The StratoSIP Manual

Pamela Zave, Gregory W. Bond, Eric Cheung, and Thomas M. Smith
AT&T Laboratories—Research, Florham Park, New Jersey USA

16 April 2012

1 Introduction

StratoSIP ("SIP at a very high level") is a domain-specific language for programming SIP applications. This manual documents Version 1.0 of StratoSIP.

The runtime environment for a StratoSIP program is a SIP Servlet container [6]. StratoSIP is compiled into ECharts for SIP Servlets (E4SS) [3]; E4SS is, in turn, compiled into Java.

StratoSIP is designed to enable all application programmers, even those with minimal knowledge of SIP, to program SIP applications easily and correctly. At the same time, it provides SIP experts with sufficient control over the detailed signaling behavior of their programs.

This manual is also designed to serve both audiences. Its main text is intended for ordinary application programmers. For SIP experts, there is supplementary information in two places. Any section may have a supplementary part at the end labeled "**For SIP experts**". In addition, footnotes clarify statements in the main text that might seem oversimplified and therefore confusing to a SIP expert.

2 The StratoSIP view of SIP

2.1 Dialogs

In SIP, an address is a Universal Resource Indicator (URI). In SIP a URI typically represents a person or a telecommunications device.

A StratoSIP program manipulates *SIP Invite dialogs,* or *dialogs* for short. Intuitively, a dialog corresponds to a telephone call between people or a multimedia session between telecommunications devices.

A dialog begins when a SIP entity, the *initiator* of the dialog, sends an initial *Invite* message. An invite message can have many fields. In the current version of StratoSIP, the two most important fields are its *From* field, called its *source* here, and its *Request URI* field, called its *destination* here. The source and destination fields of an invite message are of type URI.

The invite message is routed to another SIP entity, the potential *acceptor* of the dialog, where it is treated as a request that elicits a success or failure response. If the response is failure, then there is no dialog (and the response carries the reason for the failure). If the response is success, then a dialog is established. In addition, it is presumed that the two endpoints of the dialog are ready and able to have media communication with each other. The dialog persists until it is destroyed at the instigation of one of its endpoints, at which time the media channels are also torn down.

A dialog can be associated with any number of media channels of any media. These channels are selected by the endpoints, and the selection can change during the life of the dialog. In the current version of StratoSIP, all the media channels of a dialog are treated alike and handled as a bundle by the language features. The examples in this manual can be understood most easily by assuming that each dialog has a single two-way voice channel and possibly a single two-way video channel.

StratoSIP 1.0 handles the messages and behaviors in the basic SIP standard [8]. It also includes the *Info* message, which is defined in an extension [2].

For SIP experts:

StratoSIP is a language for manipulating invite dialogs and controling media streams, and therefore does not handle *Register* and *Options* requests, although they are defined in the basic standard.

Many extensions to the basic SIP standard have been defined [7], but they are not necessary for the StratoSIP mission. The power and generality of basic SIP for controling media streams can be seen from the examples and comparisons in [12].

2.2 Back-to-back user agents

A StratoSIP program is instantiated and run as a concurrent process. Each such instantiation is called a *box.* From the perspective of SIP, a box is a *back-to-back user agent* or *B2BUA*. From the perspective of a SIP Servlet container, a StratoSIP program is an

application, and a box is a *SIP application session*. From the perspective of Java, a StratoSIP program is a Java class definition, and a box is a class instance.

B2BUAs are powerful SIP entities. As such, they require a slightly richer vocabulary of concepts than used in Section 2.1. For example, Figure 1 illustrates one of the ways that B2BUAs can be invoked in a SIP Servlet container.

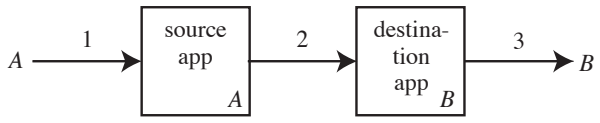

Figure 1: Source and destination applications invoked on the same call. All three dialogs have *source* = *A* and *destination* = *B*.

The figure shows a call from a caller device with URI *A* to a callee device with URI *B*. To implement the call, there are three numbered SIP dialogs and two B2BUAs or boxes. The leftmost box is the acceptor of dialog 1 and the initiator of dialog 2. The rightmost box is the acceptor of dialog 2 and the initiator of dialog 3. Although the boxes are the endpoints of some dialogs, only the devices are the endpoints of media streams. Any media control performed by the boxes is done on behalf of the devices.

In a SIP Servlet container, a URI can subscribe to a particular application in the *source region,* the *destination region,* or both. If a container is handling an invite message with *source* = *A*, and *A* subscribes to a StratoSIP application in the source region, then the container will instantiate the StratoSIP program as a box and route the invite message to it. Similarly, if a container is handling an invite message with *destination* = *B*, and *B* subscribes to a StratoSIP application in the destination region, then the container will instantiate the StratoSIP program as a box and route the invite message to it. This is called *application routing*.

The SIP Servlet standard provides a small amount of history in invite messages so that applications can be invoked correctly. For example, the configuration of Figure 1 would arise in a container with an application subscribed to by *A* in the source region and an application subscribed to by *B* in the destination region. The history is needed to differentiate the invite messages of the three dialogs; although each has the same source and destination fields, each must be routed to a different SIP entity.

The SIP Servlet standard allows the deployer of a container to choose its application router. Some application routers might invoke applications based on different fields of the invite message, or on different criteria altogether. The examples in this manual all assume application routing based on source and destination fields, however.

It is possible to decompose a source or destination application into multiple modular applications, because a SIP Servlet container will automatically compose the modules at runtime. This capability is called *application composition.* StratoSIP's support for application composition is discussed in Section 9. Note that application composition can mix B2BUAs and proxies freely.

3 Language overview

3.1 Text and graphical views

Figure 2 is an example StratoSIP program for the Redirect to Voice Mail (RVM) application. RVM behaves transparently while it waits for the outcome of a call. If the call succeeds, then RVM continues to behave transparently. If the call fails, then RVM connects the caller with a voicemail server, which will prompt for and record voicemail from the caller. In the program, comments, user-chosen identifiers, and white space follow the same lexical rules as in Java. Embedded Java code is delimited by `<*...*>`.

A program in StratoSIP may begin with an optional package definition having the same syntax as a Java package definition. Next, there can be an optional block of embedded Java code containing package imports (neither of these pieces is shown in Figure 2). The essential part of the header takes one of these forms:

`free box` *boxName*
`bound box` *boxName*

Box names have their own namespace, so these can overlap with other names used in the program. The box name may be followed by optional Java parameters, enclosed in parentheses. The parameters may be followed by an optional block of Java code for an inheritance declaration (not shown in Figure 2).

After these headers, the program contains *sections* in any order. Each section begins with a section type, and is delimited with set brackets. There must be at least one section of type `declarations` and at least one section of type `graph`. There can be one or zero `initialization` sections.

The appendix is a complete grammar of the StratoSIP language. It also includes name spaces and reserved words.

```
free box RedirectToVoiceMail
   (<* RvmToJava rvmToJava *>)

declarations {
   Dialog i, o, v;
<* URI voicemail *>;
}

initialization {
<* voicemail =
   rvmToJava.getVoicemailResourceURI() *>;
}

graph {
initial state Init;
   transition Init -> rcv(i) / o = ctu(i)
                                -> Waiting;

stable state Waiting { i <> o };
   transition Waiting -> succeeded(o)
                         -> Transparent;
   transition Waiting -> ended(o) /
      v = new(<* voicemail *>)
                          -> Redirected;

stable state Transparent { i <> o };

stable state Redirected { i <> v };
}
```

Figure 2: Example StratoSIP program for the RVM application.

Figure 3 is another view of the StratoSIP RVM program. This is a graphical notation that we often use to display the `graph` sections of a program. The graphical notation for each concept will be introduced along with the concept.

3.2 Additional context and instantiation

In addition to sending and receiving messages through SIP dialogs, a box can receive messages from Web services and other Java components through *non-SIP interfaces*. A box can communicate with *timers* by sending and receiving messages. SIP dialogs, non-SIP interfaces, and timers are all similar in the StratoSIP syntax. In each case, a box receives messages through an *input queue* associated with the entity, and sends messages through an *output queue* associated with the entity.

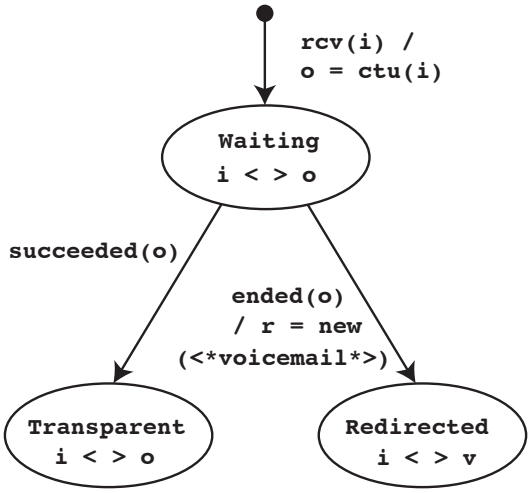

Figure 3: A graphical view of the StratoSIP RVM program.

Furthermore, a StratoSIP program can contain ordinary Java code through which a box accesses (invokes methods in) Web services, databases, and other Java components.

StratoSIP programs are usually instantiated by an application router running in the SIP Servlet container (the exception is discussed below). Application routers process initial invite messages, which are the ones that create dialogs. This processing often includes program instantiation.

A StratoSIP program must be declared in its header as either *bound* or *free*. The difference arises when an application router decides to route an initial invite to an application defined by a StratoSIP program, on behalf of a subscriber URI. If the application is defined by a *free* StratoSIP program, then the router makes a fresh instance of the program, and puts the invite in this instance's *router input queue*.

