ECharts for SIP Servlets: a state-machine programming environment for VoIP applications

Thomas M. Smith
AT&T Labs Research
180 Park Avenue
Florham Park, NJ 07932
USA
tsmith@research.att.com

Gregory W. Bond
AT&T Labs Research
180 Park Avenue
Florham Park, NJ 07932
USA
bond@research.att.com

ABSTRACT

The development of telecommunication applications that require multiple call legs is often complex due to their event-driven nature as well as the significant amount of state that must be maintained. In general, the state associated with different call legs within an application instance differs and must be separately maintained; in addition, (non-call related) application state is often required. As a means of managing all the required state information, application developers often implement ad-hoc state machine programming constructs within the application.

This paper describes an alternate approach, wherein the application logic is written in ECharts, an open-source state-machine programming language, and translated into SIP Servlet applications which can be deployed and executed on any standards-compliant container. This approach can yield great benefits for complex applications, including reusability, increased maintainability, and the prospect of program analysis.

Keywords

Telecommunications, VoIP applications, SIP servlets, ECharts, UML Statecharts, state machine programming

1. INTRODUCTION

Telecommunication applications vary greatly in complexity, from simple routing applications to sophisticated multimedia applications that persist over time, involving multiple parties and media servers. To accommodate such varying requirements, a range of application development environments has been developed. In the Voice over IP realm, these include CPL, LESS, SPL, Camel, ParlayX, and JAIN-SLEE. For services based on SIP, the SIP Servlet standard (codified by the Java Community Process in JSR116 [10]) has gained considerable industry support since its introduction, being supported by at least four commercial implementations at this time.

Appears in IPTComm 07, Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications, pages 89-98, New York, NY, USA, 2007. ACM. Copyright ©2007 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or permissions@acm.org.

The SIP Servlet standard is an extension of the servlet model of programming that has gained great popularity in the world of web application development. This familiar model simplifies the process of learning to develop SIP-based applications without sacrificing application generality. Contrasted with application development environments such as CPL [12], LESS [16], and SPL [5], which are designed for simplicity and safety, the SIP Servlet standard allows nearly unfettered use of SIP, making it a powerful tool for creating complex applications.

SIP Servlet applications implement standard SIP Servlet interface methods like `doInvite` and `doSuccessResponse`, which are invoked by the servlet container upon receipt of corresponding SIP messages. Simple proxy applications (e.g., routing applications) can be created very easily by creating a `doInvite` method that manipulates the received message as desired before instructing the servlet container to send the message on towards its destination.

For more complex applications involving mid-call switching, call termination, or third-party call control, applications that implement SIP back-to-back user agents (B2BUAs) are often required. Such applications must maintain the SIP dialog state for multiple calls, and react to any legal SIP message that may be received on any of these calls. This greatly complicates the application development task. It is common for application developers to create Java state-machine constructs to maintain the state, check this state within the `doXXX` methods, and conditionally perform actions and change state in these methods. This can result in application logic that is distributed throughout the servlet, making it difficult to develop and maintain.

To get the benefits of a SIP Servlet application environment while simplifying the process of developing complex applications, we have harnessed the power of ECharts, an open-source state-based programming language [3]. We have created an adaptation layer that allows a developer to create an application in the ECharts language and automatically translate it into Java code that executes in a SIP Servlet container. Since ECharts runs within a host language (Java in this case), the full scope of the SIP Servlet API is available to the application developer, while the task of maintaining state and reacting to events is declaratively expressed in the ECharts language. This work, comprising approximately 3000 lines of code, has been released as open-source

software under the Common Public License. This paper describes our framework, called ECharts for SIP Servlets, as well as the attendant benefits of using the framework.

2. THE ECHARTS PROGRAMMING LANGUAGE

State machines are commonly used in the telecommunications application domain, since they provide a formal way of representing responses to asynchronous inputs using limited historical information; they can be implemented efficiently; and static analysis and testing techniques for state machines are well developed.

ECharts is a state machine-based programming language for event-driven systems derived from the standardized UML Statecharts language [13]. ECharts began its life in 1999 at AT&T Labs Research to support Voice Over IP service development. Since that time the language has been used to implement a number of production systems such as the consumer-oriented Voice over IP product that is currently marketed in the U.S. as AT&T CallVantage. The language has continued to evolve in response to developers' needs. In 2006, ECharts was released by AT&T as open-source software under the Common Public License.

2.1 Other state machine languages

Prior to developing ECharts we looked for an existing language that met our service-development needs. In addition to supporting state machines we required support for code reuse, scalability to large numbers of concurrently executing state machines, and support for Java, the language used to implement the surrounding service execution infrastructure. To this end we evaluated two existing standardized state machine-based languages: UML Statecharts [13] and SDL [1].

