Abstractions for Programming SIP Back-to-Back User Agents

Pamela Zave
AT&T Laboratories—Research
pamela@research.att.com

Gregory W. Bond
AT&T Laboratories—Research
bond@research.att.com

Eric Cheung
AT&T Laboratories—Research
cheung@research.att.com

Thomas M. Smith
AT&T Laboratories—Research
tsmith@research.att.com

ABSTRACT

In SIP services, *back-to-back user agents (B2BUAs)* are powerful but difficult to program correctly. StratoSIP is a high-level, domain-specific language for programming SIP B2BUAs safely. This paper describes the four major abstractions on which the language is based. It explains how each abstraction is used in programming, and how it is implemented in SIP. Because the abstractions are derived from the Distributed Feature Composition (DFC) architecture, StratoSIP programs compose easily with each other at runtime. The implementation of StratoSIP runs in SIP Servlet containers.

1. INTRODUCTION

SIP (IETF RFC 3261) is the dominant protocol used for IP-based multimedia systems. *User agents*,, which are usually SIP clients running on endpoint devices, may initiate and terminate multimedia calls. A *back-to-back user agent (B2BUA)* is a software process containing two or more user agents back-to-back, as shown in Figure 1. A B2BUA is often hosted by an application server in the signaling path between endpoints. A B2BUA is a powerful SIP entity because it acts as user agent on each *dialog* that it participates in, which means that it can perform all call-control functions.

B2BUAs can be used to implement SIP-based telecommunication services in a way that is both modular and compositional. This is not a property of arbitrary B2BUAs, but rather of those designed and composed according to the principles of the Distributed Feature Composition (DFC) architecture [6]. The DFC architecture is a proven technology for rapid development of high-quality, easily modifiable telecommunication services [14].

This paper reports on our progress toward making the benefits of DFC available to SIP programmers. StratoSIP

IPTCOMM '09 July 7-8, 2009, Atlanta, Georgia, USA

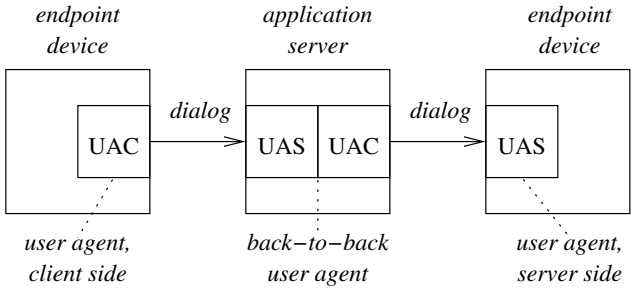

Figure 1: A typical back-to-back user agent.

("SIP at a very high level") is a new language for programming B2BUAs, designed specifically for this purpose. It is based on a set of abstractions adapted from DFC. The abstractions do not restrict what a B2BUA can do, and they support DFC-style modularity and composition.

There are several reasons why SIP programmers may have avoided B2BUAs in the past. First, there are many motivations for deploying services in SIP endpoints, and B2BUAs are usually associated with application servers in the network. Our implementation of B2BUAs runs well in endpoints, however, so there is no need to avoid them to keep services in endpoints [3].

Second, most network services are implemented as SIP proxies, which are restricted in their behavior but can nevertheless perform a wide range of routing functions. From a broader perspective, routing services are the easy ones. There are many other valuable services that can only be implemented in B2BUAs because they require initiating and terminating calls, multi-party control, and media control. Also, this paper will show that, when modularity and composition are supported, some services that could formerly be implemented in proxies must now be implemented as B2BUAs.

Third, SIP B2BUAs are inherently complex and therefore difficult to program. For this reason, the StratoSIP abstractions hide most of the details of SIP signaling, so that programming B2BUAs becomes a relatively easy task. The design of these abstractions is based on long experience, and they have been subjected to extensive formal modeling and verification. This makes StratoSIP programming safe as well as easy.

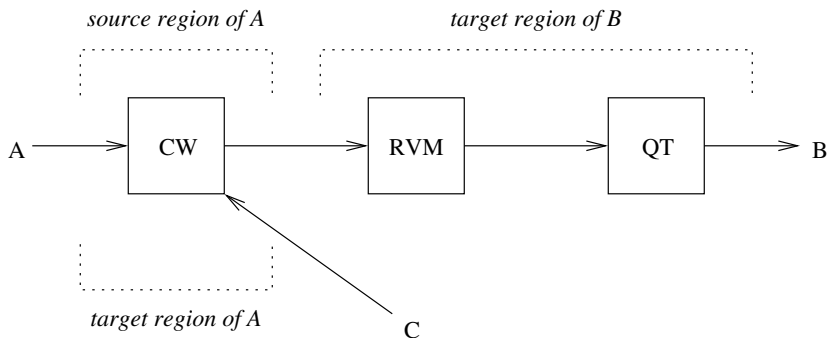

Figure 2: A DFC usage with three feature boxes. *A, B,* **and** *C* **are endpoint devices.**

Fourth, B2BUAs appear to present a danger of disrupting end-to-end signaling. Our implementation of StratoSIP is designed to preserve correct end-to-end behavior of SIP signaling (except when that disruption is an intentional effect of the B2BUA's function and purpose). Like the other signaling properties of our implementation, important aspects of end-to-end correctness have been modeled and verified.

The main focus of this paper is the abstractions in StratoSIP that make B2BUAs easy and safe to program, and make them composable modules within the DFC architecture. Section 2 introduces the most relevant aspects of DFC. Section 3 gives an overview of StratoSIP programs, while Sections 4 through 7 present the four principal abstractions in StratoSIP. Section 8 describes our implementation of StratoSIP, which runs in SIP Servlet containers. The remainder of the paper discusses related and future work.

2. MODULARITY AND COMPOSITION IN THE DFC ARCHITECTURE

Modularity and composition in StratoSIP are adapted from modularity and composition in the Distributed Feature Composition (DFC) architecture. This section gives a brief overview of how DFC achieves these goals.

The unit of modularity is a *feature,* which can be any increment of functionality that modifies or enhances basic connection service between two telecommunication endpoints. According to the DFC architecture, in the presence of features telecommunication service is provided by a graph called a *usage*, as shown in Figure 2. The nodes of a usage are *feature boxes*, and each feature box is a concurrent software process that implements a separate feature.

The edges of a DFC usage graph are complete *internal calls* providing two-way, reliable, FIFO signaling connections. This means that each feature box is a signaling endpoint for the internal calls that it participates in. The word "call" in *internal call* emphasizes the fact that signaling along each edge conforms to a protocol that is based on common telephony protocols. The word "internal" is used to distinguish an edge in the graph from the informal, end-to-end meaning of "call" in telecommunications. A DFC process sends a *setup* message to initiate an internal call.

The DFC messages include status indicators, the two most important of which are *avail* and *unavail.* The *avail* message travels from the callee or receiving end of a call to the caller or initiating end. It indicates that the entity identified by the target address is available for communication. Its dual is *unavail*, which indicates that the targeted entity is not available.

The DFC messages also include messages that set up and tear down media channels between endpoints connected by signaling paths.

The goals of DFC are modularity and composition. Modularity is achieved when a feature can be designed and implemented independently of all other features, and when features can be added to a system or deleted from it without changing other features. Composition is achieved when all the features applicable in a situation work properly in a cooperative and synergistic way.

A well-designed DFC feature box has the properties of *transparency, autonomy,* and *context-independence.* The next few paragraphs define these properties and explain how they contribute to modularity and composition.

Transparency means that when a feature box is not actively performing its intended function, it is unobservable by other boxes in the graph. It is acting as an identity element. Transparency supports composition by ensuring that an inactive feature does not interfere with active elements in a usage.