If the application is defined by a *bound* StratoSIP program, on the other hand, the program can have at most one instance at a time for each of its subscribers. If it is already instantiated for the subscriber, then the router routes the invite to the existing instance rather than make a new one. As with new instances, the router puts the invite in this instance's *router input queue*.

As a consequence of this difference, an instance of a free program can receive at most one initial invite message per instance, while an instance of a bound program can receive any number.

Occasionally the input message that causes instantiation of a program is a non-SIP message rather than

a SIP invite. The instantiating program, which may be a Web service, is responsible for choosing and instantiating the program, creating an association between itself and a non-SIP interface in the box, and putting the message in the interface's input queue.

For SIP experts:

In the terms of the SIP Servlet standard, a box or program instance is a *SIP application session.* Bound boxes are implemented using the *session key* mechanism in SIP Servlet containers.

3.3 Object classes

Variables of all types used in a StratoSIP program must be declared in a declarations section. Declarations have a Java-like syntax, as shown in Figure 2.

StratoSIP has four built-in object classes Dialog, Message, Timer, and NonSipInterface. Objects of these classes have very particular language-specific semantics, which are explained in Sections 4, 5, 6, and 7, respectively. Variables of these types cannot be initialized in an initialization section.

The built-in object classes have corresponding Java classes of the same names. Furthermore, the Dialog and Message classes offer interfaces that can be used by StratoSIP programs in embedded Java code. For example, methods of the Java Dialog class can be used to access the history of a dialog, and methods of the Java Message class can be used to manipulate SIP messages.

In addition to the four object classes native to StratoSIP, there are two domain-specific Java classes used in StratoSIP programming: URI and MessageModifier. Variables of type URI and MessageModifier, as well as ordinary Java classes, can be initialized using embedded Java code in an initialization section. Programmer interfaces to the Java object classes Dialog, Message, URI, and MessageModifier are described in Section 8.

A StratoSIP program is compiled into a Java class which is a subclass of the Java class Box. All the local variables of a box, whether built-in or declared, are variables of this subclass and may be accessed by the programmer.

3.4 States and annotations

At the top level of control, a StratoSIP program is a finite-state machine. All the *states* and *transitions* of the finite-state machine are found in the `graph` sections of the program. Graph sections, states, and transitions can be in any order, as their textual order has no significance.

A state is introduced with a statement in one of these forms:

`initial state` *stateName* `;`
`stable state` *stateName* `{` *annotation* `}` `;`
`transient state` *stateName* `;`
`terminal state` *stateName* `;`

State names have their own namespace, so they can overlap with other names used in the program.

A program must have exactly one *initial state.* When a program is first instantiated, its initialization section is executed (if any), and then it enters its initial state. An initial state has no in-transitions, and must have out-transitions.

A program can have any number of *terminal states*, which have in-transitions but no out-transitions. A program need not have any explicit terminal states, because termination is often handled implicitly.

A program can have any number of *stable* and *transient* states, each with in- and out-transitions. When a program is quiescent in an initial or stable state, it is examining its input queues for input messages, and will react as soon as it finds a non-empty queue and reads its first message. If more than one input queue contains a message, the program will choose one nondeterministically and read its first message. A program in a transient state, in contrast, does not react to inputs.

In a graphical view, the initial state is represented as a black dot, stable states are ovals, transient states are small circles, and terminal states are black bars. Figure 4, which is the program for the Quiet Time (QT) application, has initial, stable, and terminal states. QT prompts the caller to find out whether the call is urgent or not. If urgent, the box allows the call to proceed and behaves transparently. If not urgent, the box causes failure.

Stable states must be annotated as well as named. These *annotations* have the important function of declaring the desired media states of the devices to which its SIP dialogs connect.

For example, in the waiting and transparent states of RVM, `i` and `o` are *dialog variables* referring to the program's two current SIP dialogs. The annotation `i < > o` declares the programmer's intention that the endpoint devices reached by these dialogs should be connected. The details of the media channels will be negotiated using SIP signals not visible to the StratoSIP programmer.

There are also state annotations concerning signaling and timers; see Sections 5 and 6, respectively.

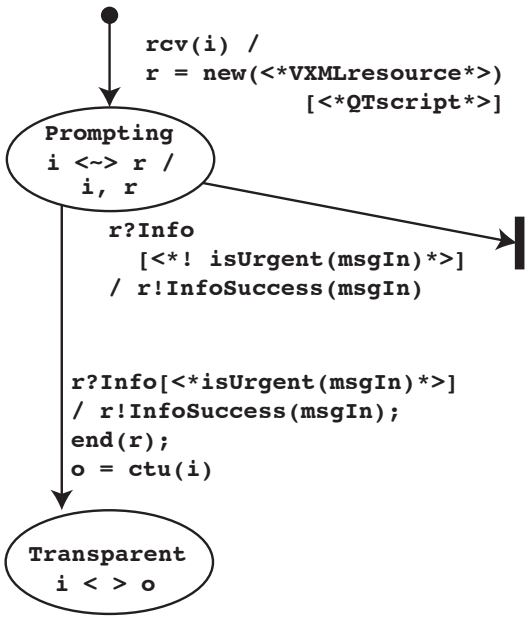

Figure 4: StratoSIP program for the Quiet Time (QT) application. The black dot is the initial state, the ovals are stable states, the black bar is a terminal state, and the arrows are state transitions.

3.5 State transitions

A *state transition* always has a *current state, guard,* and *next state*. It may also contain a sequence of *actions*, internally separated with semicolons and separated from the guard with a slash. The usual syntax of a transition is:

`transition`
currentState `->` *guard* `/` *actionSequence* `->` *nextState* `;`

Note that the slash and action sequence are optional.

A transition is triggered when its *current state* matches the current state of the program, and its guard becomes true. It is executed as follows:

1. If the transition contains actions, execute them in the order listed.
2. If there are any entrance actions associated with the next state of the transition, execute them.
3. Enter the next state.

Note that entrance actions are implicit in StratoSIP semantics, rather than being programmer-defined. An example of an entrance action is given in Section 3.8.

In a graphical view, a transition is represented as an arrow from its current state to its next state. The guard and action sequences of the transition are written as a label on the arrow.

3.6 Guards

Transitions out of initial or stable states and transitions out of transient states have different guards. A guard out of an initial or stable state has the form:

inputPredicate `[` *messagePredicate* `]`

where the message predicate and its enclosing brackets are optional.

An input predicate is made true by the reading of a message in an input queue. All the input predicates are built-in and domain-specific; they are discussed in Sections 4 through 7 and listed in the grammar. Message predicates, on the other hand, are always Java code that evaluates to a boolean. Within a message predicate, the received message can be referred to by using the built-in variable `msgIn` of type Message. If a guard has both predicates, then it is true only if both predicates are true of the same input message.

The triggering of transitions out of initial or stable states works as follows:

1. Choose a non-empty input queue, read the first message from it, and assign this message to `msgIn`.
2. Find a transition from the current state whose guard is made true by the input message. If more than one transition has a true guard, choose one nondeterministically.
3. If the chosen true guard has side-effects, execute them.
4. Execute the remainder of the transition as specified in Section 3.5.

It is very important to note that the StratoSIP semantics defines a default, implicit transition from every state on every possible input message. These implicit transitions can be over-ridden by explicit transitions written in the program. Because of these default transitions, all StratoSIP programs can handle any input at any time. This guarantees they have the property of being *input-enabled*, which in turn guarantees that they cannot deadlock.

Implicit transitions on input messages, out of initial and stable states, are discussed in Sections 4.2 (initial invites), 4.4 (media control), 4.6 (dialog termination), and 5.2 (status messages).

Transitions out of transient states are guarded by *state predicates* enclosed in brackets. Most state predicates are written in Java, although there are two built-in ones (see Section 4.5). As soon as a program enters a transient state, these predicates are evaluated in the context of the current state variables, and a transition whose guard is true is executed. The guard `!` represents the negation of the disjunction of the state predicates guarding all other out-transitions from the same state, so it acts as an "else." A transient state must always have one out-transition guarded by `!`.

3.7 Actions

Most actions are built-in and domain-specific. Actions can:

- create dialogs (Section 4.3),
- destroy dialogs (Section 4.6),
- re-assign dialog variables (Section 4.7),
- send status messages within dialogs (Section 5.1),
- alter implicit handling of status messages (Section 5.2),
- set and cancel timers (Section 6),
- and send messages through non-SIP interfaces (Section 7).

An action can also consist of embedded Java code. All actions are listed in the grammar.

Actions that send messages can be enhanced by programmer-defined *message modifiers*. Semantically, a message modifier is an object of type MessageModifier that can modify messages. Syntactically, a message modifier is enclosed in brackets and written as embedded Java code (which must evaluate to an object of type MessageModifier). The various uses of message modifiers are explained in context. The MessageModifier class is documented in Section 8.

Many elements of StratoSIP have *argument lists*, which are always delimited by parentheses. An argument list may have one required argument, which must be listed first if present. An argument list may also have one or more optional arguments, which may be listed in any order after the required argument. Optional arguments are identified by *field names*. For example, here is a language element with one required argument and one optional argument:

```
ctu( dlg, src = <*myURI*> )
```

The optional argument with field name `src` requires a value of type URI. Although `myURI` is just a variable name, it must be delimited as Java code because variables of type URI are not native to StratoSIP.

At the end of the grammar in the appendix, a table gives the required and optional arguments for each language element with an argument list.

