Beyond Discrete E-services: Composing Session-oriented Services in Telecommunications

Vassilis Christophides[1], Richard Hull[2], Gregory Karvounarakis[1,2], Akhil Kumar[2], Geliang Tong[2], Ming Xiong[2]

[1]Institute of Computer Science, FORTH,
Vassilika Vouton, P.O. Box 1385, GR 71110, Heraklion, Greece.
{christop, gregkar}@ics.forth.gr

[2]Bell Laboratories, Lucent Technologies, Murray Hill, NJ 07974.
{hull, akhil, tong, xiong}@research.bell-labs.com

Abstract. We distinguish between two broad categories of e-services: *discrete* services (e.g., add item to shopping cart, charge a credit card), and *session-oriented* ones (teleconference, collaborative text chat, streaming video, c-commerce interactions). Discrete services typically have short duration, and cannot respond to external asynchronous events. Session-oriented services have longer duration (perhaps hours), and typically can respond to asynchronous events (e.g., the ability to add a new participant to a teleconference). When composing discrete e-services it usually suffices to use a process model and engine that composes the e-services as relatively independent tasks. But when composing session-oriented e-services, the engine must be able to receive asynchronous events and determine how and whether to impact the active sessions. For example, if a teleconference participant loses his wireless connection then it might be appropriate to trigger an announcement to some or all of the other participants. In this paper we propose a process model and architecture for flexible composition and execution of discrete and session-oriented services. Unlike previous approaches, our model permits the specification of scripted "active flowcharts" that can be triggered by asynchronous events, and can appropriately impact active sessions. We introduce here a model and language for specifying process schemas (essentially a collection of active flowcharts) that combine multiple e-services, and describe a prototype engine for executing these process schemas.

1 Introduction

The use of web-accessible e-services will revolutionize the way that many e-commerce and consumer software applications are provided. Until now, much of the research (e.g., see [3,6,7,18,24]) and emerging infrastructure (e.g., IBM's Web services Toolkit, Sun's Open Net Environment and JiniTM Network technology, HP's e-speak, Microsoft's .Net and Novell's One Net initiatives[3]) in e-services has been

[1]See www.alphaworks.ibm.com/tech/webservicestoolkit, www.sun.com/sunone, developer.java.sun.com/developer/ products/jinni, www.e-speak.hp.com, www.microsoft.net/net, and www.novell.com/news/onenet, respectively.

focused on the composition of *discrete*, *short-running* tasks such as "add an item to a shopping cart", "charge a credit card" or "check the availability of tickets". While the APIs of such services may include several methods that can be invoked, they are typically *unresponsive* or *unaware* of *asynchronous events* arising from other e-services or from external applications. In contrast, there are several kinds of *session-oriented* e-services that do need to respond to asynchronous events during their life-cycle. Such e-services arise in telecommunications, c-commerce [4], and cross-organizational workflows [16]. We use the term "responsive" for e-services that need to respond to asynchronous events in their environment, and "insular" for e-services that can be isolated from such events. This paper introduces the AZTEC framework, which uses a new process model and architecture that enables highly flexible, scripted handling of asynchronous events in composite e-services involving responsive sessions.

Many e-services for telecommunication applications, such as voice calls, teleconferences, internet-based multimedia chat or collaboration, single- or multi-participant streaming video sessions and interactive games, are session-oriented and responsive. For example, all of them need to respond (by shutting down) if they are used in conjunction with a pre-paid billing account that runs out of money. It may be desirable to impact a teleconference if a participant drops out (e.g., because they move out-of-range of wireless connection), perhaps by informing the other participants. Another class of examples arises when using "presence" services, which generate messages when someone becomes present on a network (e.g., by turning on their cell phone, or typing something on a keyboard). Presence information could be used to automatically connect an invited participant into a teleconference, or to alert viewers of a streaming video that an interested friend has just become present.

Moreover, the notion of collaborative commerce (c-commerce) is focused on supporting all aspects of electronic communication and interaction between the (human and automated) participants in commercial transactions. Many of these interactions have long duration (ranging from a phone call to a catalog-sales company to the month-long process of obtaining a home mortgage), and are impacted by events generated by the involved participants, and by external applications. As just one example, it may be useful to automatically monitor customer sessions at a web-based storefront and proactively intervene, by enabling a sales person or product expert to join into the session [2]. Finally, in the context of cross-organizational workflows, sub-workflows are usually packaged as outsourced e-services. Then, an execution of a sub-workflow may both generate and respond to multiple events [16].

To provide a framework for assembling, executing, and monitoring insular (discrete) and responsive (session-oriented) e-services we combine elements of the workflow and event paradigms. This combination permits a loosely coupled process definition environment. More specifically, workflow-style constructs can be used for specifying how to respond to a given event type, but the specifications of these responses do not need to be embedded into a single workflow schema. This allows a modularization of the specification of a composite e-service, in which the event responses can be considered as logical building blocks. This is consistent with the increasing autonomy between activities (e.g., outsourced e-services) that a web-enabled distributed architecture can provide.