Autonomy means that when a feature needs to perform some function, it does so without help from other boxes. A DFC feature box can act autonomously because it sits in a signaling path between endpoint devices, where it can observe all the messages that travel between them. Because it is a protocol endpoint, it can absorb or generate any messages that it needs to. It can even reconfigure the usage by tearing down some internal calls and setting up others. Autonomy supports modularity by making a feature box self-contained.

Context-independence means that a feature does not know or need to know what is at the other ends of the internal calls it is participating in—those calls might lead to endpoint devices or to other feature boxes. This is essential for composition because it means that a box reacts identically to a stimulus from an endpoint device or from another feature box.

For example, consider the Record Voice Mail (RVM) and Quiet Time (QT) features serving endpoint device *B*, as seen in Figure 2. RVM is initially transparent. It reacts to *unavail* by placing a new call to a voicemail resource, and connecting the caller with the resource so that the caller can record a voice message. When Quiet Time is enabled, as soon as it receives an incoming call, it connects the caller to a

media server that implements an interactive voice-response session. The server announces that the subscriber wishes not to be disturbed unless the call is urgent, then prompts for a touch-tone choice. If the caller says the call is urgent, then QT disconnects the server, places a transparent outgoing call, and goes permanently transparent. If the caller says the call is not urgent, then QT sends *unavail* upstream, tears down its incoming call, and terminates itself.[1]

Either of RVM and QT behave correctly and make perfect sense acting alone. They also behave well when composed, because of context-independence. If RVM receives an *unavail* from downstream, it does not know or care whether that message was generated by the callee's endpoint device, QT, or some other feature box. In all cases it gives the caller an opportunity to record a voice message.

A DFC usage is assembled dynamically and evolves over time. The mechanism for assembling usages is the *DFC routing algorithm*, executed by a *DFC router*. A DFC router has a different purpose from IP routers. The purpose of an IP router is to find the destination of a message, while the purpose of a DFC router is to insert feature boxes in the paths of setup messages.

Each time a box sends a setup message, that message goes to a DFC router that chooses a box to receive it, and forwards the *setup* to the receiving box. Then the receiving box sends an acknowledgment directly to the sending box, and an internal call is formed between them.

Every continuous routing chain from one endpoint to another contains a *source region* and a *target region*. The source region comes first; it contains feature boxes working on behalf of the source address in its role as caller. The target region contains feature boxes working on behalf of the target address in its role as callee. Each address *subscribes* to some (possibly empty) set of feature box types in each region. In the figure, the routing chain from A to B has Call Waiting (CW) in the source region of A, and RVM and QT in the target region of B.

A simple routing chain from endpoint to endpoint begins when the calling endpoint creates a setup message with the *new* method and sends it to a DFC router. To continue the chain, a feature box takes a setup message it has received and applies the *continue* method to it. The *continue* method returns a setup message that is suitable for continuing a chain rather than starting a new one, and the box then sends it to a DFC router.

Correct composition of features depends on assembling the boxes of a region in the correct order. For example, RVM and QT only interact correctly if RVM comes first. This order is governed by a *precedence* relation in router data.

Feature box types fall into two categories: *free* and *bound*. When a DFC router is working on a setup message and selects a free box type as its destination, the router creates a new feature box (program object) as an instance of its type. Thus each free feature box is a transient, anonymous instantiation of its type. Bound feature boxes are different because, for each address subscribing to a bound feature box type (in either region), there is at most one instance of that box type at any time. When a router is working on a setup message and selects a bound box type as its destination, if there is currently an instance of the bound box for that address, then the *setup* goes to the existing instance.

In Figure 2, CW is a bound box type. As often happens, A subscribes to it in both source and target regions, because Call Waiting should be applicable to both outgoing and incoming calls. Because CW is bound, C's call to A is routed to this instance in A's target region. This allows A to use CW to switch between B and C if desired. The routing chain from C to A has no features in the source region of C, and CW in the target region of A.

There are other aspects of DFC routing, but this brief introduction should be sufficient because this paper focuses on signaling. Taken as a whole, the DFC routing algorithm supports transparency, autonomy, and context-independence of feature boxes.

3. STRATOSIP PROGRAMS

A StratoSIP program defines a SIP B2BUA that approximates a DFC feature box. A DFC internal call becomes an invite dialog in SIP.

The implementation of StratoSIP requires only the SIP messages used in the general SIP standard RFC 3261. StratoSIP also handles some additional message types, as discussed in Section 7.

Semantically, a StratoSIP program is a sequential, deterministic finite-state machine. Figures 3 and 4 are StratoSIP programs in graphic form. A black dot is an initial state, while the labeled states are *stable* states. The program implementation has an input queue for each established invite dialog, and one for incoming invite messages that establish new dialogs.

Each explicit state transition is labeled by a guard, optionally followed by a slash and a list of actions. The execution cycle of a StratoSIP program is as follows. In a quiescent state, if there is a message in one or more input queues, the program chooses among the nonempty input queues. It takes a message from the chosen queue and finds the unique transition—which may be explicit or implicit—whose guard is made true by the message. It then executes that transition, which may entail a state change and many actions, both explicit and implicit. When all of these actions have been performed, the program becomes quiescent and is ready to process the next input.

Note that this cycle definition automatically makes the program *input-enabled*, which means that it can read and process any message from any input queue at any time. We add to this definition the constraint that the language implementation must choose fairly among the input queues. With this constraint, we have a guarantee that there is no deadlock or livelock among B2BUAs.

The Attended Transfer (AT) feature enables a trainee agent in a customer-service center to transfer a customer call to a more experienced agent. The Attended Transfer program (Figure 3) is used in examples throughout this paper. Initially the feature receives an *invite* from a customer, labels the dialog it initiates c, and continues it transparently to a trainee agent t. Once customer and trainee are *Talking*, the trainee can use a Web interface to move into a *Consulting* state, in which the customer is on hold and the trainee is talking to an expert agent (e). After consulting with the expert, the trainee can resume talking to the customer, or

[1]Note that usages are dynamic, evolving as the features perform their functions. Thus any picture of a usage is a snapshot. Figure 2 is only accurate with respect to A and B after QT places a transparent outgoing call and before this call fails or is disconnected.

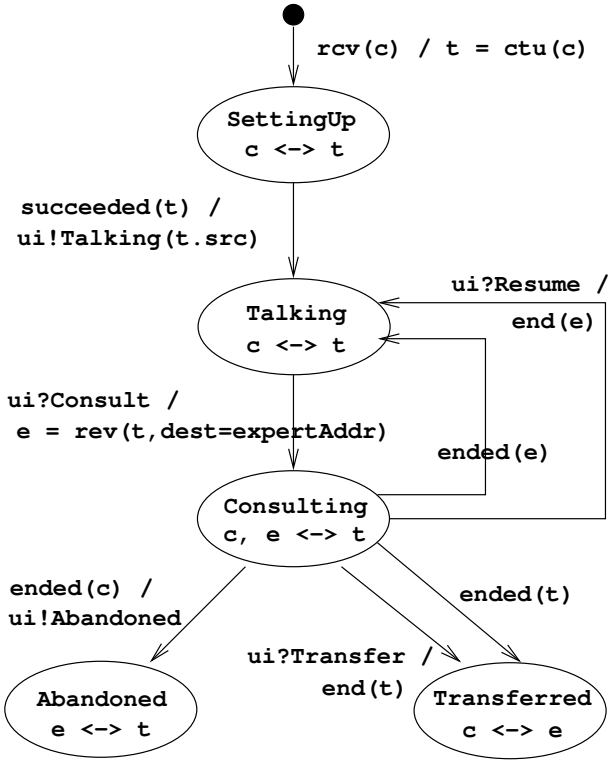

Figure 3: StratoSIP program for the Attended Transfer feature.