3.8 Program termination

A terminal state is one in which a StratoSIP program can terminate. Every terminal state has an implicit entrance action that ends all active SIP dialogs. There are no explicit transitions out of terminal states.

Typically, a box in a terminal state lives long enough to handle (automatically) all the messages needed to clean up the ended dialogs, then terminates and is destroyed.

It is possible for a box in a terminal state to receive messages other than the ones related to ended dialogs. Messages from timers and non-SIP interfaces will be ignored.

The one special case occurs when a bound box in a terminal state receives a new initial invite message. In this case the box goes immediately into its initial state, from which the box reacts to the message. When the box moves from the terminal state to the initial state, the initialization section is not re-executed, and the Java state does not change. Meanwhile the cleanup of ended dialogs continues in the background.

3.9 Completeness and correctness

SIP is a very complex protocol. Even for experts, it is extremely challenging to program a SIP application that conforms perfectly to the SIP standard, handles all possible events and conditions correctly, and reliably implements the desired functions.

StratoSIP programmers do not have to worry about this complexity, because the language handles it automatically. By means of defaults and runtime libraries, the StratoSIP implementation provides the following guarantees:

- All transactions are completed correctly.
- All dialogs are terminated correctly.
- All SIP race conditions are handled correctly.
- All runtime failures are handled correctly.
- Application programs cannot deadlock.
- Matching of protocol states between dialogs is done correctly.
- Offer/answer negotiation of media channels is used correctly.

The runtime libraries that provide most of these guarantees are based on formal models that have been verified with the model-checker Spin [4].

In addition, static semantic analysis of StratoSIP programs detects and diagnoses a wide variety of possible programmer errors. For example, Section 4 presents many constraints on how dialog variables can be used. StratoSIP programmers need not worry about what will happen if they violate these constraints, because static analysis during compilation will catch all such violations.

4 Dialog states

4.1 Active dialogs

SIP *Invite* dialogs and dialog variables are the central concepts in StratoSIP. Each dialog in which a program is participating has a unique internal identifier in the program state. Although dialog identifiers are like opaque pointers in not being directly readable by programs, they are the values of dialog variables.

In the RVM program (Figure 3) there are three dialog variables: `i` (for "incoming") refers to the dialog connecting the box with the caller, `o` (for "outgoing") refers to a dialog potentially connecting the box with its subscriber (who is the callee), and `v` (for "voice mail") refers to a dialog connecting the box to a voice mail server.

Dialog variables must be declared in a declarations section. These declarations follow Java syntax, as can be seen in Figure 2. There is a distinguished "no dialog" identifier "`-`" to which all dialog variables are automatically initialized. Dialog variables never appear in initialization sections.

Active dialogs are the dialogs currently being controlled explicitly by the program. Each active dialog must have a dialog variable that refers to it.

Figure 5 shows the predefined states and events of an active dialog in StratoSIP. The StratoSIP view of a dialog is an abstraction of the SIP view. In the StratoSIP abstraction, the events shown are atomic operations, and the SIP messages that implement the events are hidden.

An active dialog is in one of the three states *Incoming, Outgoing,* or *Succeeded.* A dialog may exist after leaving these states, while it is being cleaned up, but it can no longer be observed or controlled by the program.

The set of active dialogs is so important that each stable state must list, in its annotation, the variables referring to the dialog active in that state. It is not possible to have active dialogs in initial or terminal states.

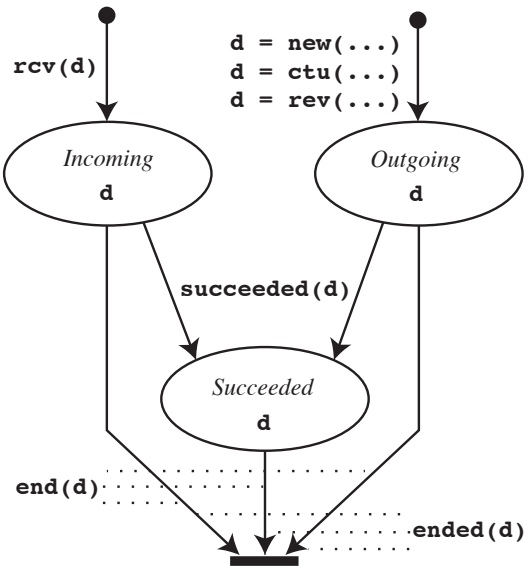

Figure 5: A finite-state machine showing the states and events of an active dialog in StratoSIP. This is not a StratoSIP program, although it uses similar notation.

4.2 Guards that create dialogs

`rcv(`*dialogVar*`)` is an input predicate that is made true by an initial invite message in the box's router input queue. As side-effects of this input predicate, a new dialog is created, the dialog enters the *Incoming* state, and the dialog's identifier is assigned to the dialog variable *dialogVar*.

Almost all StratoSIP programs have a transition with a `rcv` input predicate coming out of their initial states. (The exceptions are programs instantiated by non-SIP messages, as discussion in Section 3.2.)

Each initial or stable state without an explicit transition guarded by `rcv` has the following implicit transition:

```
transition
state -> rcv(unwanted)/end(unwanted) -> state ;
```

This transition rejects an invite without affecting the box state in any way. Without it, the box might not be input-enabled.

A `rcv` guard can be followed by an optional message predicate, which can refer to fields of the invite message. If the message predicate is not satisfied by the invite message, then the invite message does not make the overall guard true. If the invite message does not make any other explicit guard true, it is handled by the implicit transition above.

For SIP experts:

Another side-effect of `rcv` is to send a SIP 183 message to establish the dialog.

4.3 Actions that create dialogs

A new dialog enters the *Outgoing* state as a result of one of these actions, all of which create and send an initial invite message:

- *dialogVar2* = `new(`*URI*`)`
- *dialogVar2* = `ctu(`*dialogVar1*`)`
- *dialogVar2* = `rev(`*dialogVar1*`)`

In each case, *dialogVar2* points to the new dialog.

The functions `new, ctu,` and `rev` all tell the implementation something about the role of the new dialog with respect to other dialogs being handled by the box. This information helps to determine some of the fields of the initial invite message. It also guides application routing and, if it is being used, application composition (see Section 9). Examples of the use of the three functions will be given shortly.

Each use of `new` requires an argument of type URI that will be the destination of the new dialog. This argument can also be written in a named style as `dest =` *URI*. A `new` can also have an optional `src` argument of type URI that will be the source of the new dialog.

Each use of `ctu` or `rev` requires a "seed" dialog whose invite is used to make the invite for the new dialog. This argument can also be written in a named style as `seed =` *dialogVar1*. A `ctu` or `rev` can also have optional `src` and/or `dest` arguments of type URI. If either of these optional arguments is not present, then the corresponding source or destination field of the invite message will be copied from the seed dialog.

The function `ctu` (*continue*) is the most common of the three functions. It is used to create a dialog that is continuing a chain of dialogs from caller to callee, such as the chain in Figure 1, in a normal way. When the chain in Figure 1 was assembled, each B2BUA used its incoming dialog as a seed for its outgoing dialog.

The use of `ctu` is illustrated by the Attended Transfer (AT) program in Figure 6. This application enables a trainee agent in a customer-service center to transfer a customer call to a more experienced agent. A consult command from a non-SIP interface causes the program to put the customer on hold and connect the trainee with an expert. From the consulting state, the trainee can resume talking to the customer or transfer the customer to the expert.

On receiving a dialog from the customer in its initial state, the program continues the dialog; this is transparent behavior for the call as a whole. The

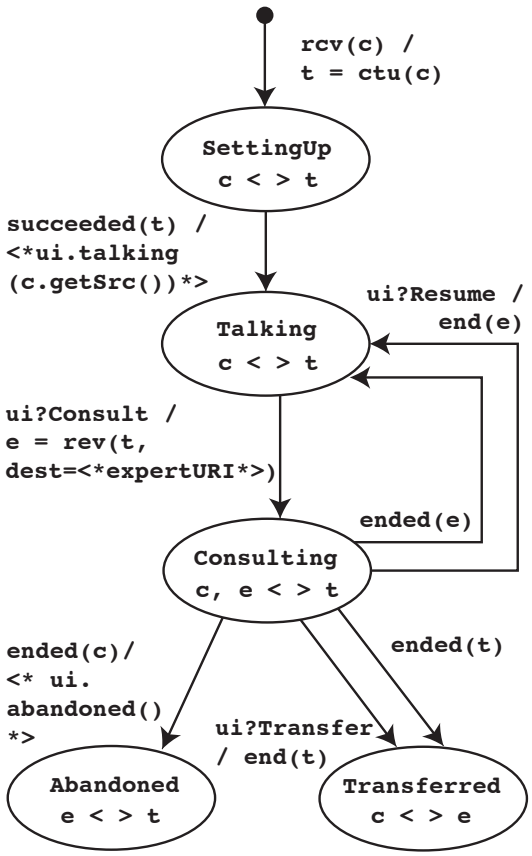

Figure 6: StratoSIP program for the Attended Transfer (AT) application.

program will remain transparent unless and until it gets a consult command from the trainee, who is its subscriber.

The function `new` is used to create a dialog that is a new branch of the assembled graph of applications, with little relationship to existing dialogs. `new` is usually used to reach a media server, and is also in common use.

For example, the QT program (Figure 4) uses `new` to create a dialog to a VoiceXML server, which is a server that participates in interactive voice-response sessions, following a script written in VoiceXML. This server will be connected to the caller, and will use announcements, prompts, and touch-tone recognition to discover whether the caller thinks the call is urgent or not. The program uses `new` because the server is a temporary part of the call, with no real relationship to the caller or callee. In QT the dialog made with `new` has a placeholder source URI, first because there is no obvious candidate to be the source, and second because there is no need to route the dialog to any

applications on behalf of a source URI.

