Session Data Types: An Abstraction Layer for Shared-Experience Communications in Converged Applications
(Extended Abstract)

Alfred V. Aho\textsuperscript{1,3}, Glenn Bruns\textsuperscript{1}, Dennis Dams\textsuperscript{1}, Richard Hull\textsuperscript{1},
John Letourneau\textsuperscript{1}, Kedar Namjoshi\textsuperscript{1}, Frans Panken\textsuperscript{2} and H. van Tellingen\textsuperscript{2}

\textsuperscript{1}\{aaho,grb,dams,hull,}
topgun,knamjoshi\}
@alcatel-lucent.com
Bell Labs Research
Alcatel-Lucent
600 Mountain Ave.
Murray Hill, NJ
USA

\textsuperscript{2}\{frans,tellingen\}
@alcatel-lucent.com
Bell Labs Research Netherlands
Alcatel-Lucent
Hilversum
Netherlands

\textsuperscript{3}aho@cs.columbia.edu

Computer Science Dept.
Columbia University
New York, NY
USA

1. Introduction. The shift to an all-IP telecommunications core holds the promise of rapid creation of a broad variety of collaborative interactive multi-media multi-person applications, which take advantage of network-hosted communication services. An important category of such services is shared-experience communication services. By this we mean services that enable a person to be in a live interaction with other people (e.g., in a phone call, in IM, in a peer-to-peer video session), or with network components (e.g., with an audio menu-driven system, watching TV or a YouTube video), or both (e.g., two people in different places talking on the phone with simultaneous shared viewing and controlling of a YouTube video, etc.) With currently available technology, the creation of such blended services, and their incorporation of them into broader applications, typically requires familiarity with a plethora of standards and protocols at various levels of abstraction, including things like SIP (and possibly IMS), Parlay/ParlayX, PacketTV or analogous, XMPP, etc. The goal of the research reported here is to develop a novel approach for allowing web-level application programmers to incorporate shared-experience communication services into their applications, while insulating them from the heterogeneity and intricacy of the telecommunications standards and protocols.

More specifically, we are developing a framework that will support a coherent, extensible family of Session Data Types (SDTs) for managing communication services. Session Data Types are a form of abstract data types based on state-transition systems – they provide an API with clearly defined methods that can be invoked, and can be queried at any time about the session state. In addition, with their focus on communication services, they can be configured to provide notifications, and can respond to events coming from “within the network”, e.g., events coming from SIP phones that work their way through an IMS infrastructure into the SDT. As described below, SDTs are based on a natural, high-level model of shared-experience communication “sessions”.

The research described here is still in a preliminary stage, and specific aspects of the approach and constructs described in this paper may undergo revision as the research progresses.

We believe that the SDT framework can become a useful component within the larger framework of a carrier-grade Service Delivery Environment (sometimes called Service Delivery Platform). In particular, application developers could use SDTs to gain access to shared-experience communications services, and use complimentary tools to gain access to things like end-user presence and location, personalization capabilities, and end-user profile data. In particular, given that SDTs will provide a north-bound interface based on SOAP [8] and WSDL [7], and so they will provides a natural basis for combine shared-experience communication services with other services using both native languages such as Java or high-level web-services languages such as BPEL [1].

2. Motivating discussion. Suppose that an application developer wants to create a “family portal” application, that enables families and their friends to easily stay in contact, including the ability to form conference calls in simple ways (e.g., one family member calls another one, and later adds more family or friends, or a family member calls into a designated number). Suppose further that different families will have different policies concerning who can participate in such calls (e.g., friends can join only if a parent is already present).
What would be involved in creating such an application today? We focus on the case of working on top of a SIP or IMS infrastructure. In order to support a variety of policies concerning access rights to the family portal, the developer will probably want to work at a level of abstraction that makes it easy to express and enforce such policies. The conceptual model for this abstraction layer will typically be based on the participants in the call, the states they are in (e.g., active in the call, being invited into the call, self-inviting themselves into the call). Enforcement of the policies might include trapping events concerning when participants enter such states, checking a database concerning family membership, and then blocking or allowing the state changes. The developer must also address, at the SIP level, the mechanics of setting up audio call connections, incorporating a media server for bridging the audio if more than two participants arrive, and translating certain SIP-level events into notifications at the abstract level for policy enforcement.