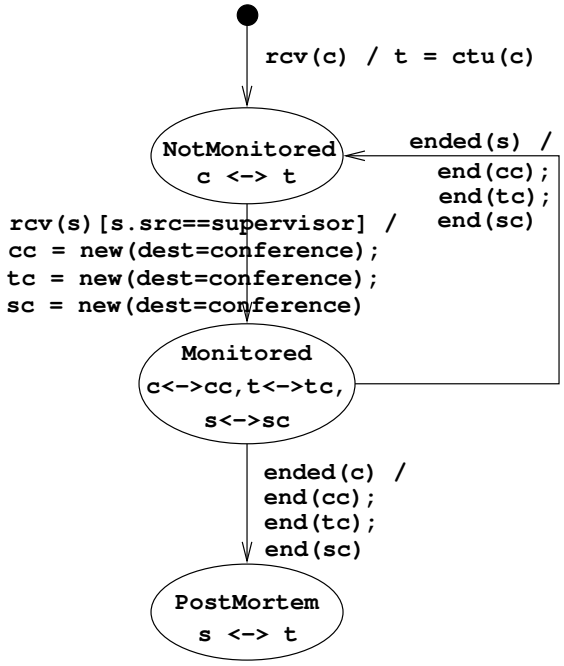

Figure 4: StratoSIP program for the Trainee Monitoring feature.

transfer the customer to the expert. If the customer gets tired of waiting and hangs up during the consultation, then trainee and expert are informed, and enter an *Abandoned* state in which they are still talking.

The Trainee Monitoring (TM) feature allows the supervisor of a trainee agent to listen to the trainee's conversations and whisper advice. The Trainee Monitoring program (Figure 4) is also used in examples. Initially the customer and trainee are talking normally, just as in AT. If the trainee's supervisor (s) calls in, then the program goes to a *Monitored* state in which all three parties are connected to a conference bridge, so that the supervisor can listen and whisper to the trainee. If the supervisor hangs up, then the customer and trainee are still talking; if the customer hangs up, then the supervisor and trainee are still talking.

Note that a Trainee Monitoring box must be a bound box, because only a bound box can receive an *invite* after the *invite* that instantiates the box. An Attended Transfer box is a free box.

StratoSIP is written in textual form, with graphic versions of programs being generated automatically. This paper makes no attempt to present the complete language syntax or semantics. The language aspects that are omitted or abbreviated are ordinary aspects of programming languages such as declarations, control structures, predicates, and manipulation of data structures such as messages. StratoSIP has embedded Java syntax for access to Java data structures and APIs. For comparison, the textual form of the AT program is given in the appendix.

StratoSIP is a descendant of Boxtalk [16], which was a language designed for DFC but never implemented.

4. ACTIVE DIALOGS

From the perspective of a StratoSIP program, an *active dialog* is an invite dialog that exists and can be observed and manipulated by the program. In its initial state, a StratoSIP program has no active dialogs.

An active dialog always begins when the program sends or receives an initial invite message. The program receives an *invite* through the guard `rcv(`*dialog1*`)`, which becomes true when and only when the invite message arrives. The unique identifier of the active dialog is assigned to the dialog variable *dialog1*, where it acts like a pointer to the dialog.

The program can send an initial *invite* through an action such as *dialog2* `= new(`*dest*`)` or *dialog2* `= ctu(`*dialog1*`)`. In both of these actions *dialog2* is the variable for the new dialog. The operation name `new` or `ctu` refers to the *new* or *continue* method of the DFC routing algorithm, respectively, used to create the *invite*. The method ensures that the message is correctly formed for a DFC application router. The argument *dest* is the destination address (SIP Request URI) in the *invite* resulting from the `new` action. The argument *dialog1* is the variable pointing to the dialog whose *invite* is continued to make the *invite* resulting from the `ctu` action.

At all times, the source (SIP From) and destination addresses of the *invite* that began a dialog can be referred to as *dialog*`.src` and *dialog*`.dest`, respectively.

Both the AT and TM programs begin with a transition labeled `rcv(c) / t = ctu(c)`, where c points to the dialog from the customer, and t points to the dialog to the trainee. These transitions end in stable states annotated with the dialog variables c and t. In StratoSIP every stable state must be annotated with the variables pointing to the dialogs that are active in that state. Nothing is more important for the programmer to remember than what dialogs are active in any state.

In addition to `rcv, new`, and `ctu`, the only other way to create an active dialog is the operation `rev` for *reverse*. *Reverse* is a DFC routing method similar to *continue*, except that it is used when the new dialog has reversed roles with respect to the dialog it is derived from. In AT, the dialog to the consulting expert is derived from t with *reverse*. In t the trainee is the destination of the dialog, while in e the trainee is the source of the dialog.

In StratoSIP the four operations that create active dialogs all appear to be atomic and instantaneous. The implementation of `rcv` immediately replies with a *183* message to establish the dialog, and does not wait while the final response to the received *invite* is being determined (the final response is handled by other StratoSIP operations). The implementations of the dialog-initiating operations send the *invite*, but do not wait for a final response. Atomic, instantaneous operations keep programming simple, because the programmer does not need to think about concurrency.

The operations that destroy active dialogs are also atomic and instantaneous from the program perspective. The guard `ended(`*dialog1*`)` becomes true when the implementation receives a message indicating that the other end of *dialog1* wants it to end. Depending on the SIP state of the dialog, this might be a *cancel* or *bye* request, or any failing final response to the initial *invite*. If `ended(`*dialog1*`)` is followed by a predicate such as `[BusyHere]`, then the entire guard becomes true only if the message received is the matching SIP final response *486*. The StratoSIP implementation automatically generates any required response to the message that makes `ended` true, for example *200 OK* in response to a *bye* request.

To end an active dialog, a programmer uses the action `end(`*dialog1*`)`. Depending on the SIP state of the dialog, this might entail sending a *cancel* or *bye* request, or a failing final response to the initial *invite*. If `end(`*dialog1*`)` is followed by a message modifier such as `[Unauthorized]`, and if the action is implemented by sending a final response, then the final response is selected by the message modifier: *401*. If a failing response must be sent and there is no message modifier, the default message type is *486*.

After an `ended` guard or `end` action, the dialog is no longer active and can no longer be manipulated through StratoSIP. The implementation automatically handles any cleanup messages.

As further programming conveniences, StratoSIP has some implicit transitions and actions that end active dialogs. First, if a program for a bound feature box has a stable state with no out-transition guarded by `rcv`, then that state has an implicit self-transition `rcv(unwanted)/end(unwanted)`. Without this implicit transition the program would not be input-enabled.

Second, if a stable state with active dialog *dialog1* has no out-transition guarded by `ended(`*dialog1*`)`, then it has an implicit transition with this guard, entering the distinguished *terminal* state in which the program has terminated.

Third, the terminal state has no active dialogs. If a transition enters the terminal state, then it has implicit actions that end all remaining active dialogs. Because of the second and third rules, the AT and TM programs need no explicit transitions out of their *Abandoned, Transferred,* and *Postmortem* states.[2]

[2]Some programs need explicit transitions to the terminal state. A transition to the terminal state can be found in Figure 8.

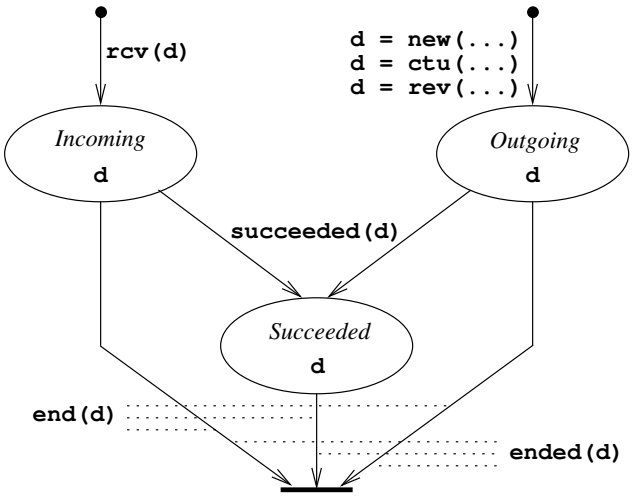

Figure 5: The states and transitions of an invite dialog in StratoSIP.

