Understanding SIP Through Model-Checking

Pamela Zave
AT&T Laboratories—Research, Florham Park, New Jersey USA
pamela@research.att.com

4 June 2008

Abstract

In recent years, SIP has become an important and widely-used protocol for IP-based multimedia services. Despite voluminous documentation, there is only scattered and informal material explaining the states of the protocol and the events that can occur in each state. To fill this gap, this paper presents a Promela model of invite dialogs in SIP. The model has been verified and validated with the Spin model-checker. The paper discusses the practical value of this model, explains some problems in SIP revealed by it, makes recommendations for solutions, and presents some directions for future work.

1 Introduction

In recent years, SIP has become an important and widely-used protocol for IP-based multimedia services. The SIP standard is documented in many IETF "Request for Comments" (RFC) documents, most notably [9]. Although this documentation is voluminous, its style emphasizes some points of view over others. One of the least-emphasized viewpoints is the state-oriented viewpoint, from which a user agent always has a dynamic protocol state, and in each protocol state some events can happen and some cannot.

To give an example of what a state-oriented viewpoint contains, the RFC on reliable provisional responses [8] says, "The UAS MUST NOT send a second reliable provisional response until the first is acknowledged." In other words, this state-oriented constraint says that in the UAS state of having sent a first provisional response, and not having received an acknowledgment of it, the UAS cannot send a provisional response.

As this example shows, state-oriented information is often present, but it is informal, scattered, and incremental. As an inevitable consequence of being informal, scattered, and incremental, it is incomplete.

The purpose of this paper is to improve the documentation of SIP by working toward state-oriented documentation that is formal, centralized, and complete. A necessary part of producing such documentation is to find the places where the standards documents are incomplete or erroneous, and to fix these problems. This paper reports on five such problems, and recommends changes that will eliminate them.

The documentation is primarily intended for people who build applications using SIP. A good state-oriented formal model of SIP would have many other uses in addition to that of user documentation, however. For example, the model could also serve as:

- a new technical viewpoint that might suggest improvements and best practices;
- a basis for checking proposed new extensions to SIP, to prevent inconsistencies and other problems;
- a basis for modeling network elements, such as proxies and back-to-back user agents;
- an aid to conformance and interoperability checking of SIP implementations.

To serve these purposes well, a model should be written in a language that is not only formal, but subject to automated analysis. Promela is a language for finite-state modeling of concurrent processes. It is well-suited to this modeling task for two reasons. First, the scope of the study is limited to aspects of SIP that can be expressed conveniently in Promela. Second, Promela models can be validated and verified using the model-checker Spin [3]. As subsequent sections will show, Spin provides "push button" verification of a large number of relevant properties of the model.

In related work, Bishop et al. provided a more complete and formal specification of TCP and UDP than was available in RFCs [1]. They were interested in *all* aspects of the behavior of these protocols, which necessitated the use of a wider-spectrum language, namely higher-order logic (HOL). HOL's automated support is a theorem prover, which is a powerful tool but unfortunately requires a great deal of time and skill to wield productively.

This paper reports on a first attempt to construct a state-oriented formal model of SIP. As such, it is inevitably limited in scope. Section 2 delimits what is and is not included.

Section 3 summarizes the formal methods used in this work. It gives a brief overview of model-checking, then explains the specification, verification, and validation techniques used on the model.

Section 4 presents the full SIP model, up to the scope given in Section 2. Section 5 presents two alternate models with slightly different properties. The paper concludes with recommendations and plans for future work.

2 Scope of the modeling

In this paper, all the models contain a SIP user agent on the client side (UAC) and a SIP user agent on the server side (UAS), modeled as concurrent processes. These processes communicate with each other only by sending and receiving messages.

As stated in Section 1, the primary purpose of the models is to assist builders of applications. For this reason, the chosen level of abstraction is that of the "transaction user" as defined in Section 17 of [9], and many phenomena that occur at lower levels of the SIP stack are not included. These phenomena include transaction timeouts, retransmissions, absorption of some messages, and *100(Trying)* messages. As a consequence, in the view of the models, transactions are reliable.

The models document SIP for a transaction user by representing everything that a transaction user can do while complying with the standards, and nothing that would violate the standards. Because no behavior of the models is erroneous, the models do not contain any handling of errors introduced by transaction users.

The primary focus of modeling is the control of media sessions. Thus the models include the offer/answer negotiation of media sessions, but are limited to covering only dialogs created by *invite* requests. *Subscribe* and *refer* requests can also create dialogs, but these requests are not considered. As a result, the dialogs in these models cannot be extended by embedded *subscribe* and *refer* dialogs, a phenomenon that is described in [10].

For purposes of simplicity and efficiency, most requests that are not concerned with media sessions are omitted. These requests are *subscribe* and *refer*, as noted above, and also *register, options, notify,* and *message*. The exception is the *info* request, which is not involved in media session negotiation, but is included because of its value for providing extensible signaling. For example, *info* requests and responses are used to control media servers [4].

For purposes of simplicity, a message carries at most one session description (SDP field). To evaluate the significance of this restriction, there is an example in [2] of a message with two session descriptions, one an offer and one an answer. The offer refers to an early media session, while the answer refers to an ordinary media session.

Finally, the models are coarse-grained in the sense that they usually do not distinguish between different messages of the same type. Thus most aspects of SIP that depend on message fields are not represented. This is the main reason why this study presents a narrower view of SIP than [1] does of TCP and UDP.

3 Semantics and analysis

3.1 A brief overview of model-checking

A Promela model has concurrent processes that communicate by sending and receiving messages. Messages in transit are stored in bounded, FIFO queues.

A model is intended to represent a range of possible behaviors. For this reason, execution of a model is highly nondeterministic. Within each process, in any state, several different steps may be executable. If there are executable steps in several different processes, then there is also a nondeterministic choice of which process to execute. A *trace* is the sequence of steps in one particular execution, so it is the result of many nondeterministic choices among possible next steps. A trace can be finite or infinite, because nonterminating models are valid and correct for some applications.

Model-checking is a form of analysis that attempts to explore all possible traces of a model, whether finite or infinite. It creates a finite state-transition graph in which each state is a state of the entire model (all processes and queues), and each transition is a possible execution step. Thus the entire graph represents all possible interleavings of executable steps.

A Promela model has a single initial state. A terminal state is one in which there are no possible execution steps, and thus no outgoing state transitions. A trace is a path through the state-transition graph, starting at the initial state. If the trace is finite, the path eventually ends at a terminal state. If the trace is infinite, the path loops through the graph forever, without entering a terminal state.

Often a model checker cannot construct a complete

state-transition graph, because the graph is too large. Because the Spin model-checker is a well-engineered tool, it offers many ways in which a user can optimize the checking for better results. When the state-transition graph can be completed, the number of transitions in it is a measure of the overall complexity of the model.

3.2 Model semantics

The purpose of our models is to describe all possible user-agent behaviors and how they can affect the other user agent in a dialog. In each state, the model should indicate which messages can be sent and which messages might be received.