2.1.1 UML Statecharts

UML Statecharts is a standardized state machine design language that is part of OMG's Unified Modeling Language (UML) standard. The benefits of UML Statecharts are that the language is visual, expressive, and that it is supported by commercial tools that perform design capture, (Java) code generation and code execution.

With regards to expressiveness, we were attracted by the language's support for hierarchical state machines, concurrent (orthogonal) state machines, and machine synchronization. However, when we attempted to design a few nontrivial services with the language we encountered short-comings:

- The language supports only a single transition priority rule for constraining transition execution order. Since a Statechart model generally incorporates hierarchical and concurrent machines, this can result in many nondeterministic situations.
- The language provides only rudimentary support for re-using models, and no support at all for parameterizing models.
- Broadcast communication is utilized for communications within a Statechart model which, in general, can result in broadcast events being lost in situations when no part of a model is ready to accept an event.
- While it is possible for a fixed number of concurrent state machines to exist statically within a state, it is not possible to create state machines dynamically.
- Finally, we found the machine termination semantics to be overly complex, requiring the developer to explicitly specify terminal states, which we consider to be redundant.

In defense of UML Statecharts, we recognize that most shortcomings exist because the language is intended to serve first as a design language, and second as an implementation language. As a design language, broadcast communications and non-determinism are advantageous since they allow the designer to defer committing to implementation details. However, our need was for a language that would first serve as an implementation language and second as a design language.

2.1.2 SDL

SDL (Specification and Description Language) has a long history in telecommunications systems development. The ITU released its first version of the SDL standard in 1980. The standard has undergone several updates since then with major revisions in 1988 (SDL-88), 1993 (SDL-92) and 2000 (SDL-2000). At the time of our evaluation in 1999 no tools yet supported the SDL-2000 specification. While tool support existed for SDL-92, the SDL-92 language itself was not a full-featured state machine-based language. For example, it lacked support for explicitly specifying hierarchical and concurrent machines. It was for this reason that we decided not to use the SDL-92 language. The SDL-2000 language, on the other hand, incorporates many concepts from UML Statecharts including hierarchical and concurrent machines. It goes further in that it defines a simple textual syntax in addition to a graphical syntax, it supports a limited (static) form of machine parameterization (genericity), it supports exception handling and it has a formally defined semantics [7].

2.1.3 Choosing ECharts

Given the apparent suitability of SDL-2000 for telecommunications service programming one might ask why our research group didn't choose to use it instead of developing a similar new language, ECharts. The reason is simply that we never intended to develop a new language. In 1999, the focus of our research was telecommunication service composition, not service programming languages. In order to expedite our research we were looking for a language with tool support in place that met our requirements. SDL-2000 did not satisfy our requirements because no tools existed for the language in 1999. Furthermore, the prospect of implementing a Java translator and runtime for SDL-2000 was daunting given the complexity of its specification and given the resources we were able to devote at the time. As a result, we quickly developed a rudimentary Java library to provide us with the language support we required. Over the ensuing years, this library evolved into the ECharts language.

It is interesting to note that, while commercial tools for SDL-2000 exist today, we are not aware of the existence of

an SDL-2000 to Java translator or runtime, lending credence to our conclusion in 1999 regarding the relative difficulty of implementing an SDL-2000 translator and runtime.

2.2 ECharts Features

ECharts inherits many features from the UML Statecharts, notably, support for hierarchical and concurrent machines. Like UML Statecharts, an environmental event arriving for a machine triggers the machine to run-to-quiescence, at which point the machine waits for another environmental event. Concurrency within a machine is defined to be interleaved and the execution of a machine transition is atomic. However, to address the short-comings of UML Statecharts we introduced the following implementation-oriented features into ECharts:

- a simple textual syntax;
- simplified or additional control over a machine's behavior, e.g., transition and port priorities, communication within machines and between machines, explicit event consumption, and simplified termination semantics;
- new classes of behavior, e.g., machine arrays;
- enhanced re-usability, i.e., parameterized machines.

The following sections provide a brief overview of some of these features. For a comprehensive overview see [3].

2.2.1 Textual and Graphical Syntax

The ECharts language, like UML Statecharts, possesses both a graphical syntax and a textual syntax. The graphical syntax for the current version of ECharts is similar to that of UML Statecharts. The ECharts textual syntax, however, is significantly different from that of UML Statecharts. UML Statecharts uses a dense XML dialect intended for storage, retrieval and exchange of Statecharts designs; it is not intended for human composition. Instead, Statecharts developers are expected to use a graphical editor for composition.