The function `rev` (*reverse*) is used least often. It is used to create a dialog that is continuing a dialog chain from the seed dialog, but in doing so is reversing the source and destination roles.

The use of `rev` is also illustrated by the AT program. If and when the program gets a consult command, it must connect the trainee to an expert. To do this, it creates a dialog seeded by dialog `t`, which is the dialog that connects AT to the trainee. However, in `t` the trainee is playing the destination role, while in the created dialog `e` the trainee must play the source role. For this reason, `rev` rather than `ctu` is used.

One of the effects of using `rev` is that the source and destination fields, which are taken by default from the seed dialog, are swapped. By default this would make the customer the destination of dialog `e`. This default is over-ridden by the optional argument `dest = <*expertURI*>`, so that the destination URI of the created invite message will be the expert's URI.

Because the functions `new, ctu,` and `rev` all create messages, each call can be followed by a message modifier that modifies the invite before it is sent. Often, the purpose of the message modifier is to add fields. For example, `new` in the QT program (Figure 4) has a message modifier (pointed to by the variable `QTscript`) that adds a field passing a VoiceXML script to the server.

In StratoSIP the operations that create dialogs all appear to be atomic and instantaneous. In particular, the implementations of the dialog-creating actions 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.

For SIP experts:

The SIP signaling compiled from `new, ctu,` and `rev` assumes that a final response to the initial invite message may come slowly, because it may entail getting a response from a person.

In some cases the programmer knows that the initial invite message is being sent to a machine such as a media server, so that the response will come quickly or not at all. In these cases, the programmer can use an optional `fastResponse` argument with this syntax:

```
r = new( <*VXMLresource*> ,
         fastResponse = <*true*> )
```

This tells the compiler to generate a different and more efficient signal sequence that assumes a fast response. For example, the dialog to the media server in the QT program could be a fast-response dialog.

4.4 State annotations for media control

As mentioned in Section 4.1, active dialogs are so important that the dialog variable for each active dialog must be listed in the annotation of each stable state. The form of the annotation controls media flow.

The annotation consists of comma-separated clauses. The most common form of clause is `i < > o`, which is called a *flowlink*, and can be found in every example program. It means that the endpoint reached by dialog `i` and the endpoint reached by dialog `o` should be connected by whatever media channels the endpoints desire. If, on the other hand, a dialog variable is by itself in an annotation, that means that the dialog should be on hold, with no media flow.

For example, in the AT `Talking` state (Figure 6), the customer `c` is media-connected to the trainee `t`. In the `Consulting` state the customer is on hold and and the trainee is media-connected to the expert `e`. In the `Abandoned` state the expert is media-connected to the trainee, and in the `Transferred` state the customer is media-connected to the expert.

Conferences are built from these two-way media connections, as shown in the program for the Trainee Monitoring (TM) application (Figure 7). This application enables the supervisor of a trainee agent in a customer-service center to monitor the performance of the trainee. When the application subscribed to by the trainee receives a dialog from the supervisor while the trainee is talking to a customer, the application forms a conference. When the application needs to form a three-way conference, it initiates three new dialogs to a conference server. It then uses flowlinks to connect each of the three external dialogs to a dialog leading to the conference server. The server mixes the incoming media channels and sends the correct mix on each of the outgoing media channels.

Dialogs might become flowlinked at a time when their protocol states do not match. The StratoSIP programmer need not worry about this, as the implementation will take the necessary steps to bring the dialogs to matching protocol states.

Each of AT and TM has its own view of the correct media paths in each of its states. These views are relative rather than absolute, however, because when applications are composed, the correct media paths are determined by the *composition* of the relevant applications. Figure 8 shows the runtime composition of these applications, in which external dialogs reach the customer, trainee, expert, and supervisor, while `t` in AT and `c` in TM refer to the same dialog. This composition is discussed in detail in [12], including how global media paths are determined by the states

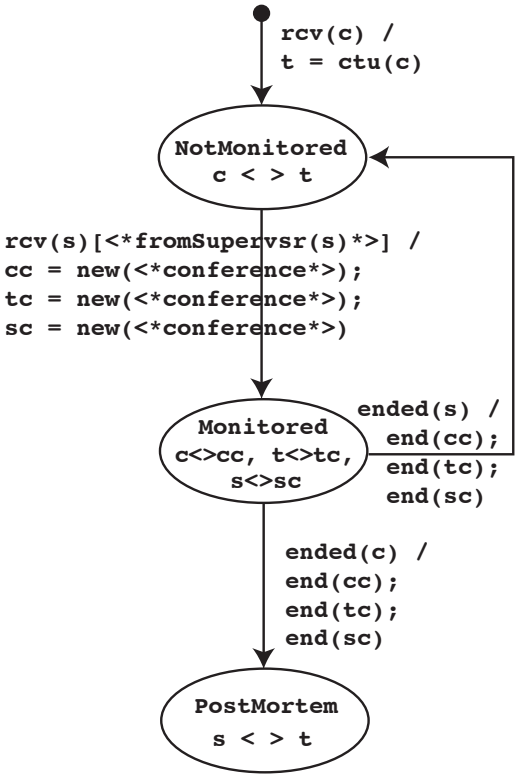

Figure 7: StratoSIP program for the Trainee Monitoring (TM) application.

of both applications.

For SIP experts:

Sometimes, when two dialogs become flowlinked, their SIP protocol states do not match. Their SIP protocol states include their states with respect to initial invite transactions, re-invite transactions, and offer-answer negotiations. In these cases the StratoSIP runtime library works to make their states match as soon as possible. More particularly, it works to set up a media channel or channels between the endpoints to which the dialogs lead, unless one or both endpoints is unwilling.

When the states of the dialogs match, messages of invite transactions (initial or re-invite) can travel end-to-end, without any modification by the application except for the necessary change in dialog identification. StratoSIP preserves this end-to-end transparency whenever possible, so that conversations between endpoints using proprietary headers are not disrupted.

StratoSIP media control is implemented using third-party call control in SIP [1]. Although the declarative style of media control in StratoSIP lightens the burden of application programming enormously, it may hide too much in some cases.

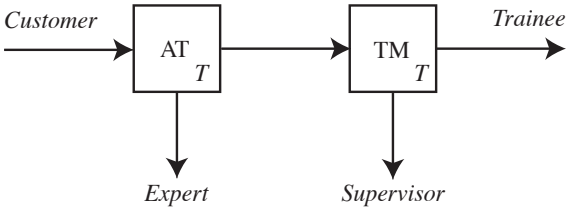

Figure 8: The runtime composition of Attended Transfer (AT) and Trainee Monitoring (TM) applications.

For example, consider a Music on Hold application that is supposed to detect when one endpoint puts the other endpoint on hold, and connect the other endpoint to a source of music. Because the end-to-end hold signal is a re-invite, and StratoSIP hides re-invites from programmers, it is not possible to implement this application in StratoSIP 1.0.

4.5 The *Succeeded* state

An active dialog reaches the *Succeeded* state when its far end is first connected to a desired party. As Figure 5 shows, once a dialog succeeds it remains succeeded, regardless of whether it is put on hold later or not.[1]

An outgoing dialog (created by `new`, `ctu`, or `rev`) succeeds when it receives the right SIP messages from its far end. An incoming dialog (created by `rcv`) succeeds when it becomes flowlinked to a dialog that has succeeded. The caller at the far end of the incoming dialog has been waiting to be connected to a desired party, and when the incoming dialog is flowlinked to a dialog with a desired party at the other end, then the semantics of the flowlink ensure that the caller will be connected to the desired party.

For an outgoing dialog `t`, `succeeded(t)` is an input predicate made true by the input message that causes `t` to succeed. The AT program in Figure 6 uses this input predicate simply to trigger an action that notifies a Web user interface. Because incoming dialog `c` is flowlinked to `t` in the target state of the transition, `c` also succeeds as soon as the program enters the target state `Talking`. However, `succeeded` cannot be used as an input predicate on an incoming dialog such as `c`.

Simply connecting a caller to some source of media is not enough for the caller's dialog to succeed,

[1] *Succeeded* does not correspond directly to any SIP concept. See below for details.

because the media may come from a server used to implement a service rather than from a real desired party. For example, consider the Quiet Time application in Figure 4. In its `Prompting` state, the QT application is connecting the caller to a VoiceXML server for the purpose of finding out from the caller whether the call is urgent. The media server is not the caller's desired party.

In Figure 4, the media linkage between the caller dialog `i` and the server dialog `r` is written with a tilde between the arrowheads. This indicates that it is a *pre-flowlink* rather than a flowlink. A pre-flowlink controls media in exactly the same way as a flowlink. The difference between a flowlink and a pre-flowlink is that the pre-flowlink *will not propagate the succeeded state from* `r` *to* `i`, as a flowlink would.

To understand why this matters, we need to consider Figure 12, which shows the application composition of QT with Redirect to Voice Mail, and Figure 3, which shows the program for RVM. RVM makes its decision on whether to redirect or do nothing based on whether outgoing dialog `o` succeeds or not. Figure 12 shows that `o` in RVM is the same dialog as `i` in QT. If QT allowed `i` to succeed simply because it is connected to a VoiceXML server, and if the caller indicated that the call is not urgent, then the caller would not be redirected to voice mail by RVM. So the purpose of StratoSIP's *Succeeded* state is to help different applications integrate and interact in the best way.

"Interacting in the best way" is necessarily somewhat subjective, which is why StratoSIP gives the programmer control over it. Note that the `Redirected` state in RVM uses a flowlink rather than a pre-flowlink, even though the caller is connected to a server. This reflects the programmer's judgment that a person's voice mail is a good proxy for the person.