These concepts will be illustrated with a very simple model that is a small fraction of the size of the real SIP models. To make it simple, the only message types are *invite, invSucc* (any 2xx response to an invite), *invFail* (any 3xx-6xx response to an invite), *ack*, *bye,* and *byeRsp* (any 200 OK response to a *bye* message). Furthermore, there are no re-invites. For this simple model, the UAC and UAS processes are defined as follows.

```
proctype UAC() {
bool endedc = false;
           reqc!invite;
inviting:  do
           :: irps?invFail -> goto preEnd
           :: irps?invSucc -> sakc!ack;
              goto confirmed
           :: reqs?bye -> brpc!byeRsp;
                          goto preEnd
           :: brps?byeRsp -> assert(false)
           od;
confirmed: do
           :: irps?invFail -> assert(false)
           :: irps?invSucc -> assert(false)
           :: reqs?bye -> brpc!byeRsp;
                          goto preEnd
           :: brps?byeRsp -> assert(false)
           :: reqc!bye -> goto byeing
           od;
byeing:    do
           :: irps?invFail -> assert(false)
           :: irps?invSucc -> assert(false)
           :: reqs?bye -> brpc!byeRsp
           :: brps?byeRsp -> goto preEnd
           od;
preEnd:    endedc = true;
end:       do
           :: irps?invFail -> assert(false)
           :: irps?invSucc -> assert(true)
           :: reqs?bye -> brpc!byeRsp
           :: brps?byeRsp -> assert(false)
           od
}

proctype UAS() {
bool acked = true;
bool endeds = false;
           reqc?invite;
invited:   do
           :: ackc?ack -> assert(false)
           :: reqc?bye -> assert(false)
           :: brpc?byeRsp -> assert(false)
           :: irps!invFail -> goto preEnd
           :: irps!invSucc -> acked = false;
              goto confirmed
           od;
confirmed: do
           :: ackc?ack -> assert(!acked);
              acked = true
           :: reqc?bye -> brps!byeRsp;
                          goto preEnd
           :: brpc?byeRsp -> assert(false)
           :: reqs!bye -> goto byeing
           od;
byeing:    do
           :: ackc?ack -> assert(!acked);
              acked = true
           :: reqc?bye -> brps!byeRsp
           :: brpc?byeRsp -> goto preEnd
           od;
preEnd:    endeds = true;
end:       do
           :: ackc?ack -> assert(!acked);
              acked = true
           :: reqc?bye -> brps!byeRsp
           :: brpc?byeRsp -> assert(false)
           od
}
```

The expression *reqc!invite* sends an invite to the queue *reqc*, which contains all requests sent by the UAC. (The other queues and their purposes are discussed in the next section.) A send expression such as *reqc!invite* is always executable unless the queue is full.

The expression *reqc?invite* receives an invite from the queue *reqc*; it is executable if and only if the queue is nonempty and the message at its head is an invite.

In each labeled state, the semantics of a *do* loop is to choose nondeterministically from among the executable guarded commands (executability is based only on the executability of the guard), execute the entire command, and repeat forever. Thus only a *goto* causes a change to another labeled state.

For a formal model, both *validity* and *correctness*

are important. A *valid* model is faithful to the needs and assumptions of its human readers. A *correct* model conforms to some separate formal specification of its intended properties. Because correctness is a relation between two formal objects, it can be verified formally, using model-checking or other techniques.

The models in this paper are intended to describe SIP rather than achieve a particular user goal, so their external specifications are weak. For the simple model, the only external specification is that every dialog should end, and both user agents should agree that it has ended.

To formalize this property, in the desired end state of a dialog, the two Boolean variables *endedc* and *endeds* are true. The variables are used in a linear-time temporal logic formula:

$$\Diamond\Box(endedc = true \wedge endeds = true)$$

This formula is called a *stability property*, because it says that in any trace, there is eventually ($\Diamond$) a point where the variable assertion becomes invariantly true ($\Box$) for the rest of the trace. Spin verifies that the simple model satisfies this property specification.

There is a much more useful form of specification for this study, however. Assertions on state variables can be interspersed with the executable code in the model. Each such assertion is a specification that, at the control point where the assertion is written, the assertion's predicate must always be true.

For example, the simple model of the UAS sets the Boolean variable *acked* to false when it sends an *invSucc* message. Wherever in the UAS code an *ack* message is received, an assertion states that *acked* is false, and an assignment sets it to true again. The assertions specify formally our expectations about when acks should and should not arrive. Spin reports any assertion violation as an error, so complete model-checking without any such errors is a verification that all of the assertions are satisfied. The real SIP models contain a large number of assertions.

3.3 Validity of SIP models

For our purposes, validity is as important as correctness. The most important aspect of validity is whether a model agrees with the RFCs, and this can only be decided by consensus.

Although validity is ultimately an informal property, validity in this context has formal aspects that model-checking can help us with. A valid model should include no illegal or impossible behavior, and should represent all legal and possible behaviors in a clear and obvious way. This raises subtle issues, because send events cause receive events, send and receive events create new dialog states, and new dialog states create the need to describe the events that can occur during them. In this formal sense, a valid model is a "fixed point" that contains all the consequences of its own behavior.

This section presents four formal properties that contribute to model validity and can be checked, sometimes with the help of Spin. All the models in this study were checked for these properties, which are:

- The behavior of message queues matches what is possible in implementations.
- Send events are always immediately enabled when execution control gets to them.
- A message can be received as soon as it arrives.
- The model has no unreachable code.

Each of these characteristics will be discussed in turn.

It is obvious that the behavior of message queues in the model should match what is possible in implementations. All SIP user agents must implement both UDP and TCP for signal transport ([9], Section 18). UDP makes no ordering guarantees, so messages need not arrive in the order they were sent. Promela queues, on the other hand, are FIFO. To make their behavior as unconstrained as UDP, it is necessary to have a separate queue for each message type in each direction. It is not necessary to worry about the ordering of different instances of the same message type, because the models cannot distinguish between instances.

For example, in this simple model two of the message types are *ack* and *byeRsp*, and each type has its own queue. The queues are *ackc* (for acks from UAC), *brpc* (for bye responses from UAC), and *brps* (for bye responses from UAS). The model is too simple to need acks from the UAS.

There can be exceptions to this basic rule when there is specific information to justify them. For example, the SIP standard requires that requests have sequence numbers so that they are received and processed in the same order that they were sent. Because of this, all the requests in each direction of the dialog can share a FIFO queue. The queue *reqc* carries requests from the UAC to the UAS, while *reqs* carries requests from the UAS to the UAC.

As another example of an exception, there can only be one final response in transit in each direction at a time, so *invSucc* and *invFail* events can share the queue *irps* (invite response) from UAS to UAC. In this simple model there are no final responses from the UAC to invites. This accounts for all the message queues in the simple model.

The next important validity property is that send

events are always immediately executable when execution control gets to them. For example, in the *confirmed* state of UAC, as soon as *reqs?bye*, is executed, *brpc!byeRsp* should be executable. Also, between execution of any two guarded commands in the *confirmed* state, the send event *reqc!bye* should always be executable. This characteristic is important for validity because this is what a reader of the model would normally expect and assume.