In contrast, ECharts uses a simple textual syntax intended to support human composition. We chose this approach because it required less initial development: changes in language design were less costly because we did not need to maintain a sophisticated graphical editor for all language features, and we believed that the language would be more likely to be adopted since text is easier for programmers to read and write using their favorite text editors. At any point during development, a graphical depiction of a machine program can be generated using tools included with the ECharts development kit. Naturally, it is possible to develop a front-end graphical editor or a back-end XML dialect translator for the language but, to date, these have not been requested by developers.

An abstract representation of the textual syntax for defining a transition is:

```
transition STATE_A - portInstance ? ObjectClass
    [ guardCode ] / actionCode
-> STATE_B;
```

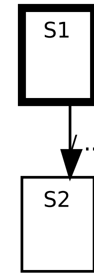

Figure 1: Graphical representation using Graphviz

This transition specifies that a state transition from `STATE_A` to `STATE_B` can occur if an instance of class `ObjectClass` is enqueued at the head of `portInstance`'s input queue, and the `guardCode` evaluates to `true`. If the transition is chosen to fire then `objectInstance` is removed from `portInstance`'s input queue and the `actionCode` is executed immediately prior to changing the machine's state from `STATE_A` to `STATE_B`. As you will see in subsequent examples, it is only necessary to specify a source and target state for a transition; all other components, such as the guard condition, are optional.

Here is an example of the canonical "Hello World!" program using the textual representation.

```
package examples;

public machine Example0001 {
    initial state S1;
    state S2;
    transition S1 - /
        System.out.println("Hello World!")
    -> S2;
}
```

Although a graphical editor has not been developed for ECharts, we have developed a tool that translates textual ECharts programs to their graphical representation using the Graphviz graph layout tools [6] developed at AT&T (available as open-source). As an example, Figure 1 depicts the graphical representation produced by our translator of the example program given above.

2.2.2 Parameterization

Any attempt to define languages usable for implementation must provide higher-level mechanisms for reuse. We choose to model parameterization of ECharts using a semantics roughly based on procedures and procedure calls. See Section 4.1 for an example of this in the context of ECharts for SIP Servlets.

2.2.3 Port Abstraction

In ECharts, messages are exchanged between a state machine and its environment via ports. A port enqueues messages arriving from the environment in FIFO order. A port message is dequeued when a transition specifying that port

fires. Any number of ports can be defined for a state machine: the port abstraction provides the designer with the ability to logically partition the environment into different message sources. See Section 3.2 for a description of how ECharts ports are used in the context of ECharts for SIP Servlets.

2.2.4 Transition Priority Rules

Parameterized state machines and the port abstraction introduce the possibility of building a machine to implement a uniform behavior that is needed in distinct contexts. However, many programs can be implemented by small changes on top of a general solution. We incorporate three transition priority rules in ECharts to allow developers to both specify and override default behaviors, as well as reduce non-determinism. While space does not permit a complete description of all the rules, Sections 4.2 and 4.5 provide examples of two of the rules in the context of ECharts for SIP Servlets. See [4] for a formal description of the transition rule semantics.

2.2.5 Machine Arrays

In real-time programming, it is often the case that only an upper bound is known on the number of incoming events that will require processing. In such a situation it is often preferable to dynamically commit the resources required to respond to the event when it occurs, rather than committing the resources ahead of time based on a worst-case upper bound. To address this need, ECharts supports the declaration of machine arrays and the dynamic creation of machine elements within those arrays. This language feature is used to support the implementation of the SIP Back-To-Back User Agent (B2BUA) machine (see Section 4.1).

3. FRAMEWORK ABSTRACTIONS

ECharts for SIP Servlets makes heavy use of ECharts abstractions and introduces some of its own. This section describes the central concepts of the framework, as illustrated in Figure 2. The framework defines a SIP Servlet subclass which is responsible for creating `FeatureBox` instances. A `FeatureBox` instance contains a specific ECharts `Machine` to execute the desired application logic. These concepts are described in detail below.

3.1 Machines

Each application specifies a particular ECharts `Machine` as its top-level state machine. An instance of this machine will be created by the framework in response to a received initial request, and the request will be submitted to the machine to advance its state logic. Since ECharts supports nested machines, the top-level machine may create an arbitrary number of sub-machines to implement the desired logic. This feature of ECharts is extremely powerful, as it allows the developer to re-use machines that are known to correctly implement certain common behaviors, such as setup, terminate, or switch a call. These machines, all written in ECharts and encapsulated in `.ech` files, provide the structure and logic for the entire application. Auxiliary Java classes may be created and referenced from these machines as desired by the application developer.