Figure 5 shows the major dialog states in StratoSIP, along with the guards and actions that cause state transitions. The guard `succeeded(`*dialog1*`)` is satisfied by the received message first indicating that there has been success in reaching the desired party. Commonly the message that makes *succeeded* true of an outgoing dialog is a *200 OK* in response to the initial invite. In StratoSIP incoming dialogs can also become *succeeded*, and messages other than *200 OK* can cause the state transition. These variations are explained in Sections 5 and 6.

In AT, a transition guarded by `succeeded(t)` indicates that the trainee is now talking to the customer. The transition sends a message to the trainee's user interface (a Web application). On receiving this information, the application displays a *Consult* button for the trainee to click when needed.

In AT, *ui* is a non-SIP message port allowing communication between the feature and its Web user interface. A program can have any number of non-SIP message ports, which look and act like active dialogs except that they are neither created nor destroyed by dialog operations. AT also informs the user interface when it enters the *Abandoned* state. The messages *Consult, Resume,* and *Transfer* are sent by the user interface because of mouse clicks in the trainee's browser.

Satisfying the guard `succeeded` is equivalent to sending or receiving an *avail* message in DFC. The equivalent to *unavail* in DFC is a failure response to an initial *invite*, which satisfies the guard `ended`.

Whenever a dialog is active, it is possible to send and receive miscellaneous messages through the dialog, using the syntax *dialog1* `!` *messageType* and *dialog1* `?` *messageType* respectively. Section 7 will discuss this further.

5. COMPOSITIONAL MEDIA CONTROL

The state annotations in StratoSIP control media channels. If a state is annotated *dialog1* `<->` *dialog2*, which is called a *link* between the active dialogs, then the implementation of the program attempts to create a media connection

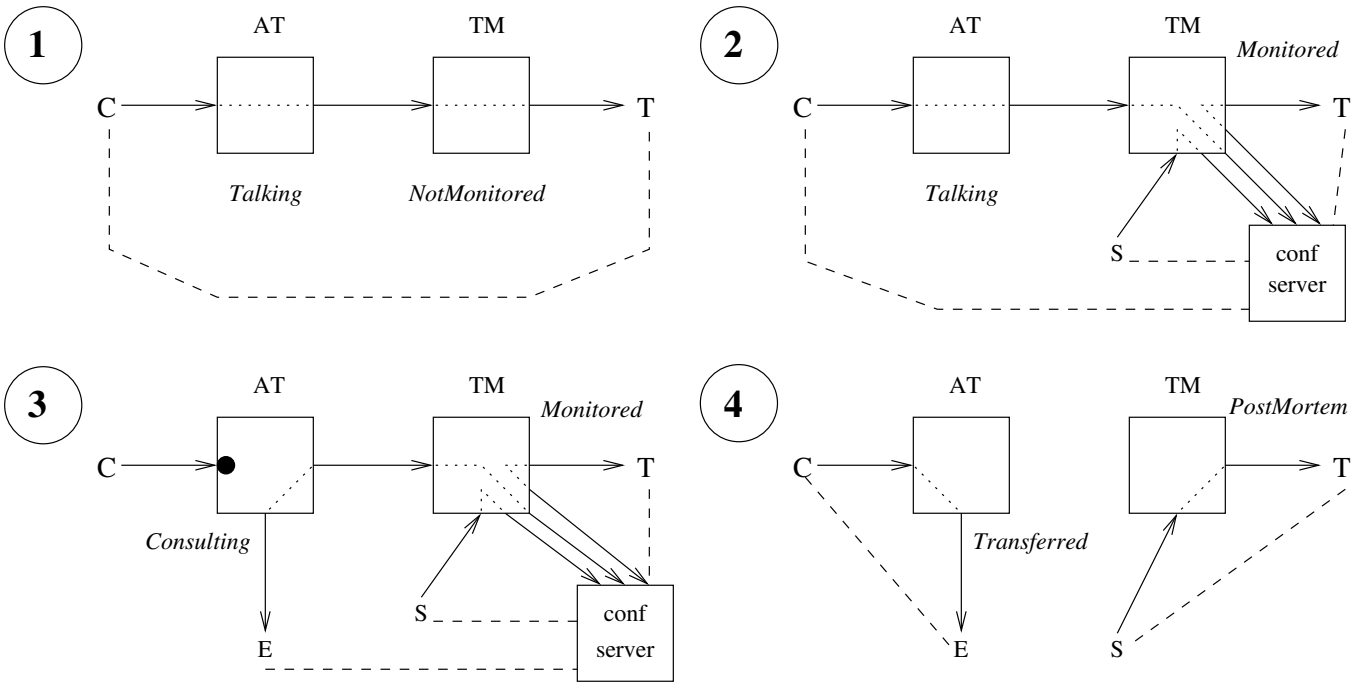

Figure 6: The composition of media control in Attended Transfer and Trainee Monitoring.

between the endpoint devices to which *dialog1* and *dialog2* lead. If a state annotation contains *dialog3* standing alone, then the implementation of the program attempts to put media at *dialog3*'s endpoint device on *hold.* Holding means that a media connection is established or not, according to the choice made at the endpoint device, but even if a connection is established no actual media packets are being transmitted.

StratoSIP assumes that a SIP message contains at most one session descriptor. StratoSIP treats descriptors as wholes, so a media connection between endpoints might include multiple channels transmitting several different media.

We say that implementation of a StratoSIP program "attempts" to produce the desired media state, because no feature has unilateral control over media connections. Rather, correct media control is a *composition* of the states and goals of all the features in a signaling path. This is illustrated by Figure 6, which shows what happens when a trainee's address subscribes to both AT and TM.

The figure contains four snapshots that occur in numbered order. In Snapshot 1, AT is in its *Talking* state and TM is in its *NotMonitored* state. The arrows show dialogs among the customer C, the trainee T, and the two features. The dotted lines within boxes are a graphic representation of the links in the boxes' current states. The dashed line represents where there should be an actual audio channel, connecting the endpoint devices along the shortest possible path.

In Snapshot 2, the trainee's supervisor S has called in, and TM is now in the *Monitored* state. There is a conference server whose address is *conference* in the TM program. The links within the TM program have changed, and the correct media paths have changed along with them. In general, assuming that all endpoint devices want media connection, there should be a media connection between two devices if and only if there is an unbroken path of dialogs and intra-box links between them.

In Snapshot 3, the trainee has clicked "Consult" and AT is in its *Consulting* state with new endpoint E (the expert). The state in AT has new annotations (the dot shows that C is on hold) and the correct media connections have changed with the annotations. This shows clearly that correct media behavior is not determined by one feature, but rather by the composition of both of them.

Note that the implementation of AT works concurrently with all dialogs to effect the changes from Snapshot 2 to Snapshot 3. If it did one change at a time, the transition would be much slower, and there might be undesirable intermediate media states.

In Snapshot 4, the trainee has clicked "Transfer," which caused AT to change media links and end its dialog t. Because this is the same as TM's dialog c, TM is now in its *PostMortem* state.

In [15] we argue that the two media-control primitives *link* and *hold* are sufficient for a wide range of applications. State annotation is all that a StratoSIP programmer does to control media for his application. The implementation of a StratoSIP program automatically performs compositional media control, so that the actual media channels are correct regardless of how many features are in the signaling path. They also work correctly when the endpoint devices alter media channels by issuing *re-invite* messages.

Note that the implementation of an annotation change must work correctly regardless of the state of media signaling when the change occurs. Multiple features and endpoints may be attempting changes concurrently. A nervous trainee's rapid clicking may initiate a change before the previous change has been completed. The tremendous complexity of these situations in SIP means that it would be imprudent to trust an implementation without formal anal-

ysis and verification.