The only reason that a send might not be executable is that its queue is already full. It is awkward to check for blocked sends directly, but there is an easy indirect way. First, do an exhaustive check of a model with proposed queue sizes. Then increase the sizes of all the queues and check again. If the number of state transitions in the second check is the same as in the first, then the additional space in the queues has not added any new behavior to the model. This proves that the proposed queue sizes are adequate to prevent blocking of send events. Because the example model is so simple, all its queue sizes are 1.

The third validity property is that a message can be received as soon as it arrives. For example, in the *confirmed* state of UAS the following receive events are specified: *ackc?ack, reqc?bye,* and *brpc?byeRsp*. It should not be possible for there to be some other message waiting in one of the UAS queues to be received. If there were, the model would not be giving the reader a clear and complete description of the possible next steps.

This property can be achieved simply by constructing the model so that in every user-agent state, there is a receive event for every message type in every queue directed to that agent. The model above is constructed in this way, with the single exception of *invite* messages. Each dialog begins with a unique, single invite, so it is easy to determine by inspection that no invite can ever be received later.

Finally, the model should have no unreachable or useless code. Unreachable code deceives the reader into seeing more behavioral possibilities than there really are.

In a guarded command, each guard is always technically reachable because Spin checks to see if it is executable. However, if the guard is never executable, the entire guarded command is useless. To determine whether a guard is ever executable in any trace, it is necessary to put a statement after it in the body of the command. If the guard is never executable, then the body of the command is not reachable, which will be reported by Spin. In the example model, every guarded command has a body.

For those receive events that we believe are never executable in any trace, it is convenient to use *assert(false)* as a body. If that body is ever reached, Spin reports an error. For example, neither user agent can receive a *byeRsp* message in an *inviting* or *invited* state. After a model has been checked completely, it can be cleaned up for readability by removing all the guarded commands with *assert(false)* as their body.

For those receive events that should be executable sometimes but require no further action, it is convenient to use *assert(true)* as a body, because it has no effect on execution. For example, the UAC can receive *invSucc* in its *end* state, when the order of *bye* and *invSucc* from UAS has been reversed in transit, but the message comes too late to matter. After a model has been checked completely and all such bodies are known to be reachable, they can be removed for readability.

4 Basic model of invite dialogs

The basic model of *invite* dialogs in SIP is too large to print in this paper (441 lines without comments or whitespace). It can be viewed at `http:// www.research.att.com/ ~pamela/ sip.html`. This section presents its structure and interesting characteristics.

In addition to the six message types introduced in Section 3.2, this model uses nine new ones. There are new request types *prack, update, cancel,* and *info*. Provisional responses are partitioned into unreliable ones *unProv* and reliable ones *relProv*. The *200(OK)* responses to *prack* and *update* are named *prackRsp* and *updSucc,* respectively. Finally, the failure response to an update (a 491) is named *updFail*.

The channel partition is similar to that explained in Section 3.3. There is one channel in each direction for requests; *invSucc* and *invFail* share a channel, as do *updSucc* and *updFail*, for the same reason. All other message types have their own channels.

4.1 Omissions

A few SIP phenomena are omitted from the model, for the reasons presented here.

First, responses to *cancel* requests are not included, because they are required for transport reliability only, and make no difference to the transaction user. This omission may seem a bit strange, because cancels can succeed or fail. However, a cancel fails only when the dialog it is meant to destroy is already gone (for some other reason). When the canceling UAC receives notice of failure of the cancel, it has nothing to do.

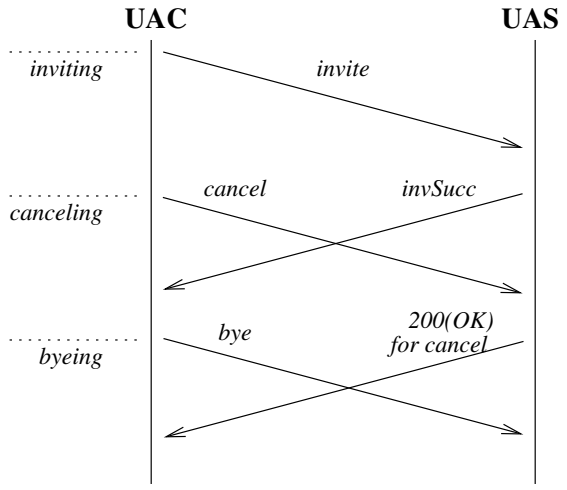

Figure 1: A scenario in which canceling has no effect. Labels on the left are UAC states in the model.

It is much more important that a *cancel* request can succeed without having the desired effect ([9], Section 9). This scenario is illustrated by Figure 1. As the figure shows, in the *canceling* state the UAC responds to invSucc with bye, because that is the only way to end the dialog. The *200(OK)* response to the cancel is irrelevant to the UAC at the application level.

Second, a UAS never sends a subsequent relProv until it has a received a prack for the previous one. This is recommended by the standard [8], and the model follows this recommendation.

Third, a user agent does not send *update* requests within confirmed dialogs, using re-invites instead. This is recommended by the standard [6], and the model follows this recommendation.

Fourth, unreliable provisional responses do not have SDP fields. It seems clear that reliability is required for offer/answer exchange, and that reliability is now available in the form of reliable provisional responses [8].

4.2 State variables

In addition to having almost the same labeled dialog states as the simple model in Section 3.2, this model has a large number of state variables.

Most important of all, the model records each UA's current media state in a variable *media*. The four possible values of this variable are *noFlow* (there has been no offer/answer exchange), *offering* (this UA has made an offer but has not yet received an answer), *offered* (this UA has received an offer but has not yet answered), and *flow* (the most recent offer/answer exchange is completed). Each message whose type can ever contain an SDP field contains a value *offer, answer,* or *none*. This value indicates whether the message is carrying an offer, an answer, or neither, respectively.

In the UAS, the variable *media* is initialized to *noFlow*. The UAS receives the initial *invite* message with the following code:

```
reqc?invite,sdp;
if
:: sdp != none -> assert(sdp == offer);
   media = offered
:: sdp == none; assert(true)
fi;
```

When the invite is received, the local variable *sdp* receives the value of the SDP field in the message. An *if* statement in Promela executes exactly one of its guarded commands (and blocks if no guard is executable).[1] In this case the *if* statement is a case statement on whether the received value of *sdp* is *none* or not. If it is not *none*, then it is an offer, and *media* is updated to *offered*; if it is *none*, *media* continues to have the value *noFlow*.

As we shall see, the *media* variable is critical for understanding the state of a UA, and indispensable for interpreting received messages, because a UA cannot tell from syntax alone whether an incoming SDP field is an offer or an answer. This means that any correct implementation of a SIP user agent must have a similar state variable.

In the UAC, there are state variables whose value is important when the UAC is in the labeled state *inviting*, after sending the initial invite and before receiving a final response to it. One of these variables is the Boolean *dialog*, which starts false and becomes true when the dialog is established. The other *inviting* state variables are explained in Sections 4.4 and 4.5.3.