Our contribution in this paper is a model, a language and an engine for creating composite e-services involving sessions. We propose an *Active Flowchart Model*, supporting the specification of process schemas (that define a composite e-service) essentially as a collection of "active flowcharts", i.e., flowcharts coupled with the event types that can trigger them. The language used to define active flowcharts is called XASC (XML-based Active Service Composition). It is XML-based in two ways: the active flowcharts are themselves specified in XML, and the interfaces between the flowcharts and e-services are based on SOAP XML messages [21]. The engine is event-driven, supports explicit prioritizations between flowchart enactments, as well as simultaneous execution of multiple process enactments. The engine forms one component of the AZTEC system, which also provides the means for automatic generation of process schemas using higher-level specifications.

Since the main focus of this paper is on the process model and runtime execution engine for composite e-services involving sessions, we do not address issues such as publishing, registering, or selecting atomic or composite e-services using emerging technologies like UDDI (Universal Description Discovery and Integration), and WSDL[4] (Web Services Description Language). We note that AZTEC builds on top of standards for accessing session-based telecommunications services (e.g., SIP, Parley), and that Lucent and other providers are currently developing technologies (e.g., SoftSwitch, PacketIN, Flexant) that enable invocation and interaction with session-oriented telecommunications services programmatically, i.e., as e-services.

The paper is organized as follows. Section 2 identifies key issues raised when composing session-oriented e-services, and motivates the various elements of our approach. Section 3 presents the Active Flowchart Model, including a high-level specification of the XASC language and its semantics. Section 4 describes the architecture and run-time environment of AZTEC, including both the execution engine and the components for automated generation of process schemas. Section 5 discusses related research, and Section 6 presents our ideas for future work.

2 Motivation and Approach

This section examines more closely the fundamental issues that arise in composing responsive, session-oriented e-services, and then describes the key components of the approach taken by AZTEC. We introduce a representative class of composite e-services involving sessions to provide grounding for the discussion. The example is called "(Design your own) Smart Teleconference", and comes from telecommunication applications. Similar needs are also exhibited in the context of collaborative commerce and cross-organizational workflows.

With existing technology it is possible to request that a phone-based audio teleconference be set up, to run from a start time to an end time, with a given number of ports (i.e., participants). This kind of teleconference supports a very limited set of automated functions – people can join the teleconference or exit it, and perhaps the group can request extensions of the teleconference (e.g., add 15 minutes). The charge

[4]See www.w3.org/TR/SOAP, www.uddi.org, and www-106.ibm.com/developerworks/library/w-wsdl.html respectively.

for the teleconference is based on the maximum number of ports requested not the number of ports used. Adding more ports typically requires operator intervention.

Based on emerging technologies in the telephony network it will soon be possible to dynamically assemble and invoke much richer forms of teleconferencing. In our hypothetical Smart Teleconference application, a user will be able to request that a multimedia teleconference be established with a given start and end time, a given set of invited participants, a set of different interaction formats (e.g., audio bridge, internet-based text chat, internet-based collaborative web browsing, shared view of video streams), the use of presence services, and guidelines concerning quality of service, costs, and billing model (e.g., pre-paid or account-based). As a particular example, Rick may request that a smart teleconference involving an audio bridge and possibly a text-chat bridge be set up to run between 10 AM and 12 noon, involving five participants, namely Akhil, Geliang, Gregory, Ming, and Vassilis. A presence service is to be used, both to identify whether an active participant loses their wireless connection, and to automatically connect an invited participant if he/she has not been active but becomes present on a network (e.g., by turning a wireless device on). Also, the teleconference will be charged to Rick's pre-paid account (which happens to have $25 in it), and the overall expected cost is to be minimized.

In this example, during execution, there will be four sessions in operation, each potentially interacting with the others. For example, if a participant drops from the audio conference then the other participants might be notified through the text-chat session. Likewise, the presence server may lead to the automated connecting of new participants. Finally, the billing session may receive periodic updates about the services being rendered by the other sessions, and may impact the other sessions (by shutting them down) if the account runs dry.

In the AZTEC framework we view the sessions as being wrapped, to form *session objects* with an API that includes synchronous function calls and generated events. The wrappers would typically translate between the internal representation of functions and events and the SOAP interface [21] supported by the e-service realizing the session. The wrappers can also transform asynchronous function calls into synchronous function calls (e.g., to yield a synchronous function that asks the telephony network to place a phone call and then give as a return value information about whether the call was answered, busy, or rang until a time-out occurred).

A primary purpose of a process schema composing multiple session objects is to specify how these objects are to interact. We identify three challenges of managing these interactions:

1) Knowing *what* is interacting: Inquiring about the explicit state of external session objects is not always possible. From a requester viewpoint the objects might have to be treated as black boxes supporting a well-defined but limited interaction interface.

2) Knowing *how* the session objects should impact each other: The impact of one session object on another will be application dependent, i.e., depending on the goals of a specific composition of e-services. In addition, the logic used when reacting to events generated by session objects will depend on the state of other session objects and the state of the overall process enactment.

3) Knowing *when* the interactions will take place: Interactions with autonomous session objects can't be specified statically, since the service requester has a limited

ability to supervise the session object, and the session object may be reacting to multiple non-deterministic events not directly visible through its interface.

The AZTEC framework responds to items (1) and (2) by permitting the use of flowcharts to specify how to respond to a given session-object event. In particular, these flowcharts can probe the session objects as to their current state, and then impact one or more of the session objects. The flowcharts provide many of the advantages of workflow models (e.g., separation of control logic from tasks performed), but also support lower-level data manipulation constructs (e.g., to restructure or merge data from different events). As will be seen below, constructs are provided in AZTEC so that one flowchart enactment may launch another one.