Achieving correct compositional media control is no small task. Our implementation is the culmination of many years of work on this problem. In [15] we specified the problem formally in temporal logic, showed a solution, and partially verified the solution using model-checking. This solution is relatively simple because it uses a signaling protocol designed to make composition easy.

In [4] we presented a quite different algorithm that satisfies the same specification but with SIP signaling. This algorithm uses SIP in the third-party call-control style of RFC 3725. It has also been partially verified by model-checking. This is the algorithm for compositional media control built into StratoSIP.

The StratoSIP implementation of media control hides from the programmer all messages that carry session descriptions, which are the messages in *invite* and *re-invite* transactions. The implementation handles these messages automatically, doing whatever is necessary to match the current state annotations, as described in [4]. The main exception to the hiding is that a StratoSIP program has access to the non-media aspects of initial *invites*, through the operations that create dialogs.

In AT, when t receives a message causing that dialog to enter the *succeeded* state, the implementation will send a message with the same semantics to c, and c will also enter the *succeeded* state. This is another result of the link between t and c in AT.

Figure 6 also illustrates the role of precedence in governing feature interaction. If the order of the features were reversed, then when AT was in its *Consulting* state and TM was in its *Monitoring* state, the supervisor would not be able to hear or join the conversation between the trainee and the expert. Clearly the precedence order in the figure causes the features to interact in a better way.

6. EARLY MEDIA

In SIP, "early media" is media flow before the callee endpoint device has sent *200 OK* indicating that the callee is present.

Most discussions of early media, for example RFC 3960, assume that the source of early media is the same as the source of non-early media, namely the callee's endpoint device. Often, however, a media channel is used for control purposes. Telephony features use the audio channel for getting information from the caller about how to handle the call, for authenticating the caller (as opposed to authenticating the device that the caller is using), and for informing the caller about the progress of the attempt to reach the callee. None of these early-media streams can come from the callee's endpoint device, because the device has not even received an *invite* yet.

Early media is a problem in SIP because an initial invite transaction incorporates both negotiation of the first media connection and signaling that the call has succeeded. General-purpose early media, as described above, requires separating the two. Early media is not a problem in DFC because messages can be used to set up media channels at any time; these messages are independent of status messages such as *avail* and *unavail*.

How can we approximate this aspect of DFC in SIP? The SIP Working Group showed some awareness of this problem by standardizing the reliable provisional response (RFC 3262) and the *update* message (RFC 3311) soon after the second version of the SIP standard (RFC 3261). Although it seems that these extensions should solve the problem, they do not. The reason is that third-party call control relies on the following property of invite transactions: If an *invite* has no session description, it is soliciting an *offer* session description from the recipient. If the recipient sends a successful response, that response must carry an offer, and the inviter then sends an *answer* session description in the *ack* message. Because of this capability, the *invite* and the *offer* can come from different ends of the transaction. Neither reliable provisional responses nor *update* messages allow this freedom [13].

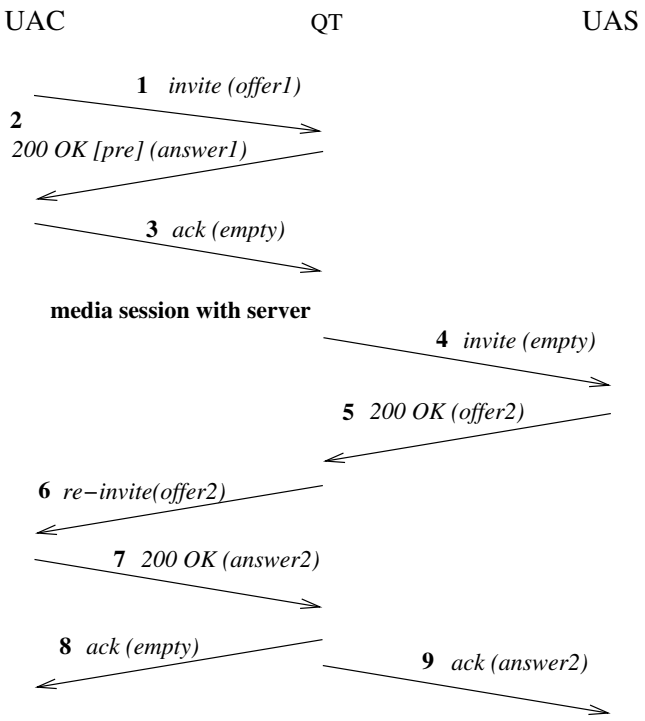

Figure 7: SIP signaling for early media.

The StratoSIP approach is to use *invite* (and *re-invite*) transactions for all media control, because that is the simplest and most general solution [12]. To illustrate this solution, Figure 7 shows the SIP signaling among two user agents and a feature box with early media. The feature is Quiet Time (QT), first introduced in Section 2.

In the figure, QT acts as the recipient of the initial invite transaction, answering the offer to establish an early media session (Messages 1 and 2). The *ack* is labeled *empty* because it has no session description (Message 3).

When the early media session is over, if the caller has indicated that the call is urgent, QT solicits an offer from the UAS (Messages 4 and 5). Eventually the UAS receives an answer in the *ack* message that completes the transaction (Message 9).

Because the protocol states on its left and right are different, QT must continue the rightward *invite* transaction (Messages 4, 5, and 9) with a different leftward *re-invite* transaction (Messages 6, 7, and 8). These message translations illustrate the fact that programming a SIP B2BUA to

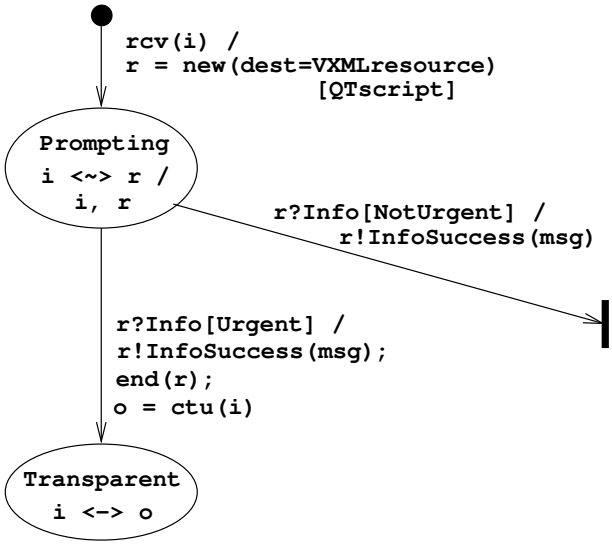

Figure 8: StratoSIP program for the Quiet Time feature.

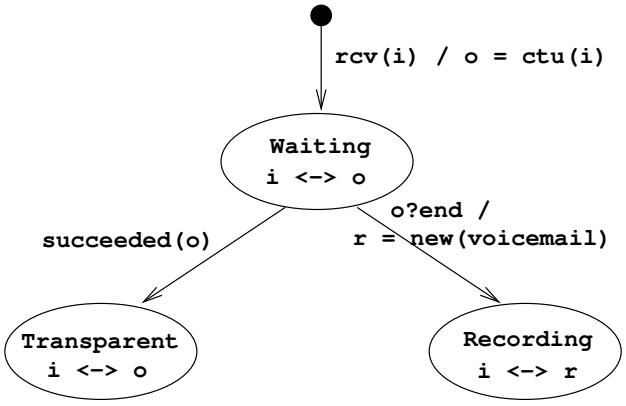

Figure 9: StratoSIP program for the Record Voice Mail feature.

act transparently often entails far more work than merely relaying messages from side to side.[3]

With this approach, one problem remains. On receiving Message 2, the UAC or any feature between the UAC and QT, such as Record Voice Mail (RVM), think that the callee is now present. A human caller will quickly learn the difference, but RVM will not, and will go permanently transparent. As explained in Section 2, it is essential for composition that RVM be triggered by call failure, which is now impossible because *200 OK* precludes failure of the initial *invite*.