If developing just one such application, it is pragmatic to create the code base as a silo. However, suppose now that numerous applications are to be developed, all of which involve enabling groups of people to easily come in and out of audio communication. These applications might involve diverse logic concerning how people enter and leave the calls. As a contrast with the Family Portal, key logic in the Flooding Basement example (see below) focuses on events when a person exits a phone call, rather than when they enter. In general, if each such application is developed in isolation, there will be a large amount of redundant code across them, but it would be hard to re-use that code because of the different top-level requirements of the applications.

The Session Data Type approach proposed here will establish a natural abstraction layer, that can be reused across a broad variety of converged applications that need to take advantage of shared-experience communication services. In particular, in connection with IMS-based audio communication, the SDT approach will hide the SIP-level mechanics of setting up calls and bridges, and provide a natural API that application developers can work against. It will provide similar hiding for other media types, including peer-to-peer video, on-demand video, broadcast video, and IM. This should yield a substantial cost savings in the software life-cycle costs for these applications. SDTs will be built up from a common core set of constructs, which will be layered on top of diverse forms of media. Also, the framework will be extensible, and the core SDT constructs will help provide a uniformity as the extensions are created.

Admittedly, abstraction frameworks such as Parlay/ParlayX [3] already provide some of the benefits we anticipate from the SDT framework. A key difference between SDTs and Parlay/ParlayX is that the SDT framework is based on an explicit state-transition system for the communication sessions, and applications can query that state at any time. This permits a clean separation of responsibilities between SDT and application, which appears especially useful in contexts where multiple media are involved, and in contexts involving a rich session structure with multiple sub sessions. In particular, the complete information and management of such “rich” sessions can be handled by SDTs, whereas when using Parlay/ParlayX the application will have to explicitly manage different sub sessions of the rich session as individual Parlay/ParlayX sessions. Also, the SDT subscription language is richer than that of ParlayX, because it is based on the state-transition system and can include conjunctions of atomic predicates.

3. The Session Data Type framework. Before describing the Session Data Types themselves, we briefly consider, at a logical level, the architectural framework that the SDTs will be used in. As shown in Figure 1, a Communication Service Provider (CSP) might support one or more network capabilities (e.g., Circuit, IMS, IPTV), and support a Session Management Infrastructure riding above that. This infrastructure would support one or more SDTs. Application developers would create converged applications using a WSDL interface to the SDTs. This might be used in conjunction with the SOAP protocol (and WS-* standards, e.g., [6]). The applications can query the state of the SDT, subscribe for notifications from the SDT, and can request that for some of the notifications the SDT wait for a directive from the application (“notify/respond” subscriptions). As an optimization, the subscriptions specified by the application developer might be mapped into the Session Management Infrastructure, in order to “customize” the SDT’s operation for this particular application, as suggested by the dashed line from the Service Creation Environment to the Session Management Infrastructure. (A comparison of the SDT framework with other frameworks is presented towards the end of this document.)

As suggested in Figure 1, the SDT can respond to events “above” (the converged application) and from “below” (from various devices, that interact exclusively through their native networks, e.g., SIP/IMS, PacketTV). In some cases, an end-user’s device might also have web browser capabilities, which provides a third natural path of signalling, from the end-user’s device to the converged application via a portal (or perhaps, in the future, through web services running on the devices and communicating via SOAP and WSDL to the
application). The application developer may think in terms of the kinds of events that are coming from the network, but with regards to protocols she is thinking exclusively in terms of standard web protocols, notifications, and operation requests (i.e., not working at the messaging level as in SIP).

4. Session Data Types. We now describe the conceptual model underlying SDTs. We begin by listing three underlying principles which are guiding their design.

(a) Factor out commonalities for session management. The SDT framework provides a uniform, modular layer of abstraction for essentially all shared-experience communication services (and possibly some closely related ones). While the underlying implementations of services for diverse media and devices are varied, web-level application developers will want to focus primarily on the high-level sessions that they can enable. These sessions in turn revolve around three basic building blocks: participants (and their devices), media “channels” (understood in a very broad sense) that connect the participants into shared-experiences, and the ability to manage access permissions, so that some or all of the participants can manipulate appropriate aspects of the session and participation in it. The SDT framework supports the notion of multiple subsessions, possibly spanning multiple media or possibly within a single media (e.g., audio teleconference with the possibility of audio “whisper” conversations, i.e., side conversations for a subset of the participants).