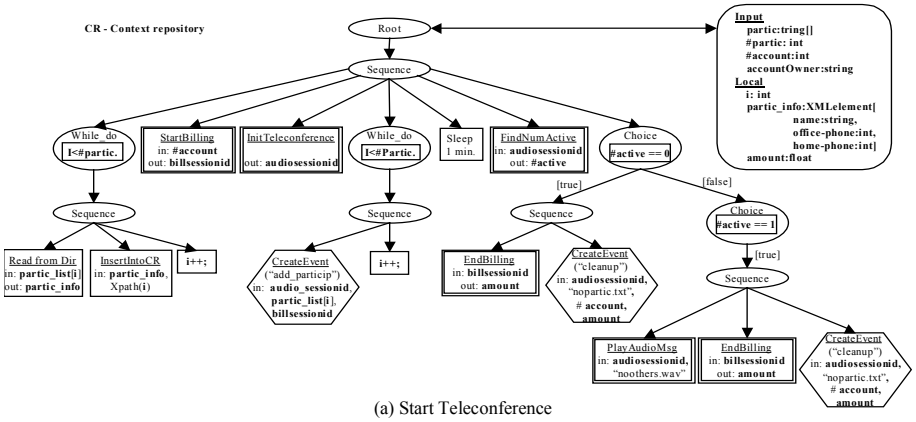
One implication of item (3) is that it cannot be predicted in advance when or how frequently a given session object event will occur. This is the fundamental motivation for incorporating the event-driven paradigm into our model, rather than attempting to extend any of the commonly used workflow models, so that all the flowcharts could be combined into a single workflow schema.

We note that the enactment of a single flowchart may involve numerous requests against external session objects (and perhaps databases, etc.), and may thus take hundreds of milliseconds or even multiple seconds or minutes to execute. This is in marked contrast with the approach taken by action algebras based on situation calculi. In those models, the action taken when an event is received is viewed as atomic and instantaneous [20]. The potential for long-running enactments and for events to arrive soon after one another leads to the potential for interleaved flowchart enactments, which is another implication of item (3). In some cases there may need to be some prioritization and/or interaction between two or more flowchart enactments. In AZTEC, we provide both priorities and also the ability for one flowchart enactment to suspend, examine, modify and resume another.

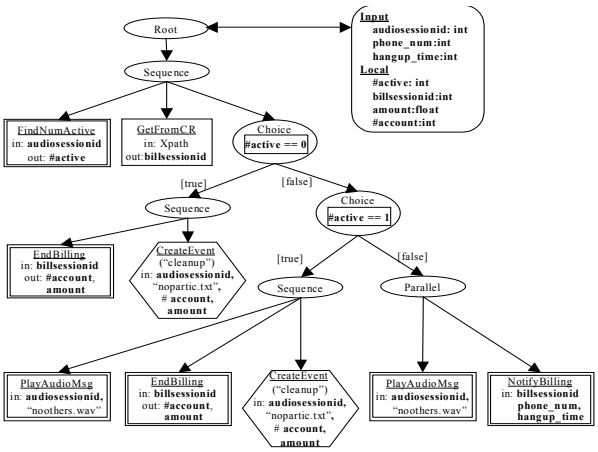
Finally, one could naturally ask: Why use flowcharts, as opposed to a full-fledged programming language, as in Java beans? One motivation concerns the need for some flowchart enactments to access, and perhaps manipulate, the state of another enactment. Since flowcharts have restricted flow-of-control logic that is easily accessible, they are easier to reason about and manipulate than unrestricted code blocks. The other motivation stems from an important goal of AZTEC, which is to enable the automated construction of process schemas, i.e., composite e-services, to achieve some high-level requirements. Although automated assembly is not the focus of this paper, we make a few remarks about how AZTEC will support this process in Subsection 4.3.

3 The Active Flowchart Model

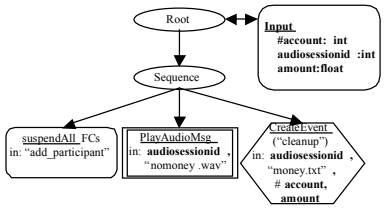
A key focus of the AZTEC platform is to support session-oriented services, responsive to asynchronous events. A key design criterion for AZTEC is to enable the specification of highly customized and/or personalized reactions to asynchronous events coming from external e-services. To this end, we introduce a process model for active flowcharts i.e., flowcharts that can be invoked asynchronously whenever a matching event occurs (in the style of Event-Action rules [25]). To illustrate how the