We solve this problem by attaching a *preliminary* tag (*pre*) to any SIP message, such as Message 2, that is not meant to convey final success of the initial invite transaction. This tag decouples status from media signaling, as in DFC. If RVM is to the left of QT in Figure 7, the *200 OK [pre] (answer1)* will not make *succeeded* in its dialog true, but the subsequent *re-invite* will. Figures 8 and 9 show the StratoSIP programs for QT and RVM.

On receiving an initial *invite*, QT initiates a new dialog to a VoiceXML server, passing the server a URL pointing to a script. (The URL is supplied by the message modifier `[QTscript]`.) In the *Prompting* state the VoiceXML server is linked to the caller.[4] Following the script, the server plays an announcement to the caller and then prompts for a touch-tone indicating whether the call is urgent. Unlike the Web interface to the AT feature, the server returns the results to QT in SIP *info* messages (RFC 2976).

Note that the link inside *Prompting* is written with a tilde rather than a hyphen. (Ignore the state annotation after the slash, which will be explained in Section 7.) This means it is a *prelink*, which acts in all respects like a link except that it does not propagate final success from dialog r to dialog i. The prelink causes Message 2 in Figure 7 to have a *pre* tag.

If the caller says his call is urgent, then QT continues i to o and enters a state with a normal link. Because of it, dialog success from the right (Message 5, which has no *pre* tag) propagates to the left (Message 6 has no tag).

A normal link propagates success but does not generate it. If there were another feature with early media to the right of QT, then Message 5 would have a *pre* tag, and so would Message 6.

If the caller says his call is not urgent, then QT goes to the distinguished terminal state, which is represented graphically by a bar. An implicit action will end i by sending *bye*. If RVM is to the left of QT, it experiences the end of dialog o before o has succeeded. RVM interprets this as call failure, and is triggered to connect the caller with a voicemail server.

StratoSIP also has two versions of the state annotation for holding a dialog. If a state is annotated with a dialog variable not linked to another dialog variable but preceded by a tilde, the annotation is called a *prehold*. The difference between normal hold and prehold is that prehold does not generate success. In other words, if there is a transition with guard `rcv(`*dialog1*`)` going into a state in which *dialog1* is held normally, then the implementation responds with a normal *200 OK*. If a transition with the same guard goes into a state in which *dialog1* is preheld, then the implementation responds with *200 OK* with a *pre* tag.

When StratoSIP features interact with other SIP elements, they receive no *pre* tags, and the *pre* tags they send are ignored. This loss of information reduces the ability of features to discriminate cases, but causes no other harm [12].

7. STATUS MESSAGES

The previous three sections show how the StratoSIP primitives for active dialogs, compositional media control, and early media automatically handle the SIP messages involved in creating invite dialogs, destroying them, and controlling media sessions. These are the request types *invite, ack, cancel*, and *bye*, and all their final responses. None of these messages is manipulated directly by a StratoSIP programmer.

We call other SIP messages *status* messages, and allow

[3]It is also necessary to reformat the session descriptions between the two transactions, because the channel order in *offer2* may not match the channel order in *offer1*.

[4]Note that QT's dialog with the media server is not shown in Figure 7. To give more detail, *offer1* is sent to the server, and *answer1* is received from the server.

programs direct access to them. At the same time, StratoSIP has mechanisms for dealing with them implicitly if desired. These implicit mechanisms are convenient, and they also serve the important purpose of ensuring well-defined default handling of every message. Without them, StratoSIP programs would not be input-enabled.

The current StratoSIP types for these messages are *Ringing* or *180, Forwarded* or *181, Queued* or *182, Progress* or *183, Info, InfoSuccess, InfoFailure, Options, OptionsSuccess,* and *OptionsFailure*. RFC 3261 says that *options* requests and their responses can occur within invite dialogs. The provisional responses above are all unreliable.

Register and *message* requests are not included in this list because they do not belong in invite dialogs (RFCs 3261 and 3428, respectively). *Subscribe* and *notify* requests can be sent within invite dialogs (RFC 2543), but, following the advice of RFC 5057, StratoSIP does not allow it. *Subscribe, notify, refer, prack, update,* and reliable provisional responses will be discussed further in Sections 9 and 10.

Message types can be used as patterns in guards, with optional predicates, such as `r?Info[Urgent]` in the QT program. Message types can be used as constructors in actions, with optional message modifiers, such as `r!InfoSuccess(msg)` in QT. Every constructor for a response type takes as a mandatory argument the message to which it is a response. `msg` is a built-in StratoSIP variable, defined during the execution of a transition as holding the message that triggered the transition.

Status messages can be handled explictly, as described in the previous paragraph. If a status message is not handled explicitly, then implicit mechanisms take over. Not surprisingly, if a request or *18x* message is received from a dialog that is linked to another dialog, implicit handling of the message is to forward it to the other dialog. If a request or *18x* message is received from an dialog that is held, implicit handling of the message is to save it in a forwarding queue for that dialog. If two dialogs are newly linked, and one has a nonempty forwarding queue, then the queued messages are immediately forwarded to the other dialog.[5]

Implicit handling of status responses is different, because responses must go where the requests came from. If a received response corresponds to a request generated by this feature, then the response is dropped. If a received response responds to a request that came from a dialog, and that dialog is still active, then the response is sent to that dialog.

Most of the time, the relationships among active dialogs governing media and status are the same. Dialogs are status-linked or status-held in the same way that they are media-linked or media-held. For those states in which this is not the case, an optional set of state annotations after a slash indicates the status links and holds.

This is illustrated by the *Prompting* state of the QT program. The two dialogs are media-linked but not status-linked, because status messages from i are intended for the callee, and status messages from r are intended for the feature. If the feature proceeds to the *Transparent* state, then all queued status messages from i will be forwarded to o.

8. IMPLEMENTATION

SIP Servlet containers are application servers running SIP, HTTP, and converged applications. Figure 10 shows the runtime environment of our implementation of StratoSIP, which is designed for SIP Servlet containers. A SIP Servlet container can run in the network or in an endpoint [3].

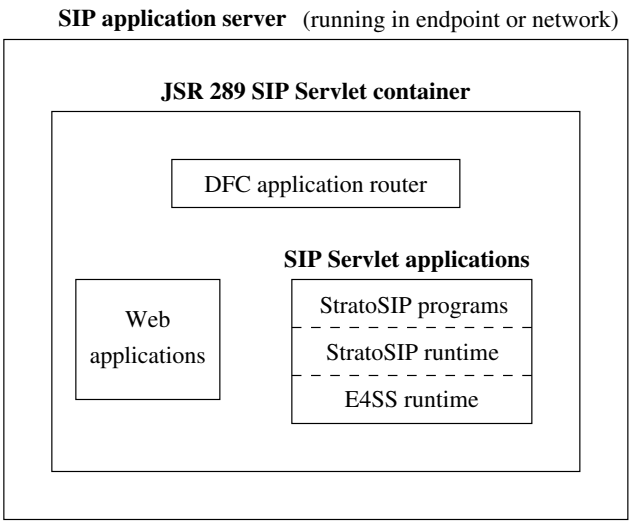

Figure 10: The runtime environment of our implementation of StratoSIP.

The current standard for SIP Servlet containers [7] provides for a deployer-supplied application router and an API to it. This API was designed with DFC in mind, and a DFC router is part of our ECharts for SIP Servlets (E4SS) suite of open-source tools [5]. There are now several implementations of the new SIP Servlet standard, including open-source implementations, so that anyone can run a SIP Servlet container equipped with a DFC application router.

Each StratoSIP program runs as a SIP servlet. StratoSIP is compiled into the ECharts language [1], which is also part of our ECharts for SIP Servlets (E4SS) suite of open-source tools. ECharts is compiled into Java, so all StratoSIP programs become applications in Java.