In the UAS, there are state variables whose value is important when the UAS is in the labeled state *invited*, after receiving the initial invite and before sending a final response to it. These variables include *dialog* (as in the UAC) and the Boolean *relOut*. As mentioned in Section 4.1, reliable provisional responses are sent sequentially. The variable *relOut* is true when there is a relProv outstanding, so when it is true a relProv cannot be sent. The other *invited* state variable is explained in Section 4.4.

Both UAC and UAS have the same variables that are important in the labeled *confirmed* state, when the dialog has been confirmed and not yet torn down.

[1] In Promela syntax, the arrow separating guard from command can also be written as a semicolon. The actual models use semicolons.

These variables include the Booleans *reInviting* and *reInvited*, which indicate whether the user agent has an uncompleted outgoing or incoming re-invite transaction, respectively.

When a user agent in a confirmed dialog receives a re-invite, its *reInvited* variable becomes true. In the large *do* loop that defines a *confirmed* state in the UAS, this is one of the guarded commands:

```
:: reInvited -> reInvited = false;
                ackDiff++;
   if
   :: media == flow -> irps!invSucc,offer;
                       media = offering
   :: media == offered ->
                       irps!invSucc,answer;
                       media = flow
   :: else -> assert(false)
   fi
```

Whenever *reInvited* is true, the user agent can choose to make a final response to the re-invite. The variable can remain true, and the re-invite can remain pending, for as long as the user agent chooses. This models the fact that a user agent is not required to make an immediate response to a re-invite.

There are two other *confirmed* state variables, which are explained in Sections 4.5.1 and 4.5.2.

In both user agents, there are variables that keep track of expected acknowledgments or responses, as *acked* does in the simple model. For a request type that cannot be sent until the last request of that type has been acknowledged, namely *update*, the corresponding variable *updRsped* is Boolean. For request types that can have multiple responses outstanding, such as prack, an integer such as *prackRspDiff* holds the number of requests sent minus the number of responses (prackRsps) received.

Finally, because *info* requests and unreliable provisional responses (*unProv* messages) are almost completely unconstrained, a full model might loop forever sending these messages, thereby overflowing the message queues and blocking sends unexpectedly. To prevent this, there are Boolean variables *infoSent* and *unProvSent* to limit each user agent to sending at most one message of each type. This small number is sufficient because receiving these messages has no effect on user-agent state; the only purpose of modeling them is to document when they can be sent and received.

4.3 Assertions

Writing and checking assertions is the heart of the modeling and verification process. Assertions find errors, deepen understanding, and provide reassurance.

In this model, the intention of the assertions is to elucidate the value of every relevant state variable at every important point in execution. This can be seen in the (rather large) guarded command in the UAC that receives a final *invSucc* response to the original invite:

```
:: irps?invSucc,sdp;
   assert(!relProvBuffered); dialog = true;
   if
   :: media == noFlow;
      assert(!initOffering && sdp == offer);
      ackc!ack,answer; media = flow
   :: media == flow;
      assert(!initOffering && sdp == none);
      ackc!ack,none
   :: media == offering && sdp == none;
      assert(!initOffering); ackc!ack,none
   :: media == offering && sdp != none;
      assert(initOffering && sdp == answer);
      ackc!ack,none; media = flow
   :: media == offered; assert(false)
   fi;
   goto confirmed
```

If this command is executable, then the UAC is in the *inviting* state, and the relevant state variables are *media, dialog, sdp, initOffering* (the meaning of which is explained in Section 4.4), and *relProvBuffered* (the meaning of which is explained in Section 4.5.3).

If this command is executable, then *relProvBuffered* should be false. The initial assertion guarantees this.

If this command is executable, the dialog may or may not have been established. If it is not established, receiving invSucc establishes it, which we see in the assignment *dialog = true*.

If this command is executable, *media* should not have the value *offered*, which would mean that the UAS had sent an offer to the UAC in an *update* or *relProv* message, and the UAC had not yet responded to it with an answer. The *assert(false)* guarantees that the media state is not *offered*, and that our understanding is correct.

The embedded *if* statement is a case statement on the value of *media* (and sometimes whether *sdp* is *none* or not). For each of the reachable cases, there is an assertion about what the current values of *sdp* and *initOffering* must be. In summary, for each path through this command, we know by inspection the final value of each relevant state variable.

It is easy to deduce that, when the UAC enters the *confirmed* state, *media* must be *offering* or *flow*. If you become curious about how its value might be

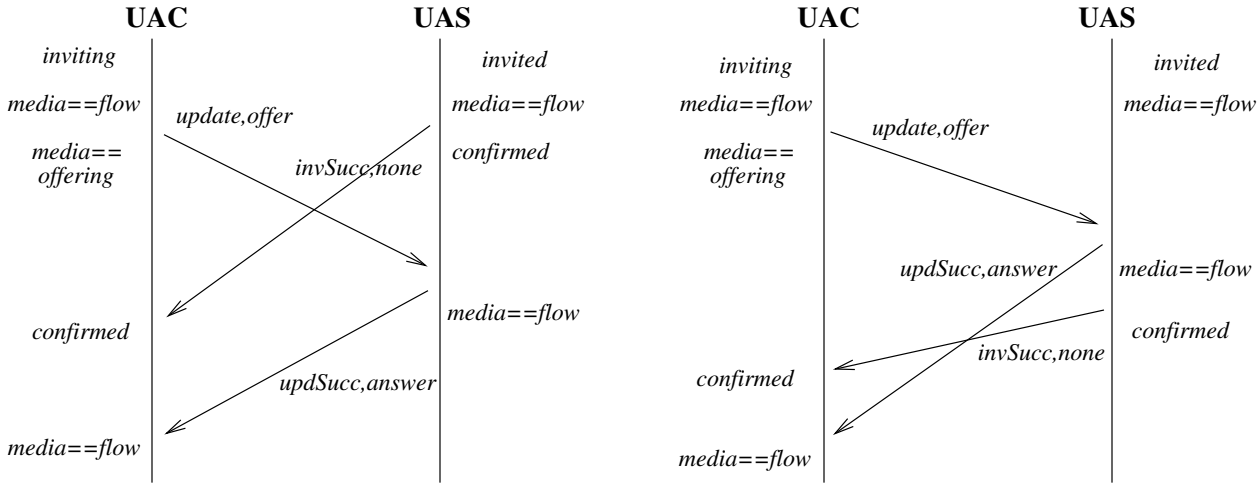

Figure 2: Scenarios in which the UAC enters the *confirmed* state with *media == offering*.

offering, it is fairly easy to discover from the model that there are two scenarios in which an update or updSucc is received by a user agent in the *confirmed* state. These scenarios are shown in Figure 2.

In the UAS code that produces the scenarios above, receiving a successful update and sending its response are in the same guarded command. They cannot be separated by the execution of any other guarded command. This reflects the fact that a user agent is required to respond to an update immediately, and contrasts with the modeling of re-invites as presented in the previous section.

There is a possible scenario very similar to that on the right side of Figure 2, in which the offer is carried in a prack and the answer is carried in a prackRsp.