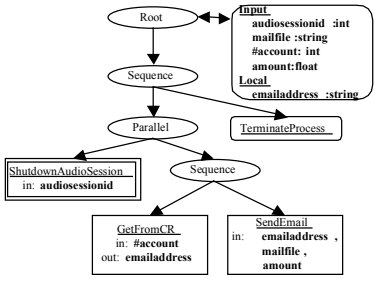
(a) Start Teleconference



(b) Participant Hang-up



(c) Out of Money



(d) Cleanup

Fig. 1. Flowcharts from audio-conference process schema

Active Flowchart Model and the AZTEC platform can support a smart teleconference we rely in the rest of the paper on a simplified example with only audio conferencing, and a pre-paid billing model. In our example, we consider an event-based interaction with two session-oriented e-services appropriately wrapped as session objects using SOAP [21]: (a) the *audio_conf_service*, that can initiate audio conferences, add participants to it, and monitor when participants have hung up; and (b) the *billing_service*, that maintains pre-pay accounts, can keep track of how much money is being used during an audio conference, and generate an event if the account runs out of money. Five active flowchart schemas need to be defined for this example: (1) **Start_audio_conference**, (2) **Add_participant**, (3) **Participant_hang_up**, (4) **Out_of_money**, and (5) **Clean_up**. We are giving below the abstract syntax of the XASC language, used to define these flowchart schemas, and we will explain its various constructs using the graphical representation of our flowcharts in Figure 1 (a concrete XML syntax is forthcoming).

An XASC process schema (e.g. Smart Teleconference) consists of a set of active flowchart schemas, an identified root flowchart, and an XML Schema [22] for the Context Repository. For each active flowchart, the enactment-priority indicates the priority with which the flowchart should be executed, if invoked by the specified event type.

<process-schema> :== <active-flowchart>$^{+}$<root-flowchart-name><CR-schema>
<active-flowchart> :== <event-name><flowchart-name><enactment-priority>

In our example, executing the XASC process schema causes the root **Start_audio_conference** flowchart to be launched. This collects participant profile data (including office and home phone numbers), initiates an audio conference, notifies the billing service about the new audio conference, and then launches an enactment of **Add_participant** for each invited participant. The latter flowchart attempts to add one participant to the audio conference, first by ringing their office phone and if no answer then ringing their home phone. If the participant answers, then the billing service is notified. Although not illustrated in our simplified example, the **Add_participant** flowchart might also be invoked in the middle of the audio conference. If a participant hangs up then the audio_conference service generates an event, which in turn launches the **Participant_hang_up** flowchart not presented here. If at least two participants remain, then the flowchart enactment simply informs the billing service that one participant dropped; if zero or one participant remains then the flowchart enactment invokes the **Clean_up** flowchart. The **Out_of_money** flowchart is launched if the billing service generates an event indicating that the account against which the audio conference is being charged has run out of money. This flowchart plays a message to the participants telling them that the account is out of money, and then invokes the **Clean_up** flowchart. Importantly, the **Out_of_money** flowchart also suspends the operation of all flowchart enactments of type **Add_participant**. Finally, the **Clean_up** flowchart has the job of requesting the *audio_conf_service* to end the conference.

Before continuing we note that our model supports four *modes of interaction* between flowchart enactments: (1) by generating events that launch other flowchart enactments; (2) by sharing data in the Context Repository; (3) by querying the status

of other flowchart enactments, and even suspending and subsequently altering the activity of other enactments; and (4) by using priorities to enforce a certain execution order between steps of flowchart enactments. More details are presented below.

Active flowcharts essentially subscribe to various event types in order to be asynchronously notified by external e-services (session objects), or other flowcharts. Both events (i.e., messages) and flowcharts (i.e., processes) are first-class citizens in our model. Despite the fact that in our example there is a one-to-one correspondence between event types and flowchart schemas, an XASC design choice was to favor modularity of definitions. Thus, different process schemas may reuse both event types related to specific session objects (e.g., in case of new flowcharts using the same e-services) and flowchart schemas (e.g., in case of new e-services composed using the same flowcharts).

```
<event>       :== <event-name><event-arg>*
<event-arg>   :== <parm-name><parm-type>
<parm-type>   :== <XMLSchema-Type>
```

Event types are defined by their name, and the name and the type of their input parameters. These parameters are used for passing data from a session object to an active flowchart enactment and vice versa (i.e., the data flow), as well as, from one flowchart enactment to another. XASC event and flowchart parameter values are XML data, typed according to a process-specific XML Schema.

```
<flowchart>            :== <flowchart-signature> <flowchart-body>
<flowchart-signature>  :== <flowchart-name> <flowchart-arg>*
<flowchart-arg>        :== <parm-name> <parm-type>
<flowchart-body>       :== start <flowchart-var>* <subflow> finish
<flowchart-var>        :== <parm-name> <parm-type>
<subflow>              :==  begin-seq <subflow>+ end-seq
                          | begin-parallel <subflow>+ end-parallel
                          | begin-choice <condition> <subflow>+ end-choice
                          | begin-loop <condition> <subflow>+ end-loop
                          | task
<condition>            :== <condition-var>+ <literal>
<condition-var>        :== <parm-name> <parm-type>
<task>                 :==  <task-input-arg>* <task-output-arg>*
                          ( <external-task>
                           |<internal-task>
                           |<event-task>   )
<task-input-arg>       :== <parm-name><parm-value>
<task-output-arg>      :== <parm-name><parm-value>
<parm-value>           :== <XMLSchema-Instance>
                          |<parm-name>
```

Flowchart signatures state their name and input parameters. Note that the signature of enabling events should subsume the signature of the flowcharts. This can be statically checked during an XASC process schema compilation [17]. The body of a

flowchart is defined using a structured workflow language introduced in [15], although extended to permit input parameters and local variables. One motivation for using this model is the relative ease of querying flowchart enactments about their state [10]. For **Start_audio_conference** the input variables are an array of participants (names), the size of the array, an account number to be billed, and the conference "sponsor", which will be notified about status and completion of the audio conference. It also uses local variables, which are private to a single flowchart enactment. These enactments can also access the Context Repository, which enables sharing of information between flowchart enactments.