Imagine, for example, that there is a Call Forwarding on Failure (CFF) application to the left of RVM in Figure 12. If RVM used a pre-flowlink for the `Redirected` state, then while the caller was recording a long voice message, RVM dialog `i` would not have succeeded. The CFF program could decide that the call is taking too long to succeed, time out, break the connection between the caller and voice mail, and forward the call elsewhere. This is not a good way for the applications to interact.

For any dialog `dlg`, `succeeded(dlg)` can be used as a state predicate, which means that it can be used as a guard on a transition out of a transient state.

For SIP experts:

No SIP concept corresponds exactly to the *Succeeded* state in StratoSIP, which is an abstraction defined to facilitate good integration and interaction among applications. It is implemented by adding special tags to SIP messages. A `succeeded` input predicate can be made true by a re-invite, by a successful response to an initial invite (*Invite 200*), or by a successful response to a re-invite.

A dialog in the *Succeeded* state is always confirmed in the usual SIP sense, although a confirmed dialog is not always *Succeeded*. In cases where it is absolutely necessary to know whether a dialog is confirmed or not, `preSucceeded(dlg)` can be used as either an input predicate on an outgoing dialog, or as a state predicate. A `preSucceeded` input predicate can only be made true by a successful response to an initial invite (*Invite 200*).

4.6 Guards and actions that destroy dialogs

The operations that destroy active dialogs are all atomic and instantaneous from the program perspective.

The input predicate `ended(dlg)` is made true by a received message indicating that the other end of `dlg` wants it to end. The guard can include a message predicate that must also be true of the input message.

The action `end(dlg)` ends an active dialog. The action can include a message modifier that will be applied to whatever message the StratoSIP implementation generates to carry out the action.

If a stable state with active dialog `dlg` has no out-transition guarded by `ended(dlg)`, then it has an implicit transition with this guard, entering a terminal state. In other words, any external dialog end that is not explicitly handled causes the box to terminate.

A terminal state has no active dialogs. To enforce this, when a program enters a terminal state, all remaining active dialogs are ended automatically. These implicit transitions and actions explain why most of the example programs have no explicit terminal states.

For SIP experts:

An `ended` input predicate can be made true by a failure response to the initial invite, a *Cancel*, a *Bye* or a *408* response to a mid-dialog request. StratoSIP automatically generates all necessary responses and waits for all necessary acknowledgments. Although StratoSIP is designed to hide the distinctions among these forms of dialog teardown, they can be probed or manipulated if necessary by message predicates or message modifiers.

Just as invite transactions are handled in an end-to-end manner, with the application being transparent, whenever possible (Section 4.4), bye transactions

are also handled in an end-to-end manner whenever possible.

For end-to-end treatment to be possible, two dialogs `dlg1` and `dlg2` must be flowlinked, and both must be confirmed, so that each is torn down with a bye transaction. Furthermore, there must be a transition between two stable states that is (1) guarded by `ended(dlg1)` and (2) includes the action `end(dlg2)`. This transition could be the implicit transition applicable if there is no explicit transition with guard `ended(dlg1)`. If explicit, the transition would most commonly look like this:

```
stable state Linked { dlg1 < > dlg2 };
transition Linked -> ended(dlg1) /
                          end(dlg2) -> Gone;
```

but it could also look like this:

```
transition Linked -> ended(dlg1) /
              dlg2!Info; end(dlg2) -> Gone;
```

or even this:

```
transition Linked -> ended(dlg1) -> Going;

transient state Going;
transition Going -> [<*overLimit(x)*>] /
                          end(dlg2) -> Gone;
transition Going -> [!] -> HoldingState;
```

In this last example the transition from stable state `Linked` to stable state `Gone` passes through the transient state `Going`.

4.7 Assignments to dialog variables

There are two invariants that dialog variables must satisfy:

- Every active dialog must have a dialog variable pointing to it.
- No two dialog variables ever have the same value unless their value is `-`, which means "no dialog".

Box initialization, dialog creation, and dialog destruction all satisfy these invariants, provided that the variable used in dialog creation does not already point to an active dialog.

Although an active dialog must have a variable pointing to it, a dialog variable need not point to an active dialog or `-`. If the variable pointing to a dialog is not re-used after the dialog ends, then the history of the dialog can still be accessed through the variable. Section 8 gives the interface for accessing dialog history through the Java Dialog object.

The actions that create dialogs (see Section 4.3) are assignment statements to dialog variables. It is also possible to have assignment statements that swap the values of dialog variables, for the purpose of changing the relationship between active dialogs and the roles they are playing in the program. Because these assignments must preserve the invariants, they take the general form of a multi-way swap such as

```
d1, d2 = d2, d1
d1, d2 = d2, -
d1, d2, d3 = d3, ctu(d3), -
```

The general rules for ensuring that a multi-way assignment preserves the invariants are:

- Dialog expressions (`new(...)`, `ctu(...)`, `rev(...)`, `-`) must not appear to the left of the `=`. These denote values, not variables.
- No variable appears more than once to the left of the `=` or more than once to the right of the `=`. Note that an expression such as `ctu(dlg)` does not count as an "appearance" of `dlg`, because using the value of `dlg` as a seed does not affect the variable or its value.
- If a dialog variable appears to the right of the `=`, it must also appear to the left of the `=`. Otherwise, two dialog variables will have the same value. It does not matter whether the variable points to an active or inactive dialog.
- If a dialog variable appears to the left of the `=`, and it points to an active dialog, then it must also appear to the right of the `=`. Otherwise, after the assignment, no variable will point to that active dialog. This rule does not apply to variables pointing to inactive dialogs, because inactive dialogs need not be the values of dialog variables.

The use of dialog variables to denote roles is illustrated by the Call Waiting (CW) application in Figures 9 and 10 (at end of manual). CW is programmed as a bound box, and is subscribed to in both source and destination regions (see Section 2.2). In CW the dialog in variable `subs` always connects the box to its subscriber, and the dialog in variable `far` connects the box to a far party. Because the initial dialog that instantiates the program may have the subscriber as its source (source region) or as its destination (destination region), the message predicates with the `rcv` input predicates coming from the initial state ensure that the incoming and outgoing dialogs are assigned to the correct dialog variables. See Section 8 for more details on these predicates.

CW is transparent unless and until its subscriber is connected to a desired party, and a new call comes for the subscriber. The new dialog is assigned to the variable `wait`, and receives a `Ringing` signal from

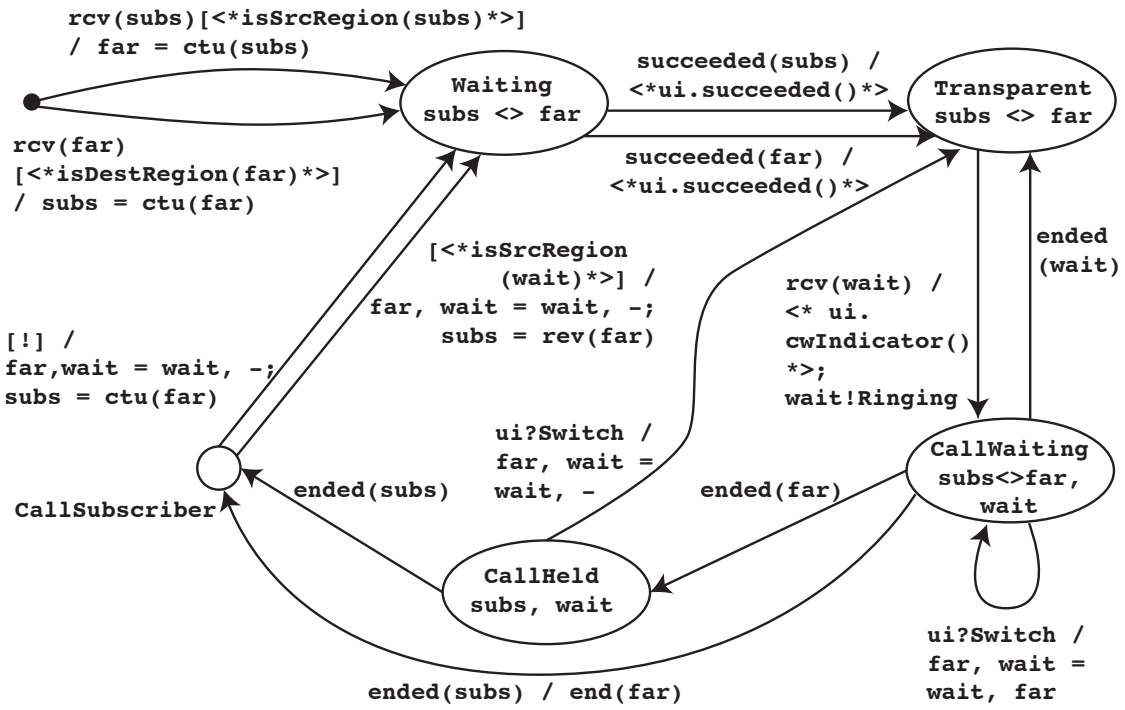

Figure 9: StratoSIP program for the Call Waiting (CW) application, graphical form. The small circle is a transient state.

CW (see Section 5.1). At the same time, CW signals to the user interface that a call is waiting.

If and when the subscriber commands CW to switch calls, CW puts the current far party on hold and connects the subscriber to the waiting call. The switch is programmed simply by swapping the values of `far` and `wait`, and returning to the `CallWaiting` state. The implementation of StratoSIP does all the rest.