(b) Separation of concerns. A primary theme here is that the SDT will have the responsibility to maintain the state information about communication sessions (e.g., who is participating via what devices, and how are they in communication with each other) whereas the application has primary responsibility to maintain state information about the specifics of the application. (e.g., if relevant to the application, then what are the relationships between the participants and do some of them have special privileges). The application can query the SDT at any time for state information, and can hold some of the state information, but only if it wants to.

(c) Structured extensibility through a hierarchy of SDTs. Unlike previous endeavors for abstraction of communication services such as the JAIN Java Call Control (JCC) API [9] and Parlay/ParlayX, the framework described here will enable the creation of a family of SDTs, rather than a single, one-size-fits-all SDT. In our view, “simple” communication tasks should be possible use a “simple” SDT, and richer communication tasks should be able to use a richer SDT. For example, many applications that want to take advantage of simple audio connectivity could effectively use an SDT with just a handful of states, to handle entry into the call and then active or suspended participation (see the AudioSDT below). But there will also be applications that want richer capabilities, such as authentication, interaction with audio menu systems, ability to record, and the ability to set up whisper sessions. We thus envision a hierarchy of SDTs focused on audio. Similar hierarchies will arise for other kinds of media, such as IPTV (simple viewing, viewing with pause and buffering capability, shared viewing across locations), peer-to-peer video, IM, and other forms of media. Finally, there will be SDTs that incorporate multiple media types (within subsessions). Importantly, in our framework there is a small number of basic constructs for assembling SDTs (including primarily states, transitions between them, grouping of states as in Statecharts [4], a cross-product construct, and a construct for subsessions). Further, there will be an inheritance relationship in this hierarchy, so that applications written against a simpler SDT can be easily ported to work on a richer one.

Although space limitations prevent a detailed discussion of the SDT constructs, we briefly illustrate the idea with a simple SDT, called AudioSDT. Figure 2 shows an informal diagrammatic representation of this SDT. An instance of this SDT might be launched at some point in time, it might include several participants that come and go, and it might or might not be terminated at some later time. Before launch (and after ter-

Figure 1: Session Data Type Framework

Figure 2: Informal diagrammatic representation of AudioSDT, a representative SDT
mination) the instance is conceptually in the Idle state. Once launched, it is globally in the Active state, and each individual participant will move through the states shown within the double rounded rectangle. Each participant starts in the Idle state, may move into either the Invited or Self-inviting state, and might later move into one of the four states in the rightmost rounded rectangle. Here the Muted state occurs if the participant is placed on mute from the network. The Suspended without audio state arises if the participant is placed into a suspended condition (he can’t hear anything and can’t be heard); and Suspended with audio state means that the participant is suspended and receiving some pre-provisioned audio. The '*' next to the Active state indicates that this is the default state that participants will enter if they leave the Invited or Self-invited states and enter the right-most rounded rectangle. In general, an application writer can indicate default states or transitions as part of the customization of SDTs. As suggested by the informal notation, the constructs for SDTs are inspired by those from Statecharts but SDTs will have a formal semantics.

In the SDT framework, the Session Management Infrastructure supports four kinds of interaction with an application, namely

1. Notify: The application developer can subscribe to be notified about events that arise in connection with the SDT. This includes situations where a participant enters a state, a participant leaves a state, a particular state transition occurs, or there is a request (either for a state transition or a more general request, e.g., coming from an end-user device up through the network). The subscription language is inspired by the pointcut language from AspectJ [2], and supports conjunction and negation on atomic predicates. (WS-Notification [6] and related protocols might be used to carry these notifications and the subscriptions that request them.)

2. Notify/Respond: For notifications about requests, the developer can indicate that they want the SDT to not take action on the request until the application indicates what action should be taken (e.g., an end-user device might send an event which is essentially a request to invite another participant into the call, but the application might want to deny that request, or insist on an authentication step before proceeding with it.)