Each flowchart has a unique root task (delimited by start and finish) and the control constructs (seq, parallel, choice, loop) are properly nested (matching respective begin and end tags). For instance, the **Start_audio_conference** flowchart performs seven sequential tasks: (a) insert participant profile information into the Context Repository, (b) initialize the billing session, (c) start the audio conference session, (d) add participants into the audio conference, (e) sleep for a short period (say, 1 minute), (f) get the number of active participants, and (g) check whether the audio conference was successfully launched. Boolean conditions of the choice and loop control tasks are evaluated against flowchart input arguments or local variables. In **Start_audio_conference** we can see the while-do loop condition (*i*<*#part*) for finding participant info (office and home phone numbers) from a directory, as well as the choice branches (*#active*=0, *#active=1*) for ending the teleconference when the number of active participants is not sufficient. Additionally, in the **Clean-Up** flowchart we can see the parallel construct used to shutdown the audio session while notifying the teleconference sponsor about the total cost.

Various kinds of XASC tasks exist with corresponding input and output parameters. External tasks (represented with double squares) make synchronous function calls to session-oriented and discrete e-services, while internal tasks (represented with simple squares) use available AZTEC functions/operations. For example, in the **Start_audio_conference** flowchart, the second and third sequence tasks are synchronous calls to the *billing* and the *audio_conf* e-services in order to initialize the corresponding sessions and obtain respectively the billing and audio session id. Internal processing tasks implement various operations, including manipulations of literal data and accesses to the Context Repository, namely insert (creating new values), set (replacing values) and get operations (reading values). Each time an XASC process schema is executed, the Context Repository is initialized as a new XML document, instance of an XML Schema specified by the application programmer. For instance, the second sequence task of the first while-do in **Start_audio_conference**, inserts into the created context repository the participant information (name, office and home phones) that will be used by an enactment of the **Add-Participant** flowchart to attempt to add a participant into the teleconference.

Last but not least, tasks may be used to explicitly generate events (represented with hexagons). In **Start_audio_conference**, the first sequence task of the second while-do loop will create an **Add-Participant** event, having as input parameters the participant name (*partlist[i]*), as well as, the initialized audio and billing session ids. Each occurrence of such event has the effect of invoking an **Add_participant** enactment asynchronously. Furthermore, in the seventh task (choice), when only one person is detected to be active (since presumably no other participant had been

reached) a voice message is played, the billing session is closed, and an event to is generated to launch the **Clean_up** flowchart. This appropriately closes the audio session, and sends an e-mail to the teleconference sponsor. The last task of **Clean_up** will terminate the entire process enactment, including the cancellation of any extant flowchart enactments, logging anything important from the Context Repository and then freeing up that memory.

A special category of internal XASC tasks (represented with rounded squares) enable to directly suspend, resume or cancel the execution of tasks of a running flowchart enactment, as well as, to examine the current state of an enactment (e.g., using techniques of [10]). The first kind of task is illustrated in the **Out-of-Money** flowchart, which will be launched when the pre-pay billing account runs out of money. Since the objective of this flowchart is to shutdown the entire teleconference, its first task will suspend the tasks of all running enactments of type **Add-Participant**. Clearly, we will not continue attempts to add a participant if the audio conference is being cancelled. The third task of the flowchart will, as previously discussed, launch the **Clean_up** flowchart.

We close this section by commenting on the final mode of interaction between flowchart enactments, namely the use of priorities. For example, the **Out_of_money** flowchart may be given priority over the **Add_participant** and **Participant_hang_up** flowcharts. The prioritization can also be used to favor certain flowchart enactments during periods of heavy load. For example, **Add_participant** enactments might be given higher priority than **Participant_hang_up** ones, because it is important to quickly connect people to the audio conference; delays in clerical updates to the billing session can also be delayed.

4 The AZTEC Platform

This section provides a brief introduction to the AZTEC platform for composing and executing discrete and session-oriented e-services that is currently being developed at Bell Labs. The overall architecture of the system is described in Subsection 4.1. Subsection 4.2 describes how the AZTEC platform executes schemas specified in this process model. Finally, Subsection 4.3 describes how AZTEC supports dynamic changes to process schemas.

4.1 Overall architecture

The AZTEC platform provides support for service selection and assembly, and for executing the resulting composite service. Note that assembly and execution can be performed as two distinct phases, or can be interleaved, thereby supporting a form of dynamic service selection and assembly. Figure 2 illustrates the main components of the AZTEC platform (shown as the large rounded-corner square), along with an Administration component (upper left) and access to web-services, the telephony network, and the wireless network (across the bottom).

In the upper right of Figure 2, we can see the **Assembly component**, that performs service selection, creation of process schemas, and if needed, the dynamic revision of

process schemas. Under typical usage, when a request for a composite service (e.g., a smart conference with various characteristics) is presented to the AZTEC platform, then the Assembly component will analyze the requirements and build a process schema that can support the requested composite service. In the current design, AZTEC's Assembly component uses a form of hierarchical planning [11], that starts by selecting a high-level "template" process schema which may have "slots" that need to be filled in, and then progressively fills in the slots with more detailed templates or grounded schemas (i.e., schemas with no slots). The templates and slot fillers are stored in the **Templates and Fillers Library**, shown in the upper left of the AZTEC platform. For more details on service assembly and the splicing technique used, readers are referred to [9].