3.2 Ports

Central to all ECharts message processing is the concept of *ports* that provide a mechanism for ECharts machines to interact with the environment. Our framework provides two port subclasses that are used to interact with the application environment.

`SipPorts` are used for the sending and receiving of SIP messages. When the container presents a received message to an application, it will be represented as a message on a named `SipPort` that the developer has defined. Furthermore, any SIP messages created by the application are sent on a (named) `SipPort`. The framework validates that the `SipSession` of the message matches that defined for the port. The use of named ports to send messages (e.g., `caller ! bye`) and process received messages (e.g., `callee ? SuccessResponse`) can increase readability of the code and thus reduce programming errors.

`NonSipPorts` are also defined for environmental events that occur outside of the SIP protocol. Examples include a click on a webpage or an RMI method call. We presently define input events for such ports, but output events are not defined. As an example, consider a conference calling application wherein a user has a web interface to the call. When the user presses a certain element on the web page, call recording should begin. In this case, a machine transition could be defined on a `NonSipPort` named `web` as: `web ? Record`, which would be satisfied when a Java object of class `Record` is received on a `NonSipPort` named `web`. In this way non-SIP events are handled in exactly the same way as SIP events from the point of view of the state machine.

3.3 FeatureBox

ECharts for SIP Servlets adds a new abstraction to those provided by ECharts: that of a `FeatureBox`. This construct is a class containing the ports and machines corresponding to a specific application instance.

The `FeatureBox` shown in Figure 2 contains three `SipPorts` that partition the total set of SIP messages into those corresponding to a *caller*, a *callee*, and a *mediaserver*. In addition, a `NonSipPort` is defined for *control* events that may occur as a result of user interaction with the media server. All messages on these ports are processed in accordance with the behavior defined by the machine contained by the `FeatureBox`.

4. BENEFITS

By casting the task of application development as definition of state machines rather than the implementation of a number of message-based callbacks, our framework allows the application logic to be consolidated into a single logical entity - the state machine. This logical entity may include nested or even dynamically-created submachines by taking advantage of those features of ECharts. This is the key benefit of our framework.

In this section, we describe in detail several more benefits that application developers may derive by using the ECharts for SIP Servlets framework.

4.1 State Machine Re-use

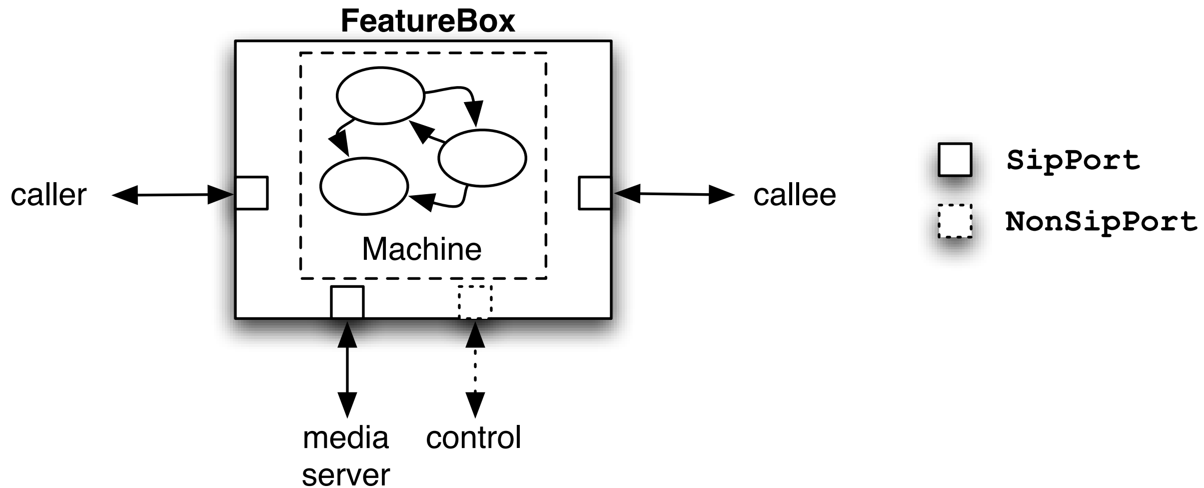

Figure 2: ECharts for SIP Servlets abstractions

Since ECharts supports the concept of nested machines, we can encapsulate common SIP application patterns in reusable, parameterized machines. These machines can be developed once, thoroughly tested and debugged, and then re-used in any number of applications. Common patterns include call setup, call teardown, call switching, and "transparent" behavior, wherein messages received from one party are propagated to another party, after appropriate changes dictated by the SIP protocol.