3. Request Session Operation: The application can invoke a valid session operation (e.g., to change a participant state) at any time. Among other things, these requests might be the result of an interaction between an end-user and the application via a browser.

4. Query: The application can use a targeted query language to ask about the current status of the SDT and participants in it.

The SDT also supports an interface to end-user devices through the underlying network. For example, with SIP phones it may be that phone SIP events come through an IMS infrastructure into the Session Management Infrastructure, which is acting as an IMS Application Server (AS). For the present, the interface between devices and the SDT is focused on requests that have to do with modifying the session state (including “off-hook” when answering a call, and “hang up” when ending it), and might also include queries against the SDT state, if the network- and media-specific telecom protocols allow this kind of information transfer.

5. The SDT framework in action. We anticipate that the SDT framework will make it relatively straightforward to incorporate shared-experience communication services into diverse applications such as those for enterprise workflow management, social networking sites, customer care, or enhancing multi-player games. We now illustrate how the AudioSDT might be used by a couple of example applications; these are focused mainly around the shared-experience communication but are suggestive of how broader applications might take advantage of the SDT.

Example: Family Portal. As suggested above, this application could be used to make it easier for families to keep in contact. We focus here on the audio component, although web-browser controls could be added, and also additional media (assuming that the underlying SDT incorporates such media). In essence, the Family Portal creates a virtual audio chatroom that family members and friends can join into at any time. In a prototype system that we’ve created, if one family member calls another, this can be trapped by the IMS infrastructure and used to “launch” an instance of the AudioSDT which operates under the direction of the Family Portal application. Additional family members or friends can be invited into the call, or family members can call a designated number to get in. Policies can be enforced from the application, e.g., that friends cannot be included into the call unless a parent is currently on the call. As shown by the prototype system, the application uses subscriptions to two kinds of events (when someone is invited into the call, and when someone invites themself into the call), maintains its own database about family memberships, and on certain occasions has to query the SDT about who is on the call (namely, to check if a parent is on the call when a friend is invited into the call). We note that the logic underlying the Family Portal application can be used in a variety of other contexts, such as in support of volunteer groups, study groups, enterprise project groups, etc.

Example: Flooding Basement. This application provides support for an emergency calling service for a
small enterprise, e.g., a plumbing company. An instance of the AudioSDT is launched in support of this application whenever a customer calls a designated number. The application places the calling customer on hold (Suspended status, possibly with music), and then locates some plumbers and a manager, and invites them into the call. (Only some of the plumbers might answer these invitations.) Once the manager is on the call then the customer is moved by the application to the Active state, so that everyone can discuss the emergency. The manager has the ability to put the customer back to Suspended status and/or back to Active status. Eventually a plumber is assigned to fix the problem and the manager and other plumbers hang up. If the customer is in the Suspended status at that point (when there is just one plumber left), then the application puts the customer back into the Active status. As a variation on this service, the application might be set up to operate in an appropriate fashion even if no manager joins the call (e.g., move the customer into the Active state if there is at least one plumber on the call and 60 seconds have passed). Note that the notifications used by this application are different than for the Family Portal; here they focus on the initial call from the customer, and then the events arising from people dropping off the call. This highlights how the same underlying SDT can be used to support two very different applications.

A key motivation for the development of the SDT framework is to facilitate applications which take advantage of two or more media. Figure 3 illustrates one simple example of this, along with a possible architectural realization. In particular, suppose that the Blender Application shown wants to take advantage of both audio communication services and IPTV broadcast services. (This might be to support variations on caller notification on the TV screen, or enabling shared viewing experiences). Suppose further that a Communication Service Provider is supporting both audio telephony and IPTV broadcasting, but over two separate network infrastructures. In this case, separate SDTs for the audio and the IPTV might be enabled, each on top the appropriate network. While the application developer is aware of the different characteristics of the different media, he is subscribing to notifications and impacting the media behaviors through the uniform paradigm of SDTs. Further, the two SDTs provide a natural way for thinking about the full set of interactions between the two media that might need to be considered. Indeed, the full space can be viewed as a kind of product of the state machines underlying the two SDTs. We note that from the perspective of the logic in the Blender Application, there will be no change if the IMS and IPTV infrastructures were provided by two separate entities (e.g., a wireless services provider and a cable TV broadcaster that operated in a loose partnership). In the other direction, one can imagine that there might be a single Session Management Infrastructure, which taps into both the IMS and IPTV networks, and which supports a unified SDT that covers both media types. In this case, the Blending Application could take advantage of subscriptions which incorporate simultaneous state information about both medias.