The process schema assembly performed by the Assembly component is traditionally viewed as a "design-time" activity. In contrast, the **Execution component** in the bottom center of the AZTEC platform is charged with the "run-time" activity of executing the process schemas. This execution engine is event driven. It interprets the process schemas, and interacts with web-services, telephony services, and wireless services through a collection of **Wrappers** (provided as part of this component) and gateways (provided by emerging products from telephony equipment manufacturers including Lucent).

The final component of AZTEC is for (dynamic) **Schema Management**. This component is used to load process schemas into the Execution component, and more importantly, to enable modifications and refinements (via the Assembly component) to process schemas in the middle of executing on a process schema. In particular, this supports the "design-time" activity of process schema assembly in the midst of "run-time" activity of process schema execution.

As with database mediators [23], humans will have an important role in the creation, maintenance, and monitoring of a running AZTEC platform. Through the **Administration component**, programmers will develop the templates and fillers that are used to support the hierarchical planning process for automated service assembly. Programmers will also specify the policies to be used when building (or modifying) process schemas for satisfying requests for composite services. Finally, humans may choose to monitor the status of service executions, and perhaps directly manipulate the process schema of a running composite service.

4.2 Execution of XASC process schemas

We now describe how an XASC process schema is executed by the Execution component of the AZTEC platform. A process enactment gets started by an *init* event generated by the system. Referring to Figure 2, this event is received by the **Event Listener** and passed - through the **Event Queue** - to the **Task Dispatcher**. The Task Dispatcher then assigns a thread from its Thread Pool to initiate the process enactment. This is a predefined procedure for all process enactments, comprising the generation of a unique process identifier for this enactment, the initialization of a **Context Repository** document, which will be used for data passing across flowcharts within this process enactments and, finally, enacting the root flowchart of this process schema. This enactment is represented internally as a DOM tree and placed into the working space of the **Flowchart Logic Server**. Note that due to the structured nature

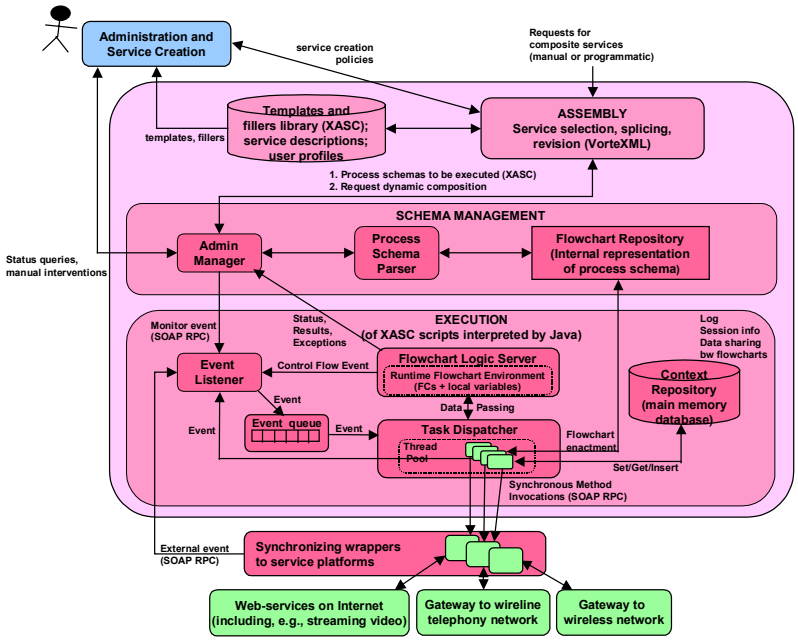


Fig. 2. AZTEC Framework for assembly and execution of session-oriented e-services

of the flowcharts, the order of execution of tasks is equivalent to a depth-first traversal of this tree representation. At this point, the flowchart is ready for execution.

More generally, in the middle of executing a process enactment the Flowchart Logic Server will be executing zero or more flowcharts in parallel. In our running example, in the middle of the teleconference there might be two flowchart enactments running to handle participants who are attempting to join the teleconference and another that deals with a participant who has just hang-up his telephone. Note also that there might be more than one process enactment running at the same time; in this case, however, flowcharts from different process enactments are not allowed to interact with each other - sharing data through different Context Repository documents and having different process identifiers.

For each one of these flowcharts, the Flowchart Logic Server maintains information about their execution state (e.g., which task is currently running) which can also be queried through internal functions. Then, for instance, whenever a task has finished the Flowchart Logic Server will generate a Control Flow Event which - when dispatched – will result to the execution of the appropriate (i.e., the next one in a sequence) task, and send it to the Event Listener module, which will place it in the Event Queue. Finally, the Task Dispatcher has the role of invoking individual tasks in

response to events that it gets from the Event Queue. In order to allow for several tasks to be executed concurrently, the invocation of the tasks itself is handled by a set of threads (Thread Pool) coordinated by the Task Dispatcher. When capable of doing more work, i.e. when there are available threads in its Thread Pool, the Task Dispatcher will ask the Event Queue for an event. The Task Dispatcher may also decide to expand its Thread Pool if no threads are available, if, for instance, most of the threads are waiting for results and do not consume much of their CPU time.