The open-source distribution includes a non-trivial example machine that implements a back-to-back user agent, which is the basis for many powerful network-based features. This machine (called `B2buaFSM`) accepts an initial INVITE request, and sends a new INVITE request after performing any application-specified modifications to the request. Responses to this new INVITE are relayed to the party that sent the initial INVITE, and subsequent requests and responses are propagated transparently by dynamically creating new machines to handle each new transaction.

An application developer may re-use this machine by simply embedding it within a state of the application's machine, as in:

```
initial state PLACE_CALL:
        B2buaFSM(box, caller, callee, myReqMod);
```

This simple declaration will instantiate a state machine that specifies B2BUA logic and execute its default behavior.

4.2 Overriding Default Behavior

Though the ability to re-use state machines is undeniably powerful, there is a limit to the number and flexibility of applications that can be created by simply reusing the default behavior of existing components. Fortunately the structure of ECharts makes it very easy to override the default behavior of embedded machines. By exploiting the transition priority rules, an application can precisely specify the behavior that should be modified.

For example, if a B2BUA is used to place a call to a media server, the call is expected to succeed or fail quickly. However the media server may send a provisional response (such as `180 Ringing`) before returning a final response. The application developer may not wish to propagate such a message to the caller in order to prevent a brief ringback tone. The default B2BUA behavior is to propagate all messages between the two parties, so the application must specify an exception:

```
/** Do not propagate 180 Ringing.
 *  Just go back to where we were before.
 */
transition PLACE_CALL -
    callee ? ProvisionalResponse180
-> PLACE_CALL.DEEP_HISTORY;
```

To accomplish the same exceptional behavior in a standard SIP Servlet formulation, one of the `doResponse` methods would need to be modified to check the status code of the response, and possibly check which SIP dialog the response corresponds to. As the exceptions become more numerous, they can result in logic distributed throughout the callback methods. In the ECharts code, the message class is obvious from the transition condition, and the ECharts port abstraction makes it clear that this exceptional behavior only applies to responses received from the callee.

4.3 Support for Non-SIP Events

The importance of *converged applications* has been widely noted [10], [11]. In general, this means that applications will depend on multiple protocols. A number of commercial SIP Servlet implementations offer support for application convergence by supporting protocols such as HTTP or SOAP in addition to SIP. Events can be presented to an application that should have equal standing to SIP events.

ECharts for SIP Servlets supports converged applications by defining a `NonSipPort` class, which is a peer of the `SipPort` class in the object hierarchy. Transitions can be defined based on events which are received on such ports and integrated seamlessly into the application logic.

Consider a conference calling application that presents a web interface to the person who is hosting the conference. The interface could display status information about the call (number and identity of participants, speaker identity, time remaining, etc.) In addition, the web interface could present controls, allowing the host to mute participants, start and stop recording, or drop participants from a call. The application could define domain-specific classes to represent these control commands, and integrate the non-SIP control events into the application logic. The example below shows the ECharts syntax specifying that the transition should occur upon receipt of a (domain-specific) Java object of class `DropLeg` on the `webHost` port.

```
transition CONFERENCE_IN_PROGRESS -
    webHost ? DropLeg /
        legToDrop = message.getLegId();
-> DROP_LEG;
```

As there is no standard mechanism to enable converged applications in JSR116, some container-specific code is required to provide a non-SIP interface from the container to `NonSipPorts`. Once this is done, then both SIP and non-SIP events can be treated in a unified way. A Java API is provided to facilitate this task.

4.4 Support for Timed Transitions

Another important category of events is *timer-based events*. These could occur in a voicemail application (redirect a call after 24 seconds of ringing) or a pre-paid calling card application (play a warning five minutes before the card balance expires). Indeed, the SIP Servlet standard mandates a timer service that is available to applications. To use the timer service in a standard SIP Servlet application, the application must follow these steps:

- Retrieve a reference to the `TimerService` from the `ServletContext`
- Create a `ServletTimer` object
- Implement the `TimerListener` interface (callback), providing the required logic for any timeout that may occur anywhere in the application.
- Cancel outstanding timers when no longer required.

The implementation of the required `TimerListener` interface means there is another place that application logic may be specified in a standard SIP Servlet application.

To integrate timer-based events into our state machine logic, we take advantage of the *timed transition* facility within ECharts:

```
/** Ring-no-answer condition
 */
transition RINGING -  delay(rnaTimeout)
-> REDIRECT_TO_VOICEMAIL;

/** Call completed successfully.
 */
transition RINGING - callee ? SuccessResponse /
    caller ! caller.createResponse(message);
-> CONNECTED;
```

In this example, if the call completes successfully before the timed transition fires, then the state machine is advanced to a new state (`CONNECTED`), which automatically cancels the timer created in the `RINGING` state. The machinery required to implement these timed transitions using the `ServletTimer` mechanism is hidden from the application developer, who focuses only on the application logic.