The implementation of StratoSIP also includes a library to support the four principal abstractions described in this paper. The SIP signaling to implement the abstractions has been modeled in Promela and partially verified using the model-checker Spin. For intuition about how the models look and what kind of verification can be done, see the similar work in [13].

As mentioned in Section 6, behavior that is transparent at the level of StratoSIP abstractions is often far from transparent at the level of the SIP implementation. Nevertheless, we are sensitive to the expectation that end-to-end signaling will be preserved by network elements. In our implementation we have taken care to preserve end-to-end signaling whenever possible.

At the time of this writing we have a StratoSIP-to-ECharts translator and have completed much of the StratoSIP library. Because we still lack runtime support for early media, handling of SIP status messages, and some other details, not all StratoSIP programs will run yet. Examples of programs that can run now, for example Attended Transfer, have been tested extensively with the testing tool KitCAT [10]. We expect to complete implementation of the initial

[5]There is an explicit action to clear the queue at any time, which can be used when this is not appropriate.

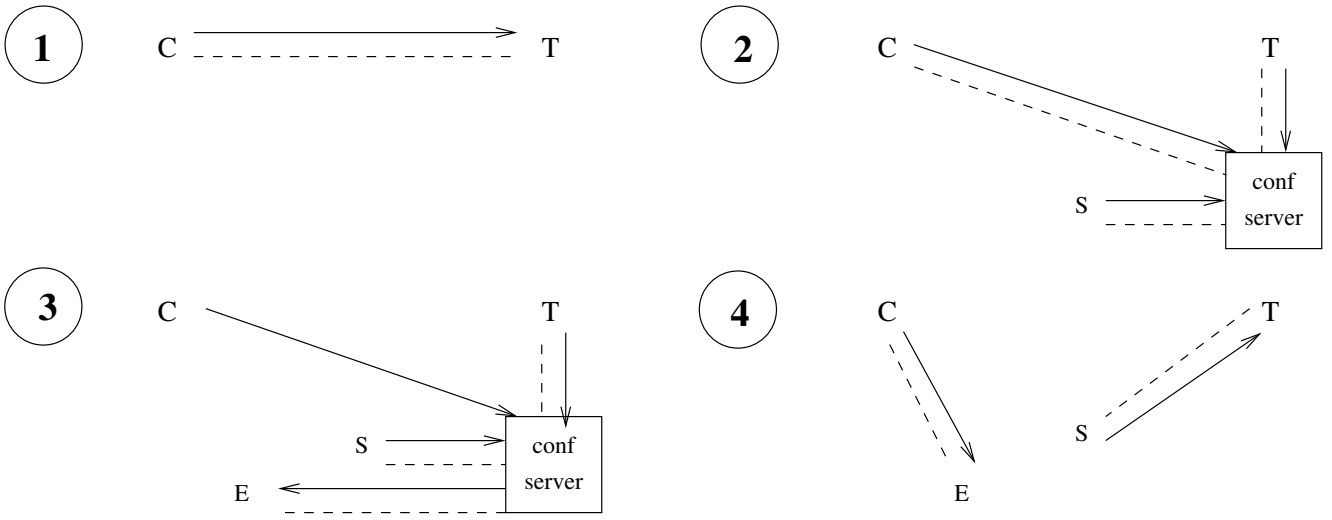

Figure 11: Attended Transfer and Trainee Monitoring implemented with *refer, replaces,* and *join*.

version of StratoSIP by the end of 2009.

Because a StratoSIP program runs as a servlet in a SIP Servlet container, its performance depends primarily on the container. We test routinely on at least one commercial and one open-source container, in order to head off any performance or integration problems that may arise.

9. RELATED WORK

Other high-level, domain-specific languages have been proposed for programming SIP services. As a fairly representative sample, we consider LESS, CPL, SPL, and Appel.

LESS [11] is intended primarily for services in endpoint devices. CPL ([9], RFC 3880) and SPL [8] are intended primarily for services in proxies. All are focused on routing logic, with LESS and CPL being more convenient than StratoSIP for constructing decision trees that govern how initial *invites* are redirected or otherwise handled. None is compositional, performs media control, or can be used to program B2BUAs.

Appel [2] is a language for expressing policies. These policies often concern routing logic, but may extend beyond the handling of initial *invites* to such area as mid-call conferencing. However, the programming capabilities of Appel are similar to those of LESS, CPL, and SPL. They fall far short of what would be necessary to implement features such as mid-call conferencing.

Feature function alone is not the only reason for choosing B2BUAs over proxies. For example, some readers may have noticed that our Record Voice Mail feature is a B2BUA, while most SIP implementations of redirection on failure (of which RVM is a special case) are proxies. Proxies are simpler than B2BUAs, so it might seem that making RVM into a B2BUA is an unnecessary complication.

StratoSIP RVM, however, works well in composition with other features, even features such as QT that use early media that is not from the callee's endpoint. A proxy RVM would fail to work in this situation, first because success of the early-media session would switch its function off, and second because (once the initial *invite* had succeeded) it could not initiate a new dialog with the voicemail server.

ECharts is a programming language featuring dynamic, hierarchical, and concurrent finite-state machines [1]. ECharts for SIP Servlets (E4SS) [5] offers a library of *fragments*, which are SIP-oriented finite-state machines that can be nested in the states of application programs.

Because we have used E4SS ourselves for some time, we can make a direct comparison between StratoSIP and E4SS programs. The top-level E4SS program for Attended Transfer has 9 states and 14 transitions. It uses one application-specific embedded finite-state machine and (directly or indirectly) 8 library fragments. Although an application programmer need not write fragments, he must understand them quite well. There are transitions at the top level, for example, whose guards examine the current states down to 3 or even 4 levels of nesting. The E4SS program for Attended Transfer does not control media compositionally, and it has not been verified.

Our final comparison is not to another programming language, but rather to another style of using SIP. As explained in Section 5, StratoSIP is implemented using the third-party call-control style of signaling (RFC 3725). Some SIP applications, particularly localized applications such as IP PBXs, now implement switching of media channels with the *refer* method and *replaces* and *join* headers (RFCs 3515, 3891, and 3911, respectively). RFC 4579 explains how to do conferencing with *refer, replaces,* and *join*.

Figure 11 shows how these capabilities can be used to produce the same behavior as produced by Figure 6. To explain the signaling behavior very informally, the change from Snapshot 1 to Snapshot 2 is initiated by two major messages: (1) S invites T, with a *join* header asking to join T's current dialog, and (2) T sends *refer* to C, requesting it to *replace* its current dialog with one to the conference server. The change from Snapshot 2 to Snapshot 3 is also initiated by two major messages: (1) T sends *refer* to the conference server, requesting it to put C on hold, and (2) T sends *refer* to the conference server, requesting it to call E. The change from Snapshot 3 to Snapshot 4 is initiated by three major messages: (1) T sends *refer* to C, requesting it to *replace* its current dialog with one to E. (2) T sends *bye*

to the conference server. (3) S invites T.

There are many differences between the two implementations. First, unlike the StratoSIP implementation, the *refer* implementation has no separation of signaling and media—the media paths are the same as the signaling paths. This made the *refer* solution much easier to invent than the StratoSIP solution.

Second, the StratoSIP solution composes with features in other network elements, while the *refer* solution does not. With the StratoSIP solution, if there are other network elements (proxies, application servers) in the signaling paths between the endpoints and StratoSIP features, the other network elements will continue to work as expected. Because the *refer* implementation allows endpoints to instruct each other directly to destroy dialogs and create new ones, if there are network elements in the original signaling paths, most likely they will be lost when the signaling paths move.