CW enters the transient `CallSubscriber` state when its subscriber has disconnected and there is still a call waiting. In this case CW must call the subscriber back. You might think that the waiting dialog must have the subscriber as its destination, but this is not necessarily so. The waiting dialog may have been an original dialog in `far` with the subscriber as its source; this dialog may have then been assigned to `wait` when the subscriber answered another call. Thus the state predicates in the `CallSubscriber` state check the direction of `wait` before deciding whether to continue or reverse it to reach the subscriber.

5 Dialog signaling

5.1 Explicit handling of status messages

As Section 4 has shown, StratoSIP conceals from the programmer most aspects of the SIP messages that create and destroy dialogs and control media. There are other SIP messages that a StratoSIP programmer can handle directly, because their use is more application-specific. In StratoSIP these messages go by the general name of *status messages*.

The current types of status message are:

```
Ringing
Forwarded
Queued
Info
InfoSuccess
InfoFailure
InfoResponse
```

The first three are used to indicate progress in connecting a caller to a desired party, and are called *provisional responses* in SIP. For example, the CW program in Figure 9 illustrates the use of `Ringing` to signal progress to the caller of a waiting call. On

receiving a `Ringing` message,[2] a SIP device will generate a ringback tone for the caller to hear.

An `Info` message is a general-purpose *request* that can be given application-specific content by means of a message modifier. For example, the QT program in Figure 4 assumes that the VoiceXML server uses an `Info` request to convey the result of the VoiceXML script execution. The StratoSIP program is required to send a *response* to the request, the two possible responses being `InfoSuccess` and `InfoFailure`.

A status message can be received from a dialog and handled explicitly by means of an input predicate of the form *dialogVar* `?` *messageType*, with an optional message predicate. For example, in the QT program transitions with `r?Info` input predicates use message predicates to check the content fields of the message, and have different results based on them.

The message type `InfoResponse` can be used in input predicates only. It matches either `InfoSuccess` or `InfoFailure`.

There are two basic ways to create and send a status message:

dialogVar `!` *messageType*
dialogVar `!` *messageVar*

In both cases the message is sent through the dialog identified by *dialogVar*. In the first case, a plain, generic message of the named type is created. In the second case, the message is created as a copy of the status message stored in *messageVar*; it has the type of the stored message and as many of the stored headers as possible. As a side-effect of either action, the sent message is stored in the built-in variable `msgOut` of type Message.

The general forms of message expressions are:

messageType`(`*messageVar2*`)` `[`*messageModifier*`]`

messageVar1`(`*messageVar2*`)` `[`*messageModifier*`]`

where either the argument or message modifier is syntactically optional. The message modifier, if present, can modify the created message before it is sent and stored.

The argument *messageVar2* is required only for creating messages of types `InfoSuccess` and `InfoFailure`. Its value must be the message to which the created message is a response. So if the second form of message expression is used and both message variables are present, *messageVar1* holds a message of type `InfoSuccess` or `InfoFailure`, and *messageVar2* holds a message of type `Info`.

The QT application shows how to receive and respond to an `Info` message. To send an `Info` message, when the response to the message is important, use the following action sequence:

```
. . . / dlg!Info[<*myMsgModifier*>];
        <* infoSent = msgOut; *>
```

Note that assignments to variables of type Message, such as `infoSent`, must be written as embedded Java code. Then the response can be handled explicitly using a transition with the following guard:

```
dlg?InfoResponse
   [<*msgIn.answers(infoSent)*>] / . . .
```

The boolean Java method `answers` returns true if and only if `msgIn` is a response to the particular `Info` message in `infoSent`.

For SIP experts:

Although the message types `Ringing, Forwarded`, and `Queued` correspond to the SIP provisional responses `180, 181`, and `182`, they are not always implemented by the corresponding SIP messages. In SIP these messages can be sent only by the callee end of a dialog, and only before the dialog is confirmed. Many applications interact in ways that make these rules overly restrictive [9].

When they cannot be implemented with SIP provisional responses, StratoSIP implements them with SIP *Info* messages. StratoSIP handles all conversions and responses automatically. On the other hand, if a StratoSIP program sends `Ringing, Forwarded`, or `Queued` to a SIP device, and if it is legal in the current dialog state to send a provisional response, then the StratoSIP implementation will send a provisional response.

For further information about application interactions related to audio tones such as ringback, and how they can best be managed, see [9].

5.2 Status linkages

This section describes implicit handling of received status messages. If a received status message makes the guard of an out-transition from the current state true, then that explicit transition handles the message. Otherwise, the message is handled implicitly.

We first consider provisional responses and `Info` requests, which are handled implicitly by *status linkages*. There are two forms of status linkage. By analogy with media linkages, a *status-flowlink* relates two dialogs. If two dialogs are status-flowlinked, then a request or provisional response received from one is immediately forwarded to the other.

[2] This only works if the device is in the correct state, see notes for experts below.

By analogy with media linkages, if a dialog is not status-flowlinked, then it is status-held. If a dialog is status-held, then a request or provisional response received from it is stored in a status queue associated with the dialog.

When two dialogs become status-flowlinked, if either has a nonempty status queue because it has been status-held, then the contents of that queue are immediately sent to the other dialog. If this is not the desired application behavior, then the queue can be emptied under program control, by means of the action `clearQueue(`*dialogVar*`)`. This clears the current contents of the dialog's status queue.

In most states of StratoSIP programs, the configuration of status linkages is exactly the same as the configuration of media linkages. This is the default for status linkages, and there is no need for any additional annotation.

In some situations, however, the programmer may need to override this default. In these cases the status linkages are listed separately following the media linkages. They use the same syntax as media linkages (except that tilde between `< >` is not used), and are separated from media linkages by a slash. If there are status linkages, every variable pointing to an active dialog must appear in them.

For example, in the prompting state of the QT program, dialogs `i` and `r` are media-flowlinked but not status-flowlinked. The media connection is necessary for voice-based interaction between the caller and the VoiceXML server. However, these two dialogs have no relationship that would justify forwarding status messages from one to the other. In the QT program, `Info` requests are handled explicitly because they are the means of communication between the server and the program. If the server happens to send some other kind of status message, it will be absorbed by QT rather than being forwarded to the caller.

Finally, we consider implicit handling of responses `InfoSuccess` and `InfoFailure`. This handling is independent of status linkages. If the received message responds to a request generated by this box, then it is dropped. If the received message responds to a request received and forwarded by this box, then the response is sent to the dialog from which the request was received, if it is still active, and dropped otherwise. If a dialog is ending with a request pending, then StratoSIP automatically generates a response to clean up the transaction before the dialog ends.

6 Timers

As with dialogs, each distinct active timer is the value of a variable. Each variable of type `Timer` must be declared.

A timer is set with an action *timerVar*`!TimerSet`. A `TimerSet` action must have one named argument with a Java integer value. The argument name is `sec` or `msec`, depending on whether the desired time unit is seconds or milliseconds. After this action, the timer pointed to by *timerVar* is *active*.

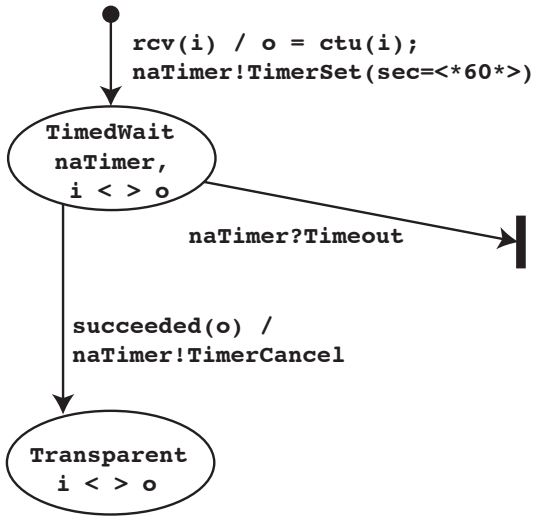

Figure 11: StratoSIP program for the No-Answer Timeout (NATO) application.

A timer becomes inactive in one of two ways. It can generate a timeout, which is received with an input predicate of the form *timerVar*`?Timeout`. Or it can be canceled by the action *timerVar*`!TimerCancel`. An inactive timer can be reset and reused. Note that the `Timeout` input predicate is exceptional in not assigning a new value to `msgIn`.

An active timer is similar to an active dialog in that any stable state with an active timer must be annotated with the timer variable. This is illustrated by Figure 11, which is a program for a No-Answer Timeout (NATO) application. If the outgoing dialog `o` does not succeed within 60 seconds, this application will cause it to fail so that other applications can take over.

NATO is useful as an auxiliary to other applications. For example, it could be placed after RVM, QT, and other applications as in Figure 12. This would have two benefits: (1) Even though each application could have its own no-answer timeout function built in, NATO can supply that function once for all

of them. (2) If each other application had its own no-answer timeout function built in, the times might be different, causing unexpected interactions among the applications. If there is one centralized function, then behavior is easily predictable. After a timeout forces failure, the failure propagates leftward. First the application just to the left of NATO has a chance to handle failure of its outgoing dialog. Eventually it either generates success or propagates failure to the application on its left, and so forth up the chain [10].

If a state is annotated with an active timer `tmr`, then it must have an out-transition guarded by `tmr?Timeout`. Timer and dialog variables can be placed in the annotation in any order.

7 Non-SIP interfaces

As with dialogs and timers, each distinct non-SIP interface is the value of a variable. Each variable of type `NonSipInterface` must be declared. Non-SIP interfaces differ from dialogs and timers in that they have no pre-defined *active* state and therefore do not appear in the annotations of stable StratoSIP states.