4.5 Message Class Hierarchy

The SIP Servlet standard provides some limited support for the notion of a message hierarchy. Instead of (or in addition to) implementing a generic `doResponse` callback, the application developer may choose to implement more specific callbacks such as `doSuccessResponse`. However this only applies to SIP responses, and has a fixed level of granularity, as the standard mandates which callbacks are defined.

One of the ECharts transition priority rules is based on message class hierarchy. This is based on Java inheritance, unlike the SIP Servlet support for message hierarchy. The rule says that if two transitions are satisfied by an incoming messages, the transition defined on the more specific message class will have priority (all other things being equal). This can be a great convenience, and therefore our framework defines a fairly extensive class hierarchy. The framework includes requests as well as responses, so that, for example, a `ReInvite` class can be defined as a subclass of the `Invite` class. This could further be subclassed into a `SessionTimer` class. We define several specific response classes as well as intermediate message classes (e.g., `Response`, `FinalResponse`, `ErrorResponse`, and `ErrorResponse486`).

Having a full message class hierarchy allows for concise specification of behavior in various cases. For instance, the SIP Servlet standard mandates that applications send ACK messages upon receipt of a success response, but that applications must *not* send ACK for any other final response. Such behavior can be easily specified by taking advantage of the message class hierarchy:

```
transition AWAIT_FINAL_RESPONSE -
    callee ? SuccessResponse /
        callee ! message.createAck()
-> CONNECTED;

transition AWAIT_FINAL_RESPONSE -
    callee ? FinalResponse
-> CLEANUP;
```

Any message of class `SuccessResponse` will satisfy both

transitions. Any other final responses (e.g., error responses or redirect responses) will only satisfy the second transition. Because `SuccessResponse` is a more specific class than `FinalResponse`, such responses will cause the first transition to have priority over the second.

4.6 Automatic Termination Handling

Telecommunications applications have widely divergent purposes, but all such applications must eventually terminate their calls. In SIP, the message used to terminate a call depends on the state of the SIP dialog. Depending on the state, a call might be properly terminated by sending BYE, sending CANCEL, or sending an error response to a received INVITE, and the resulting messages that are expected in response vary in each case. To support this universal need while reducing application complexity, we have used ECharts for SIP Servlets to create a framework to automatically terminate calls under certain conditions.

If an application uses a standard B2BUA pattern (one call in, one call out, and either party may end the call), then the B2BUA software supplied in the open-source distribution will handle termination just like any other mid-call request, by simply relaying a received BYE message to the other party, and then relaying the received response. However more complex applications may involve more than two parties or may need to terminate calls based on an application-specific trigger. To support these cases, we have created a framework for automatic termination handling.

The termination handling framework tracks the SIP dialog state of each `SipPort` in a `FeatureBox`. Updates to the state are made automatically based on SIP messages sent and received by the application. If a trigger condition occurs indicating the need to terminate the application, then a new termination machine is dynamically created for each port, all executing concurrently to tear down all the application's calls according to the SIP protocol. This port termination machine can also be used explicitly in an application to selectively tear down the calls on one or more ports.

There are three conditions which trigger automatic termination behavior:

1. *Application-initiated teardown* - This condition is used when the application wishes to initiate teardown on its own. This could occur, for example, when a prepaid calling card balance goes to zero. This condition is initiated when the application machine transitions to a terminal state, i.e., a simple state with no outgoing transitions.

   ```
   /** end call if timer expires
    */
   transition IN_PROGRESS -
       delay( maxTime )
   -> END;

   /** terminal state
    */
   state END;
   ```

   Tying termination behavior to a terminal state is reasonable, as the application has declared that it has no further actions to take.

2. *Unhandled teardown message* - Another condition which results in automatic termination behavior is the receipt of a message that indicates that a call should be torn down (BYE or CANCEL), in the case where the application does not specify how to handle such a message. This frees the application developer from having to specify how to properly tear down each call at each point in the application. If the application should *not* terminate when it receives a teardown signal (such as in a sequence calling application), the developer is free to specify other (non-default) behavior.

3. *Exception handling* - The final condition under which automatic termination behavior is initiated is in the event of a programming error (specifically an uncaught exception). We chose this because there is no general way to judge the programmer's intent, and it is prudent in this case to clean up the calls to release resources.