6. Related work. The SDT framework presented here follows a long line of activities that provide various levels of abstraction in support of simplifying access to communication services, and it freely borrows key ideas from most of these earlier activities. Similar to the (Advanced) Intelligent Network (AIN) framework that emerged in the 1990s, the SDT framework supports subscriptions, notifications, and direct impact on communication services. Unlike AIN, the SDT framework uses modern web-level protocols, and provides for certain optimizations (through “pushing” some behaviors such as default transitions into the SDT). The framework also provides for a crisply defined and explicit model of session management based on state transition systems, including the ability for applications to query SDT state at any time. The Parlay and ParlayX standards [3] provide CORBA and SOAP/WSDL (respectively) access to a broad variety of network services, including multi-party, multi-media calls. These standards include substantial infrastructure for controlling access, including the amounts of service usage different applications use. The SDT framework is focused more narrowly on session-based services (but not location, presence, etc.), gives more emphasis to the idea of the network holding session state. It offers a richer subscription language, and also provides more flexibility in representing and managing sub-sessions. It also supports a family of SDTs built from a small set of constructs, rather than a single, one-size-fits-all API. The JAIN Java Call Control (JCC) framework [9] that layers on top of the JAIN API is quite close to the SDT approach, in that it provides an API and an explicit
underlying state-machine model for managing multimedia calls. The JCC was designed primarily for usage within a single application server, rather than providing an API for third-party interactions. Also, as with Parlay/ParlayX, the JCC attempts to define a single, one-size-fits-all state machine for managing calls, whereas the SDT framework envisions the development of tens of SDTs within a hierarchical framework.

SIP servlets provide an approach for encapsulating basic SIP capabilities into richer aggregates, and typically assume that access to the aggregated service is again via SIP. SIP servlets might provide a useful basis for building up SDT capabilities for networks and media where SIP is the dominant protocol; the SDT framework is intended to provide abstraction for other protocols as well (e.g., PacketTV, XMPP). The SDT framework can provide a natural abstraction layer on top of the IMS infrastructure. The Services Capability Interaction Manager (SCIM) function within IMS can provide some support for blending of services, but this is entirely SIP-based, and again at a lower level than the SDT abstraction. Similarly, most current “Service Brokers” are focused at a SIP level. (An exception is the Service Broker of [5], which provides capabilities for fine-grained blending services spanning the IMS, CAMEL, and web domains.)

7. Conclusions. The SDT framework presented here provides a targeted, straightforward, high-level abstraction layer that will provide web-level application developers convenient access to network-hosted shared-experience communication services. As such, we feel that the approach has a strong potential in enabling a rich level of creativity in the conception and development of web-level applications that take full advantage of those services. In particular, the SDT framework provides a clear division of responsibilities (in that the SDT is focused on the communication services and the application is focused on other things), and it provides a natural structure for the blending of the communication aspects of multiple applications.

Because the SDT approach is based on the use of a formally defined state-machine model, we expect that it will enable the development of rich and useful verification tools based on model checking and other modern software verification technologies. These would be used to check whether applications written against an SDT are using the SDT appropriately, whether there can be deadlock, whether there is unintended usage of network resources, etc. The tools would also help to detect feature interactions, and ensure that the applications resolve them adequately. Verification tools for communication services will be increasingly important as the number of converged applications written by third-party developers grows. We also expect that the state-machine based underpinnings of the SDT framework will facilitate OA&M, e.g., in the area of tracability for service usage and bug correction.

The SDT project is at a quite preliminary stage; the formal framework will surely evolve as we gain more understanding. A broad variety of questions remain unanswered at this point; we conclude by mentioning just one of them here. Proper error handling is often a large part of a working system. We believe that the use of a higher level of abstraction in SDTs (state machines) and their hierarchical structuring will make it easier to indicate how errors should be handled, but this needs further study.

Acknowledgements This work is supported in part by NSF grants IIS-0415195 and SOD-0613998.

References


