.
5. J. Lennox, H. Schulzrinne, and T. Porta, “Implementing Intelligent Network Services with Session Initiation Services”, http://www.cs.columbia.edu/~lennox/cucs-002-99.pdf, accessed on October 7, 2002

6. International Telecommunication Union, "ITU-TS Recommendation Z.100: Specification and Description Language (SDL)", ITU-TS, Geneva, Switzerland, 1999.
7. International Telecommunication Union, "ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)", ITU-TS, Geneva, Switzerland, 1996.
8. Telelogic Inc., "Telelogic Tau SDL & TTCN Suite", version 4.3 and 4.4, http://www.telelogic.com, accessed on Dec 20, 2002.
9. G.J. Holzmann, 'An improved protocol reachability analysis technique', Software Practice and Experience, Vol. 18, No. 2, pp. 137-161, 1988.
10. Ø. Hargen, "MSC Methodology", http://www.informatics.sintef.no/projects/sisu/sluttrapp/publicen.htm, accessed on Dec. 23, 2002, SISU, DES 94, Oslo, Norway, 1994.
11. J. Ellsberger, D. Hogrefe, and A. Sarma, "SDL - Formal Object-oriented language for Communication Systems", Prentice Hall Europe, ISBN 0-13-621384-7, 1997.
12. International Telecommunication Union, "Packet based multimedia communication systems", Recommendation H.323, Telecommunication Standardization Sector of ITU, Geneva, Switzerland, Feb. 1998.
13. J. Rosenberg, H. Shulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: Session Initiation Protocol", Request For Comments (Standards Track) 3261, Internet Engineering Task Force, June 2002.
14. J. Lennox, and H. Schulzrinne, "Feature Interaction in Internet Telephony", Sixth Feature Interaction Workshop, IOS Press, May 2000.
15. E.J.Cameron, N.D.Griffeth, Y.-J. Lin, M. E. Nilson, W. K. Shure, and H. Velthuijsen, "A feature interaction benchmark for IN and beyond," Feature Interactions in Telecommunications Systems, IOS Press, pp. 1-23, 1994.
16. Sun Microsystems Inc., "The Source for Java^{TM} Technology", version 1.x, http://java.sun.com, accessed on Feb 10, 2003.
17. B. Stroustrup, "Stroustrup: C++", http://www.research.att.com/~bs/C++.html, accessed on Feb 10, 2003.
18. W. Damm, and D. Harel, "LSCs: Breathing Life into Message Sequence Charts*", Formal Methods in System Design, Kluwer Academic Publishers, pp. 19,45-80, 2001.
19. Microsoft Inc., "Visual Studio Home Page", http://msdn.microsoft.com/vstudio, accessed on Feb 10, 2003.
20. M. Handley, and v. Jacobson, "SDP: Session Description Protocol", Request For Comments (Proposed Standard) 2327, Internet Engineering Task Force, April 1998.
21. Object Management Group Inc., "Unified Modeling Language", version 2.0, http://www.omg.org/uml/, accessed on Feb. 12, 2003.