4.4 Reliable provisional responses

Reliable provisional responses can be quite complex to use. To help clarify them, Figure 3 is a finite-state machine showing all the ways they can be used to carry SDP fields.

In each oval state of the figure, the top label is the current media state of the UAC, while the bottom label is the current media state of the UAS. In the final state, the dialog has been confirmed.

It is important to recognize the limitations of this figure. It is a synchronous model, in which sending and receiving a message are one atomic event. All messages from the UAS are shown as sent on the channel *uas*, while all messages from the UAC are sent on the channel *uac*. In reality, there are many other behaviors in which these send and receive events are interleaved with other events.

Note that in some states of this machine, the UAS cannot send *invSucc* to confirm the dialog, even though it has received an initial *invite* request and has not yet made a final response. The reason is that there is an offer/answer negotiation in progress that must be completed first. When the Promela model is in these states, the UAS's state variable *relOut* is true, and its state variable *canSucc* is false. After the UAS receives a prack and sends a prackRsp, its *relOut* is false and its *canSucc* is true.

Another limitation of the figure is that it does not include *relProv* messages without SDP, which can be sent by the UAS whenever *relOut* is false. The subsequent *prack* and *prackRsp* messages cannot have SDP, either. Sending a *relProv* message without SDP sets *relOut* to true (until it is acknowledged) but does not set *canSucc* to false.

In the model, neither user agent can send an *update* message unless its media state is *flow*. In the RFC on updates [6], the same constraint is stated in 31 lines of dense text. This text is attempting to enumerate all the ways that offers and answers can be interleaved when carried in initial invites, relProvs, pracks, prack responses, updates, and update responses. Its purpose is to say that there must have been an initial offer/answer exchange, and that there are no offer/answer exchanges in progress, which is the exact meaning of *media == flow*.

It is easy to see from Figure 3 that SDP in a relProv is an answer if and only if there is an unanswered offer from the initial invite. Because (with fully interleaved behavior) the UAC cannot infer this from any of its other state variables, it has a Boolean state variable *initOffering* for exactly the purpose of determining

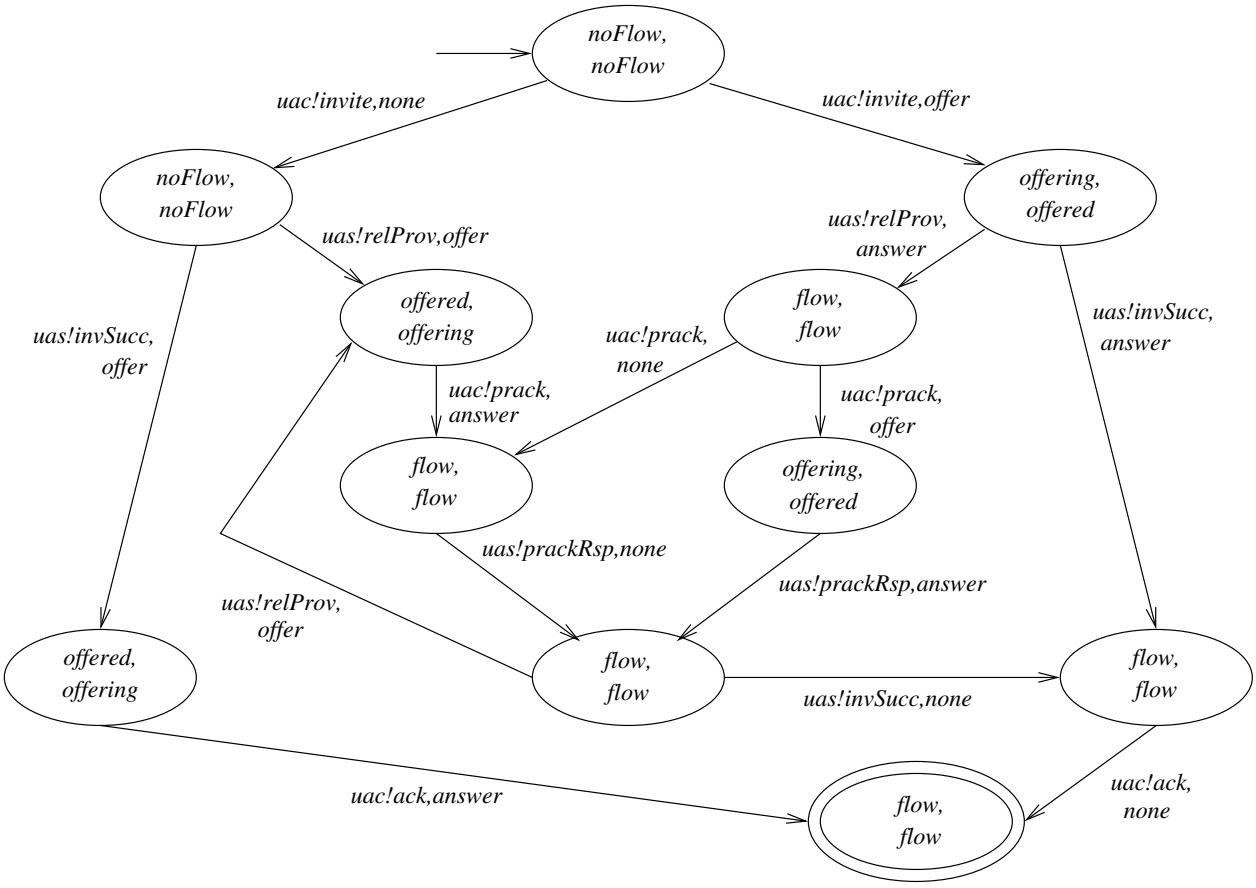

Figure 3: A synchronous model of reliable provisional responses with SDP.

whether SDP in a relProv is an offer or answer.

A reader might wonder about the following statement in [8], Section 4: "The UAC MAY acknowledge reliable provisional responses received after the final response or MAY discard them." What if the relProv is important? What if it contains SDP?

Assertions in the model can reassure the nervous reader. In the *confirmed* state of the UAC model, we see the statement:

```
:: rlps?relProv,sdp -> assert(sdp == none)
```

Thus model-checking has verified that the late relProv cannot carry SDP, and can be safely ignored.

4.5 SIP problems

Model checking uncovers some apparent problems with the standards documents. The model incorporates work-arounds for all the problems found. This is necessary because the problems show up as errors in model-checking. Without the work-arounds, known errors prevent full model exploration and the discovery of further errors.

4.5.1 Dialog establishment and confirmation

An *invite* dialog is established by the first provisional or final success response to the invite ([9], Section 12.1). There is an implicit assumption that if a message is part of a dialog, and the message does not establish the dialog, then the message is received after the dialog is established in the agent where the message is received.

In contradiction to this implicit assumption, *info* messages from the UAS to the UAC can arrive at the UAC before the dialog they belong to is established. This can occur because *info* messages from the UAS are requests, the dialog is established by responses traveling in the same direction, and requests are unordered with respect to responses traveling in the same direction. In the model, this is simply noted, and the requests are handled as if the dialog were established.