4.7 Supporting Tools

4.7.1 Interactive Javadoc Documentation

In addition to the ability to generate graphical representations of ECharts machines mentioned in Section 2.2.1, the ECharts System Development Kit (SDK) available from echarts.org also includes a tool to automatically embed interactive Scaleable Vector Graphics (SVG) machine diagrams into standard Javadoc documentation. There are a number of ways that a user may interact with the ECharts SVG diagrams.

1. Highlight transitions and states by rolling the cursor over them.

2. Display user comments for a machine, its transitions and its states by resting the cursor over the respective machine element.

3. Clicking on a state with a nested submachine navigates the browser to the diagram of the submachine nested in the state.

4. Pan a diagram that may be too large to fit in the viewing window.

5. Zoom in and out of a diagram.

4.7.2 Runtime Monitor

The ECharts Java runtime also includes a comprehensive and customizable runtime monitor to support debugging ECharts machines. Entries are logged at runtime to track the machine execution as transitions fire and ports send and receive messages. This monitor has been customized for ECharts for SIP Servlets to provide formatted views of SIP messages exchanged over ports.

5. EXAMPLE

To get a sense of how these capabilities can be used together to create a feature, consider this example: a feature, called `TimeBomb`, that will terminate a call at a fixed time after it is

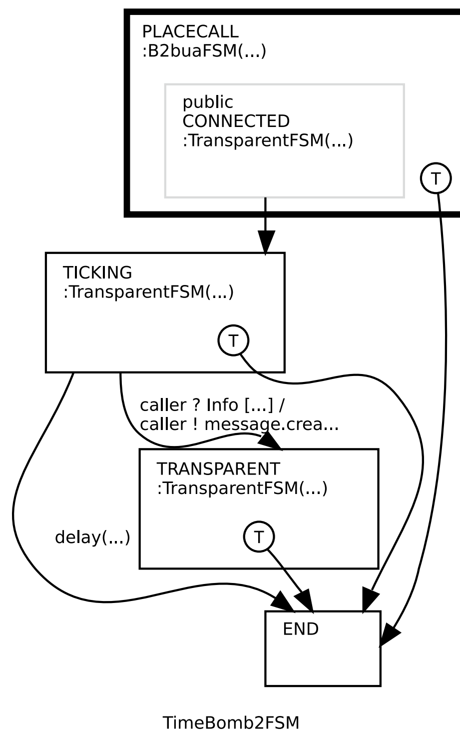

Figure 3: Graphical representation of TimeBomb program

connected, unless the feature receives a certain INFO message telling it to cancel its timeout operation. A graphical representation of the program is shown in Figure 3.

The ECharts code below represents all the state machine logic required to implement this feature. In order to make it a complete ECharts program, it would need a machine declaration as shown in Section 2.2.1 as well as a constructor to create the `SipPorts` and initialize other variables. However the application logic is completely specified below.

```
/** Place B2BUA call with no
 *  modifications to outgoing
 *  INVITE
 */
initial state PLACECALL :
    B2buaFSM(box, caller, callee, null);

/** Change state once call
 *  is connected
 */
transition PLACECALL.CONNECTED -
-> TICKING;

/** Propagate requests and
 *  responses in this state.
 */
state TICKING :
    TransparentFSM(caller, callee);

/** If in this state for at least
 *  maxDurationMsec, end call
 *  (by transition to terminal state)
 */
transition TICKING -
    delay( maxDurationMsec )
-> END;

/** If an INFO message is received
 *  from caller with "CancelTimer"
 *  in the body, then go to a pure
 *  transparent state.
 */
transition TICKING - caller ? Info
    [ "CancelTimer".equals(message.getContent()) ] /
        caller ! message.createResponse(200);
-> TRANSPARENT;

/** If embedded machine reaches a
 *  terminal state, end execution.
 *  (This will happen after completed
 *  BYE transactions.)
 */
transition TICKING.TERMINAL -
-> END;

/** Propagate requests and
 *  responses in this state.
 */
state TRANSPARENT:
    TransparentFSM(caller, callee);

/** If embedded machine reaches a
 *  terminal state, end execution.
 *  (This will happen if initial
 *  call fails.)
```

```
 */
transition PLACECALL.TERMINAL -
-> END;

/** If embedded machine reaches a
 *  terminal state, end execution.
 *  (This will happen after completed
 *  BYE transactions.)
 */
transition TRANSPARENT.TERMINAL -
-> END;

/** Terminal state: a simple state
 *  with no outgoing transitions.
 */
state END;
```

Note that three states (PLACECALL, TICKING , and TRANSPARENT) use embedded machines. The first state uses a machine to set up a B2BUA call, and the other two states use a machine that transparently propagates SIP requests and responses from one party to the other. The embedded machines referenced in this example are part of the open source distribution, and demonstrate how existing machines can be re-used to create a custom application.