A "message" received through a non-SIP interface can be of any Java object type. In the syntax, the object type is used instead of the usual message type. For example, the AT application (Figure 6) uses a non-SIP interface `ui` to communicate with a Web-based user interface. The command types Consult, Resume, and Transfer are object types, and a command is received by means of an input predicate such as `ui?Resume`. Although the built-in variable `msgIn` usually has type Message, after this input predicate has been made true, it will have type Resume.

Non-SIP interfaces are input-only—they can be used to receive messages, but not to send them. On the output side, an application can use the E4SS convergence API ([3], Chapter 9) to interact with the Java environment by means of a `SipToJava` interface. If this API is used, then a NonSipInterface object will support method calls declared in that interface. For example, the AT application invokes methods of the `ui` object in the following Java actions:

```
<* ui.talking(c.getSrc()) *>
<* ui.abandoned() *>
```

where the method `talking` has as its argument the source URI of the dialog from the customer.

8 Java object classes

In this section we document some of the most important programmer interfaces to the Java object classes URI, Dialog, Message, and MessageModifier. However, the most complete and up-to-date version of this information can be found in the Javadoc files.

URI object class:

Each box has a built-in variable `subscriber` of type URI. This variable holds the URI of the subscriber on behalf of whom the box was instantiated. It is initialized when the box is created.

Note that SIP addresses are URIs, but some address fields in SIP messages (such as the *From* field of an initial Invite message) may also contain optional display names. So it is necessary to exercise caution in equality checks between SIP address fields and URIs.

Dialog object class:

Methods of the Dialog class make it possible to access the history of the dialog. Most of the history of the dialog comes from the initial invite message, including its source address, destination address, and the region in which it was routed to or from the box (see Section 2.2). This interface includes the methods used in this manual:

```
public class Dialog {
  URI getSrc();
  URI getDest();
  SipServletRoutingRegion getRegion();
  StratoSIPMessage getInitialRequest();
  StratoSIPMessage getFinalResponse();
}
```

Note that the StratoSIP compiler translates the native type "Message" to the Java type "StratoSIPMessage." `getInitialRequest` returns the whole initial invite message, so the programmer can extract additional information from it (for example, the display name in the *From* field). `getFinalResponse` returns the message that is the final response to the initial invite.

The nature and use of the type SipServletRoutingRegion can be seen from the following excerpts from the CW application (Figure 10):

```
private boolean isSrcRegion(Dialog dialog)
{  return dialog.getRegion() ==
       SipApplicationRoutingRegion.
          ORIGINATING_REGION;
}
private boolean isDestRegion(Dialog dialog)
{  return dialog.getRegion() ==
       SipApplicationRoutingRegion.
          TERMINATING_REGION;
}
```

These methods are used by the application to determine whether a dialog was routed in the source or destination region.

Message object class:

This method of the class is applied to find out if a message is a response to a particular sent message, as explained in Section 5.1:

```
public class StratoSIPMessage
  extends SipServletMessage {
  boolean answers(SipServletRequest req);
}
```

MessageModifier object class:

StratoSIP is designed to hide most aspects of SIP signaling from the programmer. This can include both what SIP message is sent to carry out a desired action, and when it is sent.

However, sometimes a programmer needs to manipulate some part of the content of a SIP message sent by the program. To achieve both goals, an object in type MessageModifier has a distinguished method `modify` that takes a SIP message as an argument and modifies it. If a message modifier is attached to a StratoSIP action such as `end`, whenever the StratoSIP runtime environment creates an outgoing SIP message to satisfy that action, the environment will apply the `modify` method of the message modifier before it sends the message.

Because a `modify` method is general Java code, it can have side-effects such as saving the message in another place.

```
public class MessageModifier {
  void modify(SipServletMessage m);
}
```

9 Application composition

In Section 2.2, Figure 1 shows two applications, one fulfilling all the requirements of caller A, and one fulfilling all the requirements of callee B.

The SIP Servlet standard also provides for *application composition*, which makes it possible to implement a complex application in multiple, independent modules. These modules are automatically composed at runtime by the application router in a SIP servlet container. For example, in Figure 12 the application of callee B has been decomposed into two applications, Redirect to Voice Mail (RVM) and Quiet Time (QT). The figure shows how these two applications are composed at runtime.

Application composition is an optional capability of the SIP Servlet standard. StratoSIP programs run equally well with or without application composition, so the StratoSIP programmer can choose freely whether or not to use it. The remainder of this section provides information about how to use application composition, when it is desired.

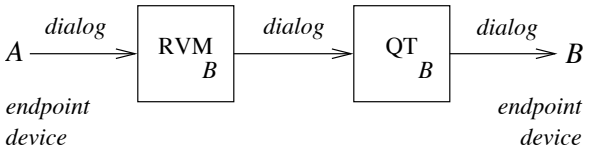

Figure 12: The runtime composition of two StratoSIP programs.

Application composition in the SIP Servlet standard is based on Distributed Feature Composition (DFC) [5]. This section documents the behavior of the DFC application router, which is powerful and general-purpose, and is available as part of the E4SS environment. Note, however, that the standard allows the use of any application router, so other routers might compose applications in somewhat different ways.

Zones:

The *source zone* of a subscriber URI is a sequence of applications. These are the applications that the subscriber wishes to use in calls for which the subscriber is playing the source role. The order of the sequence is determined by *precedence constraints*. In the same way, a subscriber has a *destination zone* of applications to be used when the subscriber is playing the destination role.

The `new` function:

If an initial invite message is created with the `new` function, it is routed to a box of the first application in its source field's source zone, if any. Otherwise it is routed to a box of the first application in its destination field's destination zone, if any. Otherwise it is routed to the destination.

The `ctu` function, no optional arguments:

To explain the effect of `ctu`, we first assume that the function is given no optional `src` or `dest` arguments.

If the seed dialog of a `ctu` was routed to an application in a subscriber's source zone (which is the application executing the `ctu`), then both the seed invite and the created invite have the same source field. The initial invite message created by the `ctu` is routed to the next application in the subscriber's source zone, if any. Otherwise it is routed to a box of the first application in its destination field's destination zone, if any. Otherwise it is routed to the destination.

Similarly, an invite created by a `ctu` in a subscriber's destination zone is routed to the next application in the subscriber's destination zone, if any.

Otherwise it is routed to the destination.

The `ctu` function, optional arguments, and regions:

If an application in a source zone uses a `ctu` with an optional `dest` argument, this has no immediate effect on routing (eventually it will affect which subscriber's destination zone is used). If an application in a destination zone uses a `ctu` with an optional `src` argument, this has no effect on routing.

If an application in a source zone uses a `ctu` with an optional `src` argument, then the created invite will be routed to the first application in the new source's source zone, if any. Otherwise it is routed to a destination zone or destination as above. One effect will be to truncate the source zone of the original source, unless this application is the last one in the zone. Another effect will be to create a *source region* consisting of multiple source zones, unless the new source does not subscribe to any source applications.

Similarly, if an application in a destination zone uses a `ctu` with an optional `dest` argument, then the created invite will be routed to the first application in the new destinations's destination zone, if any. It may truncate the original destination's zone, and it may create a destination region with multiple destination zones.

Free and bound boxes:

If an invite is routed to a *free* application, then the box is a new instance of its application.

At any one time, there can be at most one instance of a *bound* application per subscriber. If an invite is routed to a *bound* application, and the subscriber already has a box that is an instance of this application, the invite is routed to this existing box.

Note that bound applications such as CW often appear in both a subscriber's source and destination zones. The same box is routed to in both zones.

The `rev` function:

The `rev` function can only be used by an application that appears in *both* its subscriber's source zone and its subscriber's destination zone. The seed dialog was routed to the application in one of these zones.

The `rev` function swaps the source and destination fields from the seed. It also inverts the zone (source or destination) in which routing is taking place. It also over-rides the source and destination fields with optional arguments, if any. After these transformations, the created invite is routed as if it were created by `ctu`.

As an example, say that the seed argument to a `rev` was routed to its application in the source zone, and that there are no optional arguments. Then the created invite's destination is the same as the seed invite's source. The created invite will be routed to the next application in its destination's destination zone. Thus the reason that a `rev` can only be used by an application subscribed to in both zones is that it must have a well-defined place in the application ordering of each zone.

Examples:

For concise, well-focused examples of DFC routing, see [6]. For insight into how DFC routing organizes and composes a large set of applications, see [11].

References

[1] Eric Cheung and Pamela Zave. Generalized third-party call control in SIP networks. In *Proceedings of the 2nd International Conference on Principles, Systems and Applications of IP Telecommunications*, pages 45–68. Springer-Verlag LNCS 5310, 2008.

[2] S. Donovan. The sip info method. IETF Network Working Group Request for Comments 2976, 2000.

[3] ECharts for SIP Servlets (E4SS) manual. `http://echarts.org/ECharts-for-SIP-Servlets-Manual.html`, 2009.

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

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

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

[7] J. Rosenberg. A hitchhiker's guide to the Session Initiation Protocol (SIP). IETF Network Working Group Request for Comments 5411, 2009.

[8] 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.

[9] Pamela Zave. Audio feature interactions in voice-over-IP. In *Proceedings of the 1st International Conference on Principles, Systems and Applications of IP Telecommunications*, pages 67–78. ACM SIGCOMM, 2007.

[10] Pamela Zave. Modularity in Distributed Feature Composition. In Bashar Nuseibeh and Pamela Zave, editors, *Software Requirements and Design: The Work of Michael Jackson*, pages 267–290. Good Friends Publishing, 2010.

[11] Pamela Zave. Mid-call, multi-party, and multi-device telecommunication features and their interactions. In *Proceedings of the 5th International Conference on Principles, Systems and Applications of IP Telecommunications*. ACM Digital Library, 2011.