It is obvious that an invite dialog becomes *confirmed* in a UAC when the UAC receives a final response to the initial invite. It is not so easy, however, to find in [9] whether a dialog becomes confirmed in a UAS when the UAS sends a final response or when it receives an ack to it.

This is quite important. For example, Section 14.1 of [9] applies the terms *completed, confirmed,* and *terminated* to invite transactions, and uses them to specify how overlapping re-invite transactions should be avoided.

These terms are formally defined in Figures 5 and 7 of Section 17 of [9]. According to these figures, on the UAS side, a failing invite transaction becomes *completed* when the final response is sent, and *confirmed* when the ack is received. According to these figures, a successful invite transaction is never in a *completed* or *confirmed* state, going directly from *proceeding* to *terminated*. The common-sense interpretation is that these figures are wrong with respect to the definition of these terms, and that *completed* and *confirmed* have the same meanings for both successful and failing transactions.

To avoid confusion, the UAS model has been described as having a labeled state *confirmed*. In actuality it has a labeled state *completed* and a Boolean variable *confirmed* to represent the state of the initial invite transaction and therefore the dialog as a whole.

4.5.2 Re-invites

Figure 4 shows a scenario in which the UAS issues two re-invites. The first re-invite does not contain an offer, so the invSucc response contains an offer and the ack contains an answer.

This scenario is not covered in the RFCs, and it violates the clear intentions that offer/answer exchanges are sequential, and that a successful re-invite can be processed as soon as it is received. The second re-invite cannot be processed in the normal manner because it arrives at the UAC during an unfinished offer/answer exchange. The second re-invite cannot fail, because the cause of the invFail will not be understood by the UAS.

As in the previous section, the cause of the problem is that there is no enforced ordering on the *ack* message and the second *invite* request.[2] The two were sent in a proper order, but arrived in the wrong order.

The work-around used in the model is to buffer the second re-invite until the ack arrives and is processed, then handle the re-invite. This causes no additional problems because a user agent is not required to respond immediately to the re-invite. The Boolean state variable *reInviteBuffered* is true when a re-invite is buffered in the user agent.

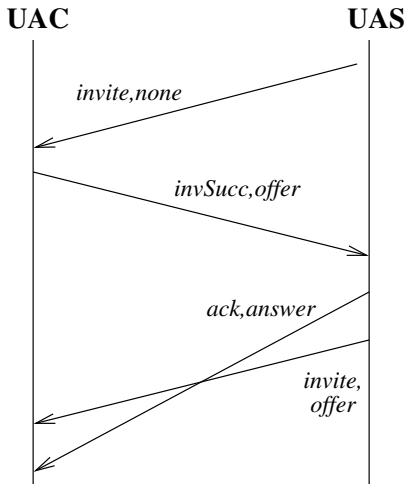

Figure 4: A scenario with a late acknowledgment to a re-invite.

The problem is in fact more general, because the late acknowledgment could just as easily be a *prackRsp, updSucc,* or *updFail*. However, not all late acknowledgments are expected to carry answers, so not all of them require delay. Consequently, the UAC model diagnoses the condition by the media state when the re-invite arrives:

```
:: reqs?invite,sdp ->
   assert(!reInviteBuffered && !reInvited);
   if
   :: reInviting -> irpc!invFail,none
   :: !reInviting && media != flow ->
      reInviteBuffered = true; bufsdp = sdp
   :: !reInviting && media == flow ->
      reInvited = true;
      if
      :: sdp != none -> assert(sdp == offer);
                        media = offered
      :: sdp == none -> assert(true)
      fi
   fi
```

If the UAC is not re-inviting when the re-invite arrives, and the media state is anything but *flow*, then a late acknowledgment carrying an answer is expected. After the *ack, prackRsp, updSucc,* or *updFail* arrives, if the media state is *flow*, then the re-invite is unbuffered and processed.

[2] Although the *ack* is technically a request, like the *invSucc* it bears the sequence number of its invite.

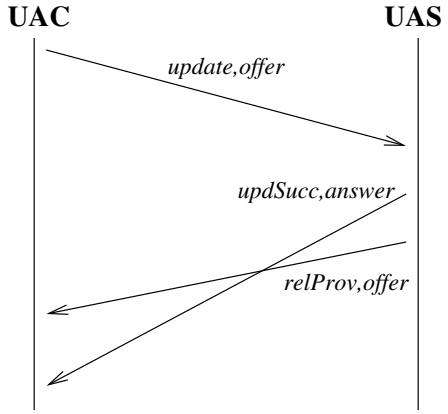

Figure 5: A scenario with a late acknowledgment to an update.

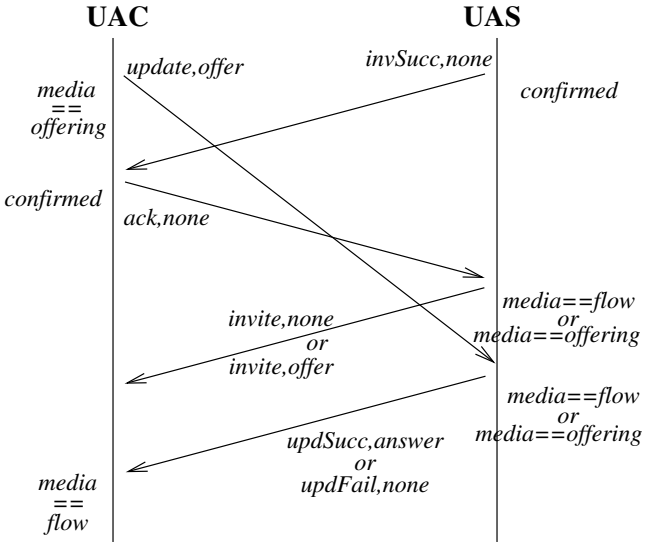

Figure 6: A scenario with a race between an update and a re-invite.

The same can happen within a UAS, but the only possible type of late acknowledgment is *ack*.

In the model with its work-around, if a user agent is expecting an *ack* message without SDP, the user agent does not wait for it. When it arrives, it has no effect on the model state except to enable the user agent to sent its own re-invites. Thus, in theory, there could be any number of *ack* messages with *sdp* == *none* in transit between the user agents.

To prevent state explosion in the model checking, the size of channels for *ack* messages is limited arbitrarily to 2. This means that a send of an *ack* message can actually block. As the blockage is an artifact of model checking, however, and never causes deadlock, it seems safe enough.

Because there can be two *ack* messages in transit, the model tracks *ackDiff*, the difference between the number of *invSucc* messages sent and acks received. All of the observations in these last three paragraphs are equally true of *prackRsp* messages in the model.

4.5.3 More late acknowledgments

Figure 5 shows a scenario very like Figure 4. In Figure 5 the dialog is not yet confirmed, and the late acknowledgment is to an update. The offer could also have been sent in a prack, in which case the late answer would have been sent in a prackRsp.

As with the previous problem, the relProv cannot be processed normally because its offer arrives during an unfinished offer/answer exchange. As with the previous problem, the RFCs do not mention this possibility.

For both update and prack cases the work-around is the same, which is to buffer the *relProv* message. The Boolean state variable *relProvBuffered* is true when a relProv is buffered.