A key feature of AZTEC in the execution of multiple flowcharts is the rich flexibility that is given for deciding the priorities with which the steps of different flowcharts should be executed. In AZTEC the prioritization is managed explicitly by separating the Flowchart Logic Server, the Event Queue and the Task Dispatcher. Clearly, when handing the next event to the Task Dispatcher for execution, the Event Queue can easily employ a scheduling algorithm to pick the event with the higher priority. Thus, events with a higher priority are processed sooner that events with a lower priority. Events with the same priority are processed in a FIFO order.

The individual tasks will primarily be function calls to wrapped e-services that are resident on the web, the telephony network, or the wireless network. In cases when a service will need to send an asynchronous event back to AZTEC (i.e. when a participant has hang-up), this event will be trapped by the wrapper. The wrappers can fill-in application specific information and send the event into the system through the Event Listener, in an appropriate internal representation. When a response is obtained, the Flowchart Logic Server is informed that the task has completed and also receives the returned values of the task (if any).

Finally, this architecture facilitates explicit control on the execution state of flowcharts. For instance, suspending a flowchart is be implemented by just *annotating* the flowchart instance in the Flowchart Logic Server's internal working space as *suspended*. As long as a flowchart is marked as suspended, the Flowchart Logic Server does not send any more events for the execution of its tasks. Moreover, when the Event Queue is asked to give the highest priority event to the Task Dispatcher, it can bypass events that have been produced by flowcharts, which are marked as *suspended*. For tasks of a flowchart which are already running, when the *suspend* command arrives, we allow them to finish their execution normally, in order to avoid inconsistencies.

4.3 Loading and modifying process schemas

We now turn to the Schema Management component of AZTEC, which supports the delivery of process schemas into the Execution component, and dealing with dynamic process schema modification and refinement. If a new process schema is created by the Assembly component, then it can be passed into the Admin Manager. This in turn passes the schema to the Process Schema Parser, which parses the XASC specification into an internal DOM representation. If the parsing is successful then the result is placed into the Flowchart Repository, otherwise, an error message is sent back to the Admin Server.

In some cases the Assembly component will provide an incompletely specified process schema to the Execution component. For example, the particular choice of an e-service might be omitted, or in fact a larger portion of a flowchart might be left

unspecified. This will permit dynamic selection of e-services and/or refinement of the process schema. How do the empty slots get filled in? The basic approach is that the Flowchart Logic Server will come to a point of the flowchart that is unspecified and notify the Admin Server. The Admin Server in turn will gather appropriate information (e.g., about the current sessions and states of active flowchart instances) and send a request to the Assembly component to fill in the needed parts of the process schema. When this comes back, the Admin Manager gives it to the Process Schema Parser, which in turn produces an internal representation for the flowchart, including a marker indicating where processing should start. Finally, the Admin Manager will send an event into the Event Scheduler indicating that the process schema has been refined and is ready to go.

It should be stressed that in order for the Admin Manager to access the Context Repository, it must go through the Event Scheduler and Task Dispatcher. This design decision is motivated primarily to keep the number of data and control flow paths to a minimum. The Assembly component might also get involved with schema execution if there is a significant exception to a running process schema. In this case, the exception can be passed to the Admin Manager, which can request the Assembly component to create a repair. The Admin Manager is also involved if a human, through the Administration and Service Creation component, wants to examine the runtime status of an XASC process enactment.

5 Related Work

Several proposals for workflow systems attempt to combine the technology of active database systems with event-based systems. Some examples of event-driven distributed workflow engines are WIDE [5] and EVE [13]. The main modules in EVE are: an event detection and logging module, rule execution module and service execution module. Upon detection of a primitive or composite event, this module will activate the appropriate rule in the rule execution module. The latter consists of ECA rules. When a rule is activated, it will check the associated condition, and if it is true, the corresponding action is performed. The action part of the rule will in turn generate events, which are fed into the event detection module. In comparison with AZTEC, EVE has more sophisticated support for events; however, EVE does not have a notion of sessions like AZTEC. Moreover, the action part of the ECA rules in EVE cannot mimic all of the flexibility that the AZTEC flowcharts offer.

Citation [19] describes an event based approach for dynamic modifications of running workflow instances using rules and predicates such as drop, replace, check, delay and process. This approach is especially useful for semantic exception handling. The AZTEC model can also support adaptability in a somewhat different way by changing the contents of the context repository during a running instance. Another interesting service-oriented model for inter-organizational workflows is given by the Crossflow project [16]. This model provides mechanisms for selection and invocation of services, and controlling and monitoring an external service. However, all these models fail to recognize the notion of a session, the only exception being the Caltech Infospheres project [8], where sessions are entities that can be explicitly specified, supported and reasoned with.