Third, whether the StratoSIP features are located in the network or in the endpoints, a StratoSIP feature composes with other StratoSIP features, while the *refer* features do not compose with other *refer* features. The StratoSIP AT and TM programs are the same, whether they are running alone or together. To make *refer* implementations of AT and TM work together, on the other hand, it is necessary to rewrite both programs completely.

Fourth, the StratoSIP solution requires far less signaling. With the StratoSIP features running in T's endpoint, to implement the example scenario from no calls at all to Snapshot 4 (except with no TM *Postmortem* state) requires 48 messages between endpoints. To perform the same work, the *refer* solution requires 81 messages.

Fifth, the StratoSIP solution only requires endpoints to know basic SIP,[6] while the *refer* solution requires implementation of several extensions.

Sixth, in Snapshot 4, the StratoSIP solution has a signaling hairpin because the signaling path between C and E goes through AT (wherever it is located), even though AT and its subscriber T are no longer involved in this connection. The *refer* solution has no signaling hairpin. We will address this disadvantage in future work.

10. FUTURE WORK

The version of StratoSIP described in this paper is our first, and not surprisingly needs many improvements.

We have already planned two major extensions for features that StratoSIP does not currently handle. One extension will make it possible to program a feature handling an arbitrary number of active dialogs. Currently there is a distinct dialog variable for each active dialog, which means that a program can handle only a fixed number of them. The other extension will make it possible to control separate media channels separately. Currently all the media channels in a session description must be controlled as a unit, because session descriptions are not dissected.

Another aspect of StratoSIP that needs improvement is its integration with full SIP. The additional SIP messages that can carry session descriptions are reliable provisional responses, *prack* requests and responses, and *update* requests and responses. Although compositional media control does not require them, there is no intrinsic reason why the implementation cannot be extended to handle them when they are received. The language should also be extended to handle *subscribe* and *notify* requests within invite dialogs. These are problematic only because of their interaction with the definition of invite dialogs (RFC 5057).

Two remaining aspects of SIP are, from our perspective, very troublesome. One is *refer* requests, for the reasons explained in Section 9. The other is the fact that SIP signaling is not necessarily FIFO [13]. We think that SIP signaling should be FIFO, and that *refer* requests should not be used except within well-defined boundaries such as those of IP PBXes. Further study will be needed to decide whether to accommodate them in StratoSIP.

The generation of progress tones such as "ringback" and "busytone" is surprisingly tricky, for two reasons. One reason is that progress tones are a major source of feature interactions; the other reason is that SIP as implemented in most endpoint devices is overly restrictive. Both of these issues are explained in detail in [12]. It would be worthwhile to develop some general strategies for correct and robust handling of progress tones, and to build them into StratoSIP.

Another important area for future work is optimization. In particular, it would be nice to get rid of the StratoSIP hairpin mentioned in Section 9. There is no conceptual problem, because when there is a signaling hairpin through the Attended Transfer feature, it has become permanently transparent. The challenge is to design and verify a distributed algorithm, initiated by AT, to remove AT from the signaling path.

Finally, we are very interested in *converged* services, combining telecommunications with Web services. At present StratoSIP programs have embedded Java code to interface with Web services. In the future, we hope to support convergence with more abstract mechanisms.

11. CONCLUSION

Although StratoSIP is a new language, its major abstractions have evolved from research that has been going on for ten years. Their long history of experience and refinement gives us confidence that they are useful and sound. The design of StratoSIP shows that they can be combined in a simple and attractive language.

Using StratoSIP, it is not difficult to program SIP B2BUAs that perform media control. The programs are guaranteed to use SIP signaling correctly, to compose automatically with other StratoSIP programs, and to preserve end-to-end signaling whenever possible.

As StratoSIP matures and becomes ready for use outside our laboratory, it should improve the implementation quality of deployed SIP services. It might also cause a significant increase in the number of people who are qualified to develop them.

Acknowledgments

We are grateful for the many contributions of our colleagues, Hal Purdy and Venkita Subramonian, to this work.

12. REFERENCES

[1] G. W. Bond. An introduction to ECharts: The concise user manual. `http:// echarts.org`.

[6] Section 6 introduced the *pre* tag, which is our own nonstandard extension to SIP. However, endpoints never need to send or interpret these tags.

[2] G. A. Campbell and K. J. Turner. Policy conflict filtering for call control. In *Proceedings of the Ninth International Conference on Feature Interactions in Software and Communication Systems*, September 2007.

[3] E. Cheung. Implementing endpoint services using the SIP Servlet standard. In *Proceedings of the Fifth International Conference on Networking and Services*. IEEE, April 2009.

[4] E. Cheung and P. Zave. Generalized third-party call control in SIP networks. In *Proceedings of the Second International Conference on Principles, Systems and Applications of IP Telecommunications*, pages 45–68. Springer-Verlag LNCS 5310, 2008.

[5] ECharts for SIP Servlets (E4SS). `http://echarts.org/ ECharts-for-SIP-Servlets`.

[6] M. Jackson and P. Zave. Distributed Feature Composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, 24(10):831–847, October 1998.

[7] JSR 289: SIP Servlet API Version 1.1. Java Community Process Final Release, `http://www.jcp.org/ en/ jsr/ detail?id=289`, 2008.

[8] N. Palix, C. Consel, L. Réveillère, and J. Lawall. A stepwise approach to developing languages for SIP telephony service creation. In *Proceedings of the First International Conference on Principles, Systems and Applicatons of IP Telecommunications*, pages 79–88. ACM SIGCOMM, 2007.

[9] J. Rosenberg, J. Lennox, and H. Schulzrinne. Programming Internet telephony services. *IEEE Internet Computing*, 3(3):63–72, 1999.

[10] V. Subramonian. KitCAT: A tool for Converged Application Testing, 2009. Submitted to IPTComm.

[11] X. Wu and H. Schulzrinne. Handling feature interactions in the Language for End System Services. In *Feature Interactions in Telecommunications and Software Systems VIII*, Amsterdam, 2005. IOS Press.

[12] P. 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.

[13] P. Zave. Understanding SIP through model-checking. In *Proceedings of the Second International Conference on Principles, Systems and Applications of IP Telecommunications*, pages 256–279. Springer-Verlag LNCS 5310, 2008.

[14] P. Zave. Modularity in Distributed Feature Composition. In B. Nuseibeh and P. Zave, editors, *Software Requirements and Design: The Work of Michael Jackson*. To appear, 2009.

[15] P. Zave and E. Cheung. Compositional control of IP media. *IEEE Transactions on Software Engineering*, 35(1), January/February 2009.

[16] P. Zave and M. Jackson. A call abstraction for component coordination. In *Proceedings of the Twenty-ninth International Colloquium on Automata, Languages, and Programming: Workshop on Formal Methods and Component Interaction*. University of Málaga, 2002.

APPENDIX

Text of the Attended Transfer program:

```
free box AttendedTransfer(ATtoJava atToJava)

declarations {
  Call c, t, e;
  NonSIPport ui;
<*Address expertAddr;*>
}

initialization {
<*expertAddr = atToJava.getExpertAddr();*>
}

graph {
initial state Init { };
  transition Init ->
    rcv(c) / t = ctu(c) -> SettingUp;

stable state SettingUp { c <-> t };
  transition SettingUp ->
    succeeded(t) / ui!Talking(t.src) -> Talking;

stable state Talking { c <-> t };
  transition Talking ->
    ui?Consult / e = rev(t,dest=expertAddr)
                     -> Consulting;

stable state Consulting { c, e <-> t };
  transition Consulting -> ui?Resume / end(e)
                           -> Talking;
  transition Consulting -> ended(e) -> Talking;
  transition Consulting ->
    ended(c) / ui!Abandoned -> Abandoned;
  transition Consulting ->
    ui?Transfer / end(t) -> Transferred;
  transition Consulting -> ended(t) -> Transferred;

stable state Abandoned { e <-> t };

stable state Transferred { c <-> e };
}
```