This is arguably a more serious problem than the previous one, because buffering a relProv is less desirable than buffering a re-invite. A user agent is supposed to respond to a *relProv* message immediately, while it is allowed to delay its response to a re-invite.

4.5.4 A race condition

Figure 6 illustrates a race condition between an update and a re-invite. There are actually two scenarios represented, one in which the re-invite carries SDP, and one in which it does not. All the top parts of "or" labels go together in one scenario, while all the bottom parts of "or" labels go together in another scenario.

There is no standardized resolution to this race condition. Yet the UAC cannot process the re-invite until it receives a response to its update, which will return its media state to *flow* whether the update has succeeded or failed.

The model resolves the race condition with the same work-around used in Section 4.5.2, which is to buffer the re-invite until it can be processed.

4.5.5 Another race condition

In an unconfirmed dialog, there can be a race condition between a *relProv* message with an offer from the UAS, and an *update* message from the UAC. The resolution of this race condition is not standardized.

Performance Measure	Basic Model	FIFO Model	Pruned Model
lines of reachable code	404	300	266
state vector (bytes)	200	88	88
depth reached	6,165	1,068	464
state transitions	780,538,240	9,043,855	3,429,348
memory usage (Mbytes)	20,904	308	105
elapsed time (seconds)	4,200	38	13

There is a usual way of resolving race conditions in SIP. In both re-invite/re-invite races and update/update races, both sides receive an invFail (4xx) response, then are allowed to retry with differing timing constraints. This usual approach does not seem to be available in the case of a relProv/update race, because a relProv is not a request, and there is no failure response it.

In the model, the work-around is that the *relProv* message always wins the race. In the UAS, the update fails and an *updFail* response is sent. In the UAC, the relProv is buffered until *updFail* is received, after which the relProv is handled normally.

4.6 Results of model-checking

The Promela model can cycle forever, so unlike the simple model in Section 3.2, it was not checked for termination.

The table (left column) shows the performance of exhaustive search of the state-transition graph on a Sun Solaris M9000 SMP machine with dual-core 2.4 GHz SPARC processors.

"Lines of reachable code" refers to the size of the Promela code (without comments or whitespace) after unreachable code (see Section 3.3) has been removed for readability. The more-readable version of the basic model is also available at `http://www.research.att.com/~pamela/sip.html`.

"State vector" is the amount of memory required to represent each state of the model. "Depth reached" is the number of steps in the longest path through the graph that does not repeat a state. "State transitions" is the number of edges in the graph, and is the best measure of model complexity. The memory usage was improved by compression, and would have been 88,896 Mbytes without it.

The elapsed time varies from run to run, so it is less reliable than the other numbers. The value of compression in minimizing memory usage seems to outweigh its time penalty.

5 Alternate models

This section presents two alternate versions of the basic model. All the models, in both model-checked and readable versions, are available at `http://www.research.att.com/~pamela/sip.html`. For comparison, the table reports their sizes and performance numbers.

5.1 The FIFO model

The first four of the five problems with the basic model all have the same underlying cause: the fact that SIP messages traveling from one user agent to the other need not arrive in FIFO order. This is obvious for the first three problems.

It is less obvious for the fourth problem, which presents itself as a race between messages traveling in opposite directions, but is nevertheless true. As we can see in Figure 6, the update and ack from the UAC arrive out of order. If they could not arrive out of order, then the UAS would receive the update before it sent the re-invite (which is enabled by receiving ack), so there would be no race.

The FIFO model removes all these problems by using only a single Promela channel for all the messages traveling in one direction.

The FIFO model is smaller than the basic model in every measure, for two main reasons. First, the work-arounds required for SIP Problems 1-4 have been removed. Second, FIFO ordering constrains the behavior much more. For example, in most states the FIFO model can receive fewer different message types.

According to the most important measure, which is the number of state transitions, the complexity of the FIFO model is 1.2% of the complexity of the basic model. Its analysis requires 1.5% of the memory to analyze the basic model, and 0.9% of the time to analyze the basic model.

The FIFO model is a faithful representation of an *invite* dialog in which each user agent uses a single TCP connection, or a sequence of TCP connections, to send messages to the other user agent. It does not matter whether messages traveling in opposite directions use the same TCP connection or not. In

either case, TCP ensures that the messages in one direction are received in FIFO order.

A signaling implementation based on UDP, or on a pool of TCP connections, cannot be guaranteed FIFO. This will be discussed further in Section 6.2.

5.2 The pruned model

Reliable provisional responses [8] were added to SIP before *update* requests [6]. Reliable provisional responses are sent by the UAS only. So that the UAC would have some capacity to modify media sessions before receiving a final response, the RFC on reliable provisional responses gives the *prack* request the ability to carry an offer, as shown in Figure 3.

Update requests allow a UAC to modify media sessions in an unconfirmed dialog, without waiting to receive a reliable provisional response to which it can respond with a prack (provided that the media state has advanced beyond *noFlow*). Thus the presence of updates makes offers in pracks redundant and their extra complexity unnecessary. In the "pruned" model, this redundant capability has been removed.

Similarly, any media modification that a UAS might attempt with an *update* request could just as easily be performed by sending a reliable provisional response (unconfirmed dialog) or a re-invite (confirmed dialog). Thus the use of updates by UASs is redundant and its extra complexity unnecessary. In the "pruned" model, this redundant capability has also been removed.

The "pruned" model is the same as the FIFO model, except for these removals. Based on the number of state transitions, the complexity of the pruned model is 38% of the complexity of the FIFO model. Its analysis requires 34% of the memory to analyze the FIFO model, and 34% of the time to analyze the FIFO model. Based on the number of state transitions, the complexity of the pruned model is 0.4% of the complexity of the basic model.

The significant reduction in complexity caused by the relatively minor change from the FIFO model to the pruned model is an excellent demonstration that the size of the state-transition graph grows exponentially.

6 Recommendations

6.1 Why complexity matters

It seems abundantly clear that modeling and model-checking SIP is a worthwhile thing to do. The investment is small compared to the thousands of person-hours that have been spent on the SIP RFCs. Yet it reveals previously undiscovered problems, and provides a useful kind of documentation that is not otherwise available.

It seems almost as clear that the complexity of the resulting model is significant. Every line of Promela code represents some case that an application developer must be aware of and must take into account. Every state and transition in the state-transition graph represents behavior that can occur "in the field" and interact with other layers of network and software. Finally, if a model becomes too complex it can no longer be model-checked, given practical limitations on time and memory.

So far all the SIP models are easy to check on a powerful machine. However, it is important to remember that the models represent only a small subset of SIP and its interesting component configurations, and that growth of complexity is exponential.

This argument can be quantified by relying on an analogy. Another research project [12] entailed modeling and model-checking a new protocol that is similar to SIP in its purpose and functions.

In that study, the models fall into two categories. There are models containing only two user endpoints, which means that they have the same configuration as the models in this paper. There are also models containing two user endpoints with a third process between them. This third process is analogous to a back-to-back user agent in SIP.