6 Future Work

We foresee several areas of further research in connection with the AZTEC framework. One such area relates to further refinement of the active flowchart model, that supports hierarchical and modular constructs for specifying active flowcharts and their interactions, including priorities and data sharing. It is also necessary to perform some kind of global consistency checking for all the flowcharts in an application to ensure that their interactions are "safe". Another research area is performance, since telecommunications applications typically require sub-second responses to most events (such as when a participant is dropped). Hence, we plan to model the performance implications of our architecture in order to ensure that it can meet stringent performance goals. Since multiple flowcharts can run in parallel and generate events simultaneously, priority assignment policies must be compatible with the real-time performance requirements. Although our discussion of the framework has focused on the running of a single enactment (e.g., to control a single multimedia teleconference), in practice the execution engine must support at least hundreds of enactments running at the same time. In such a context, it is important to study issues resulting from the interactions between concurrently running enactments. There are also performance issues related to assignment of priorities to enactments and scheduling of enactments. Finally, it is important to support recovery in our framework so that session-oriented composite e-services can be recovered from failures.

REFERENCES

1. W.M.P. van der Aalst and A. Kumar, "XML Based Schema Definition for Support of Inter-organizational Workflow", Information Systems Research (accepted).
2. V. Anupam, R. Hull and B. Kumar, "Personalizing E-commerce Applications with On-line Heuristic Decision Making ", *Proceedings of Tenth Intl. World Wide Web Conference*, June 2001.
3. B. Benatallah, B. Medjahed, A. Bouguettaya, A. Elmagarmid and J. Beard, "Self-Coordinated and Self-Traced Composite Services with Dynamic Provider Selection", Technical Report, University of New South Wales, March 2001 (Available at http://sky.fit.qut.edu.au/ ~dumas/selfserv.ps.gz).
4. R. Breite, P. Walden and H. Vanharanta, "C-Commerce Virtuality – Will it work in the Internet?", Proc. of International Conf on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet (SSGRR 2000), http://www.ssgrr.it/en/ssgrr2000/proceedings.htm.
5. F. Casati, P. Grefen, B. Pernici, G. Pozzi, and G. Sanchez. WIDE workflow model and architecture. Technical report, University of Twente, 1996.
6. F. Casati, S. Sayal, M. Shan, "Developing e-services for composing e-services", *Proceedings of CAISE 2001*, Interlaken, Switzerland, June 2001.
7. F. Casati and M. Shan, "Dynamic and adaptive composition of e-services", *Information Systems*, to appear 2001.
8. K. Mani Chandy and Adam Rifkin, "Systematic Composition of Objects in Distributed Internet Applications: Processes And Sessions", Conference

Proceedings of the Thirtieth Hawaii International Conference on System Sciences (HICSS), Maui, Volume 1, January 1997, pp.395-404.
9. V. Christophides, R. Hull, A. Kumar, J. Simeon, "Workflow mediation using VorteXML, " *IEEE Data Engineering Bulletin* 24(1), March 2001, 40-45.
10. V. Christophides, R. Hull, A. Kumar, "Querying and Splicing of Workflows," *CoopIS '02*, September 2001 (forthcoming).
11. K. Erol, J. Hendler, and D. Nau, "Semantics for hierarchical task network planning", Tech. Report CSTR3239, CS Department, Univ. of Maryland, 1994.
12. P. Fankhauser, M. Fernandez, A. Malhotra, et al., "The XML Query Algebra ", W3C Working Draft, 15 February 2001, http://www.w3.org/TR/query-algebra/.
13. A. Geppert and D. Tombros, "Event-based distributed workflow execution with EVE," Technical Report Technical Report 96.5, University of Zurich, 1996.
14. R. Hull, F. Llirbat, E. Simon, et al., "Declarative Workflows that Support Easy Modification and Dynamic Browsing" *Conference on Work Activities Coordination and Collaboration (WACC)*, San Francisco, February 1999, 69-78.
15. B. Kiepuszewski, A. ter Hofstede and C. Bussler, "On Structured Workflow Modelling", *Proceedings of CAISE 2000*, Stockholm, Sweden.
16. J. Klingemann, J. Wasch and K. Aberer, "Adaptive Outsourcing in Cross-organizational Workflows, GMD Report 30, August 1998.
17. G. Kuper and J. Siméon, Subsumption for XML Types, International Conference on Database Theory (ICDT'2001), January 2001,London, UK.
18. S. A. McIlraith, T. Cao Son, and H. Zeng, "Semantic Web Services", *IEEE Intelligent Systems*, March/April – 2001.
19. R. Muller and E. Rahm, E., "Rule-Based Dynamic Modification of Workflows in a Medical Domain," Proc. Datenbanksysteme in Bro, Technik und Wissenschaft (BTW '99), Freiburg, March 1999, pp. 429-448.
20. R. Reiter, "KNOWLEDGE IN ACTION: Logical Foundations for Describing and Implementing Dynamical Systems". Book in preparation. http://www.cs.toronto.edu/cogrobo/.
21. Simple Object Access Protocol (SOAP) 1.1, W3C Note 08, May 2000, http://www.w3.org/TR/SOAP/.
22. H.S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. "XML schema part 1: Structures", W3C Recommendation, October, 2000.
23. G. Wiederhold, "Mediators in the Architecture of Future Information Systems" *IEEE Computer*, Volume 25, Number 3, 1992, 38-49.
24. G. Weikum, (Special Issue Editor), *Bulletin of the Technical Committee on Data Engineering*, IEEE Computer Society, Vol. 24, No.1, March 2001.
25. Widom J. and, Ceri S. (Eds.). Active Database Systems: Triggers and Rules For Advanced Database Processing. Morgan Kaufmann, 1996.