Once in the TICKING state, a timed transition is declared. This is the mechanism used for signaling when time has expired. Note the transition that overrides the default transparent behavior: if an INFO request is received with "CancelTimer" in the body, then the machine transitions to a different transparent state; one with no overrides. This transition will cancel the timed transition. Note that this transition uses the ECharts *guard* syntax to specify an arbitrary boolean expression that must evaluate to true in order to satisfy the condition.

Finally, note that the automatic termination framework is used to tear down the call if the timed transition fires. Simply by transitioning to a terminal state, the application developer signals that all calls should be torn down, and the framework takes over to accomplish this. Other termination cases (e.g., caller sends CANCEL, callee returns an error response) are handled as default behavior of the B2BUA.

6. RELATED WORK

By using the ECharts model of application development in conjunction with commercially available SIP Servlet containers, we have created an effective new method of SIP application development. Of course, there are numerous other efforts underway to simplify the task of VoIP application development. Commercial vendors of SIP Servlet containers, such as Ubiquity Software [15], provide proprietary extensions to the SIP Servlet standard that support higher-level abstractions. CPL [12], LESS [16], and SPL [5] provide approaches for safe and simple SIP application development. The JAIN initiative (Java APIs for Integrated Networks) defines the Java Call Control specification [8], as well as the JSLEE application environment [9]. Parlay and Parlay X [14] provide call control APIs that are independent of the underlying network technology. The Asterisk open-source PBX [2] provides programmable VoIP or TDM services.

Finally, we are working on a domain-specific language (DSL) for programming call control services. Though ECharts provides language support for state machine constructs, it does not directly provide any language support for telecommunications-specific applications. This new language provides language primitives for call setup and teardown, syntax for specifying ports to be linked and separate notions of signal linkage and media linkage. Applications written in this DSL will be automatically translated into ECharts for SIP Servlets applications for execution.

7. ACKNOWLEDGMENTS

The authors gratefully acknowledge our close collaboration with Eric Cheung, Gerald Karam, Hal Purdy, Venkita Subramonian and Pamela Zave, out of which this work was produced. Appreciation is also due to Andrew Forrest, whose comments improved this paper.

8. REFERENCES

[1] ITU-T Recommendation Z.100: Specification and Description Language (SDL). Available from: http://www.itu.int/rec/T-REC-Z.100/en.

[2] *Asterisk :: An Open Source PBX and telephony toolkit*. Digium, 2007. Available from: http://www.asterisk.org/.

[3] G. W. Bond. An introduction to ECharts: The concise user manual. Technical Report TD-6NKLR2, AT&T, 2006. Available from: http://echarts.org.

[4] G. W. Bond and H. H. Goguen. ECharts: From lab to production. Technical Report TD-6FSMWT, AT&T, 2005. Available from: http://echarts.org.

[5] L. Burgy, C. Consel, F. Latry, J. Lawall, N. Palix, and L. Réveillère. Language technology for internet-telephony service creation. In *IEEE International Conference on Communications*, June 2006.

[6] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000. Available from: http://graphviz.org.

[7] U. Glasser, R. Gotzhein, and A. Prinz. The formal semantics of SDL-2000: status and perspectives. *Computer Networks*, 42(3):343–358, 2003.

[8] *JAIN(tm) Call Control API Specification*. Java Community Process, 2006. Available from: http://jcp.org/aboutJava/communityprocess/final/jsr021/index4.html.

[9] *JSLEE and the JAIN Initiative*. Java Community Process, 2004. Available from: http://java.sun.com/products/jain/.

[10] *JSR 116: SIP Servlet API Version 1.0*. Java Community Process, 2003. Available from: http://www.jcp.org/aboutJava/communityprocess/final/jsr116.

[11] *JSR 289: SIP Servlet Version 1.1*. Java Community Process, 2006. Available from: http://jcp.org/en/jsr/detail?id=289.

[12] J. Lennox and H. Schulzrinne. Call processing language and requirements. Technical Report RFC 2824, IETF, 2000.

[13] Object Management Group. *OMG Unified Modeling Language Superstructure Specification, version 2.0*. Object Management Group, August 2005. Available from: `http://www.omg.org`.

[14] *Parlay :: Home*. The Parlay Group, 2007. Available from: `http://www.parlay.org/`.

[15] SOOF Appcelerator(tm). Available from `http://www.ubiquity.net/products/ubiquity-appceleratorTM`.

[16] X. Wu and H. Schulzrinne. Programmable end system services using SIP. In *Proceedings of the IEEE International Conference on Communications 2003 (ICC '03)*, volume 2, pages 789–793, 2003.