It is interesting to compare a pair of models in [12], one from each of the two categories above, with all other factors in the pair held constant. The jump from checking endpoints alone to checking endpoints with a back-to-back user agent causes the number of state transitions, the memory, and the time to grow by factors of 800, 300, and 1000, respectively. These are round numbers obtained by averaging many comparisons.

If we apply these numbers to the models here, we predict that the basic model, extended with a back-to-back user agent between the UAC and UAS, would have 620 billion state transitions. Model-checking it would require 6 terabytes of RAM and 1200 hours, or the equivalent with virtual memory used and swapping time added.

This is only an analogy. The protocols and models in the two studies are different (although balanced in the sense that each has sources of complexity that the other does not have). Nevertheless, if the analogy holds at all, model-checking the basic model with a back-to-back user agent is not practical.

6.2 What to do about it

The overall complexity of SIP can be reduced considerably, and the five problems reported in Section 4.5 eliminated, by standardizing the pruned model as the definition of *invite* dialogs. This proposal does not reduce the functionality of SIP in any way.

The most important issue is that of FIFO signaling. Even if we assume that TCP is used for signal transport, there are many open questions concerning the number of TCP connections within a dialog, which messages are sent in each, and whether they are overlapped or sequential in time. This appears to be a complex issue, with many ramifications for performance and security [5].

A common interpretation of the recommendations in [9], Section 18, is that an invite dialog should have at most two TCP connections at any one time, one for the transactions initiated by the UAC, and one for the transactions initiated by the UAS. Unfortunately this recommendation does not produce FIFO signaling; two messages traveling in the same direction can still arrive out of order. For example, a request from the UAS and a response from the UAS both travel from the UAS to the UAC, but they travel in different connections because the response belongs to a transaction initiated by the UAC. Categorized by message types, Section 4.5 describes 7 non-FIFO scenarios, of which this recommendation eliminates 5, leaving 2 still possible.

To repeat the conclusions of Section 5.1, FIFO signaling is guaranteed if all of the *messages* (not transactions) from one user agent to the other use the same TCP connection, or a sequence of them (with no overlapping in time). It would be highly advantageous to the SIP community to settle on some way of doing this that meets reasonable performance and security requirements.

The other issues are more straightforward. Current recommendations that should be strengthened into requirements, from Section 4.1, are:

- A UAS does not send SDP in unreliable provisional responses [8].
- A UAS does not send a new reliable provisional response until it has received a prack from the last one sent [8].
- A user agent does not send *update* requests within confirmed dialogs [6].

The pruned model relies on the following further changes:

- A *prack* message does not contain an offer.
- A UAS does not send *update* messages.

Finally, all the models require a resolution to Problem 5, which can be summarized as: if there is a race between a *relProv* message with an offer from the UAS, and an *update* message from the UAC, then the update always fails.

7 Conclusion

This paper has presented three formal, state-oriented models of *invite* dialogs in SIP. It has discussed five places where the SIP standards are incomplete or erroneous, and proposed solutions to all the problems they cause.

As a final observation on the standards, it has already been noted that the update RFC [6] recommends using re-invites instead of updates in confirmed dialogs. The recommendation is explained as follows: "This is because an UPDATE needs to be answered immediately, ruling out the possibility of user approval. Such approval will frequently be needed, and is possible with a re-INVITE."

In fact, there is an additional way that re-invites are more powerful than updates: a re-invite can be sent with no offer, which requires its recipient to send an offer in the successful response, if there is one. There are numerous examples to show that this capability is an important ingredient in third-party call control [7, 11].

Most of the scope restrictions in Section 2 do not seem to compromise the validity and usefulness of the models presented here. In each case, it is fairly easy to see how broadening the scope would extend the models or necessitate a complementary model.

The exception is the carrying of multiple session descriptions in a single message, as in [2], which completely invalidates this modeling. The RFCs cited in this paper contain a large number of constraints concerning when offers can be sent, when answers can be sent, and which messages they can be sent in (all of which are reflected in the Promela models). It is not at all clear how the presence of multiple session descriptions in a message would interact with these constraints.

No argument or proposal from a single viewpoint can be conclusive. Many factors, with many stakeholders and at many levels of abstraction, affect the current SIP and its possible futures. Nevertheless, formal modeling of SIP has proved its value as a source of relevant information. It should be pursued as a way of shedding light on other aspects of SIP.

There are many ways that the models could be improved and exploited in their role as documentation. For example, the model could be cross-indexed to RFCs, there could be tools that construct—from a user's query—a particular trace allowed by a model,

and so forth.

Of all the possible uses for a formal model, the one worst served by the current technology is checking the conformance of implementations. There are two problems. First, there must be a suitable formal representation of the implementation that can be compared to the specification model. Second, even with such a representation, the scale of the verification could be beyond current feasibility (even though hardware implementations are now routinely verified).

Bishop et al. dealt with these problems, in their work on TCP and UDP, by building a special-purpose checker for captured real-world traces from implementations [1]. Capturing a trace is a lightweight way to get a formal representation of an implementation's behavior. The checker performs symbolic evaluation to determine whether a trace satisfies the specification. Although the results are limited by the quality of test coverage, this is a big advance in conformance technology, and one that might be applicable to SIP.

Acknowledgments

Most of what I know about SIP I learned from my colleagues Greg Bond, Eric Cheung, Hal Purdy, and Tom Smith. They are not responsible, however, for my remaining misconceptions.

References

[1] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP and sockets. In *Proceedings of SIGCOMM '05*. ACM, August 2005.

[2] G. Camarillo. The early session disposition type for the Session Initiation Protocol (SIP). IETF Network Working Group Request for Comments 3959, 2004.

[3] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[4] JSR 309: Java media server control. Java Community Process, `http:// jcp.org/ aboutJava/ communityprocess/ edr/ jsr309`.

[5] R. Mahy, V. Gurbani, and B. Tate. Connection reuse in the Session Initiation Protocol (SIP). Internet Draft draft-ietf-sip-connect-reuse-09, 2008.

[6] J. Rosenberg. The Session Initiation Protocol (SIP) UPDATE method. IETF Network Working Group Request for Comments 3311, 2002.

[7] J. Rosenberg, J. Peterson, H. Schulzrinne, and G. Camarillo. Best current practices for third party call control in the Session Initiation Protocol (SIP). IETF Network Working Group Request for Comments 3725, 2004.

[8] J. Rosenberg and H. Schulzrinne. Reliability of provisional responses in Session Initiation Protocol (SIP). IETF Network Working Group Request for Comments 3262, 2002.

[9] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. IETF Network Working Group Request for Comments 3261, 2002.

[10] R. Sparks. Multiple dialog usages in the Session Initiation Protocol. IETF Network Working Group Request for Comments 5057, 2007.

[11] Pamela Zave. Audio feature interactions in voice-over-IP. In *Proceedings of the First International Conference on Principles, Systems and Applicatons of IP Telecommunications*, pages 67–78. ACM SIGCOMM, 2007.

[12] Pamela Zave and Eric Cheung. Compositional control of IP media. *IEEE Transactions on Software Engineering*, 2008. To appear.