[12] Pamela Zave, Gregory W. Bond, Eric Cheung, and Thomas M. Smith. Abstractions for programming SIP back-to-back user agents. In *Proceedings of the 3rd International Conference on Principles, Systems and Applications of IP Telecommunications*. ACM SIGCOMM, 2009.

Appendix: Grammar in EBNF

```
// NAME SPACES

boxName : Identifier ;       // capitalized
stateName : Identifier ;     // capitalized
varName : Identifier ;   // not capitalized
fieldName : Identifier ; // not capitalized
// Comments on name spaces are lexical
// conventions only.
// Reserved variable names are "msgIn" and
// "msgOut" of type Message, "region" of
// type Region, and "subscriber" of type
// URI.  These variables are built-in and
// given their values automatically.

// BEGINNING A PROGRAM

prog : packageDef? imports? boxType boxName
       boxParams?  inheritance? section+
     ;

packageDef : 'package' qualifiedName ';' ;

qualifiedName :
           Identifier ( '.' Identifier )* ;

imports: hostCode ;

hostCode : '<*' JavaCode '*>' ;
// Host code can refer to Dialog and
// Message variables, but not to Timer or
// NonSipInterface variables.

boxType : 'free' 'box' | 'bound' 'box' ;

boxParams : '(' hostCode ')' ;

inheritance : hostCode ;

// PROGRAM SECTIONS

section : declarations
        | initialization
        | graph
        ;

declarations :
            'declarations' '{' decl+ '}' ;

decl
  : 'Dialog'
          dialogVar ( ',' dialogVar )* ';'
  | 'Timer' timerVar ( ',' timerVar )* ';'
  | 'NonSipInterface'
                ( ',' interfaceVar )* ';'
  | 'Message'
        messageVar ( ',' messageVar )* ';'
  | hostCode ';'
  ;

dialogVar : varName ;

timerVar : varName ;

interfaceVar : varName ;

messageVar : varName ;

initialization :
        'initialization' '{' initAct+ '}' ;

initAct : hostCode ';' ;

graph : 'graph' '{' graphObject+ '}' ;

graphObject : state | transition ;

// GRAPH STATES

state : 'initial' 'state' stateName
                            ('{' '}')? ';'
      | 'stable' 'state' stateName
                '{' stateLinkages '}' ';'
      | 'terminal' 'state' stateName
                            ('{' '}')? ';'
      | 'transient' 'state' stateName
```

```
                              ('{' '}')? ';'
      ;

stateLinkages :
 (     mediaLinkage (',' mediaLinkage )* )?
 ('/' statusLinkage (',' statusLinkage)* )?
;

mediaLinkage
  : v1=dialogVar mediaLinkType v2=dialogVar
  | dialogVar
  | timerVar
  ;

mediaLinkType : '<' '>' | '<~>' ;

statusLinkage
  : v1=dialogVar '<' '>' v2=dialogVar
  | dialogVar
  ;

// GRAPH TRANSITIONS

transition :
     'transition' s1=stateName '->' arcBody
                       '->' s2=stateName ';'
;

arcBody : responsiveArcBody
        | transientArcBody
        ;

responsiveArcBody :
    responsiveGuard ( '/' actionSeq )? ;

responsiveGuard :
  inputPredicate
            ( '[' messagePredicate ']' )? ;

actionSeq : action ( ';' action )* (';')? ;

transientArcBody :
  transientGuard ( '/' actionSeq )? ;

transientGuard : '[' statePredicate ']'
               | '[' '!' ']'
               ;

// INPUT PREDICATES

inputPredicate
  : 'rcv' '(' argList ')'
  | 'ended' '(' argList ')'
  | timerVar '?' 'Timeout'
  | interfaceVar '?' interfaceType
  | dialogVar '?' messageType
  | 'succeeded' '(' argList ')'
  | 'preSucceeded' '(' argList ')'
  ;

messageType
  : 'Ringing'
  | 'Forwarded'
  | 'Queued'
  | 'Info' | 'InfoResponse'
  | 'InfoSuccess' | 'InfoFailure'
  ;

timerType
  : 'TimerSet' | 'TimerCancel' | 'Timeout';

interfaceType : Identifier ;

// ACTIONS

action
  : 'end' '(' argList ')'
              ( '[' messageModifier ']' )?
  | 'clearQueue' '(' argList ')'
  | timerVar '!' 'TimerSet' '(' argList ')'
  | timerVar '!' 'TimerCancel'
  | interfaceVar '!' hostCode
  | dialogVar '!' messageExp
  | assignment
  | hostCode
  ;

messageModifier : hostCode ;

messageExp
  : messageType ( '(' argList ')' )?
                ( '[' messageModifier ']' )?
  | messageVar ( '(' argList ')' )?
                ( '[' messageModifier ']' )?
  ;

assignment
  : dialogVar ( ',' dialogVar )* '='
             dialogExp ( ',' dialogExp )* ;

dialogExp
  : ( 'new' | 'ctu' | 'rev' )
           '(' arglist ')'
                 ('[' messageModifier ']')?
  | dialogVar
  | '-'
  ;
```

```
// PREDICATES

statePredicate
  : 'succeeded' '(' argList ')'
  | 'preSucceeded' '(' argList ')'
  | hostCode
  ;

messagePredicate : hostCode ;

// ARGUMENT LISTS, IN GENERAL

argList : requiredArgs ',' optionalArgs
        |  requiredArgs
        |  optionalArgs
        ;

requiredArgs :
      positionalArg (',' positionalArg )* ;

optionalArgs : namedArg (',' namedArg )* ;

positionalArg : valExp | namedArg ;
    // For definition of valExp, see below.

namedArg : fieldName '=' valExp ;
             // For definition of fieldName
             // and valExp, see below.

// SPECIFIC ARGUMENT LISTS AND VALUE
// EXPRESSIONS
// This is in the form of a table rather
// than a grammar.
-----------------------------------------
         |  fieldName   |  valExp
-----------------------------------------
INPUT PREDICATES
'rcv'
   required
          'dialog'        dialogVar
'ended'
   required
          'dialog'        dialogVar
'succeeded'
   required
          'dialog'        dialogVar
'preSucceeded'
   required
          'dialog'        dialogVar
-----------------------------------------
ACTIONS
'end'
   required
          'dialog'        dialogVar
'clearQueue'
   required
          'dialog'        dialogVar
-----------------------------------------
CONSTRUCTORS USED IN ACTIONS
timerType 'TimerSet'
   required one of
          'sec'           hostCode(int)
          'msec'          hostCode(int)
interfaceType
   see individual interface types
messageType 'Ringing'
   no arguments
messageType 'Forwarded'
   no arguments
messageType 'Queued'
   no arguments
messageType 'Info'
   no arguments
messageType 'InfoSuccess'
   required
          'request'       messageVar
messageType 'InfoFailure'
   required
          'request'       messageVar
dialogExp 'new'
   required
          'dest'          hostCode(URI)
   optional
          'src'           hostCode(URI)
          'fastResponse'  hostCode(boolean)
dialogExp 'ctu'
   required
          'seed'          dialogVar
   optional
          'src'           hostCode(URI)
          'dest'          hostCode(URI)
          'fastResponse'  hostCode(boolean)
dialogExp 'rev'
   required
          'seed'          dialogVar
   optional
          'src'           hostCode(URI)
          'dest'          hostCode(URI)
          'fastResponse'  hostCode(boolean)
-----------------------------------------
STATE PREDICATES
'succeeded'
   required
          'dialog'        dialogVar
'preSucceeded'
   required
          'dialog'        dialogVar
-----------------------------------------
```

```
bound Box CallWaiting

declarations {
   Dialog subs, far, wait;
   NonSipInterface ui;
<* private boolean isSrcRegion(Dialog dialog) {
      return dialog.getRegion() == SipApplicationRoutingRegion.ORIGINATING_REGION;
   }
   private boolean isDestRegion(Dialog dialog) {
      return dialog.getRegion() == SipApplicationRoutingRegion.TERMINATING_REGION;
   }
*> }

graph {

initial state Init;
   transition Init -> rcv(subs)[<* isSrcRegion(subs) *>] /
      far = ctu(subs) -> Waiting;
   transition Init -> rcv(far)[<* isDestRegion(far) *>] /
      subs = ctu(far) -> Waiting;

stable state Waiting { subs < > far };
   transition Waiting -> succeeded(subs) / <* ui.succeeded() *> -> Transparent;
   transition Waiting -> succeeded(far) / <* ui.succeeded() *> -> Transparent;

stable state Transparent { subs < > far };
   transition Transparent -> rcv(wait) /
      <* ui.cwIndicator() *>; wait!Ringing -> CallWaiting;

stable state CallWaiting { subs < > far, wait };
   transition CallWaiting -> ended(wait) -> Transparent;
   transition CallWaiting -> ended(far) -> CallHeld;
   transition CallWaiting -> ended(subs) / end(far) -> CallSubscriber;
   transition CallWaiting -> ui?Switch /
      far, wait = wait, far -> CallWaiting;

stable state CallHeld { subs, wait };
   transition CallHeld -> ui?Switch / far, wait = wait, - -> Transparent;
   transition CallHeld -> ended(subs) -> CallSubscriber;

transient state CallSubscriber;
   transition CallSubscriber -> [<* isSrcRegion(wait) *>] /
      far, wait = wait, -; subs = rev(far) -> Waiting;
   transition CallSubscriber -> [!] /
      far, wait = wait, -; subs = ctu(far) -> Waiting;
}
```

Figure 10: StratoSIP program for the Call Waiting (CW) application, textual form.