XTML:
Extensible Telephony Markup Language

A Unified Framework for Delivering Next-Generation Enhanced Telecommunications Services

Table of Contents

1 Introduction

This document introduces XTML, the Extensible Telephony Markup Language, which is an XML-based service description language and associated service execution framework that is designed to provide a unified approach for the delivery of next-generation enhanced telecommunications services. The goal of this document is to describe the goals of the XTML architecture, to describe the architecture itself, and to define the requirements that must be met by conforming XTML applications and application servers. This document is intended to provide only a high-level overview of XTML. For more detailed information, please refer to the documents listed in the "Related Documents" section.

A basic understanding of XML, the Extensible Markup Language, is assumed on the part of the reader. For those without such an understanding, the following brief description will have to suffice:

XML is a tag-based language that provides a structured way organizing information in the form of documents. It's syntax is roughly similar to HTML, although HTML is not an XML-conformant language. XML is a derivative of SGML, Simple Generalized Markup Language. An XML document organizes information into a hierarchical set of information elements. An information element may contain nested sub-elements, and it may contain attributes and pure text as well. XML has proven its value as a standard way of transferring structured information between systems. Further information on XML is available at www.xml.org.

2 XTML Conceptual Overview

This section describes the driving forces behind the development of XTML, its design goals, its relationship to alternative XML-based telephony-oriented vocabularies, and the overriding architectural model that XTML services execute within.

2.1 The Fall and Rise of Enhanced Telecommunication Services

Historically, the market for carrier-class enhanced telephony services has not been a success. As measured against the demand for services, market growth rates have been moderate, the number of vendors providing solutions has been small, and the cost of creating and delivering solutions has been high. Most vendors have provided closed, proprietary, and non-interoperable solutions. This has fragmented the market and reduced the choice that is available to service providers. It has also meant that service providers have faced a range of problems:

- Increased maintenance costs due to supporting multiple, incompatible enhanced services platforms,
- Increased risk due to unavoidable vendor lock-in, and
- Reduced quality due to lack of competition and poor integration of services across platforms.

The end result is has been slower-than-expected market growth rates, and a paucity of enhanced services delivered to end-user subscribers.

Standing in stark contrast to this is the rapid growth of the Internet services market over the past several years. There are many causes of the "Internet explosion" phenomenon, but one of the key factors driving the growth of the market has been standardization on HTML as the common service description language and HTTP as a common communications protocol. Standardizing on these two protocols has allowed a mass market to form for Internet-enabled software and services. The benefits to consumers have been significant:

- Consumers have much greater choice and reduced risk of vendor lock-in.
- Consumers can select best of breed products from rather than relying on a single vendor. For example, consumers can purchase an HTML browser, a web server, an application server, and web applications each from a different vendors, choosing the best product in each category rather than relying on a single-vendor solution from end to end.

The Internet architecture also balances the competing needs for stable, managed evolution of standards with the desire for innovation and product improvement. In order for a mass market to form around a technology standard, the technology specification cannot be a moving target—it must be stable enough that vendors can build products to meet the specification and customers can understand how to assemble complementary products that interoperate per the standard.

On the other hand, vendors need the ability to innovate in order to improve their products and drive long-term market growth. The Internet software architecture balances these two needs quite effectively. The HTML/HTTP standards are stable, carefully managed standards that allow web server and browser vendors to build interoperable products. On the back-end, however, application server processes are extensible via component architectures such as COM, Enterprise Java Beans, and Corba to support the addition of new features and capabilities into the overall architecture without fragmenting the market. The result is a mass market that offers consumers optimum choice while enabling rapid ongoing product innovation.

It's easy to conclude that a similar standardization of frameworks and tools in the telecommunications space would bring similar results, but unfortunately the situation is not that simple. The telecommunication space provides a more challenging problem to solve. Whereas the internet space has benefited from the standardization of a single subscriber interface—the html-based browser—and a single communication protocol—http--standardization at this level is not possible in the telecommunications arena. Telecommunications solutions must allow subscribers to communicate over an ever-expanding variety of physical and network interfaces (landline, cell phone, pager, browser, Palm Pilot, etc),. For the foreseeable

future, enhanced services platforms will need to be able to "speak" a variety protocols and support a burgeoning portfolio of services.

In such an environment, open service delivery frameworks are a necessary but not a sufficient condition for success. In addition to openness, these service delivery frameworks must be extensible, in order to support the increasing variety of networks and interfaces required. An open, but static, service description language and delivery framework will never be able to satisfy the market demand, because the capabilities of emerging technologies will quickly outstrip the ability of a static language to describe them.

XTML is designed to solve this problem. XTML is an open, extensible service description language and execution framework that provides a standard way of describing and delivering next-generation, network-based communications services. It provides a standard, XML-based core language that all service developers can use to describe any communication service running over any network. It provides the open extensibility features required that allow vendors or groups to extend the language to describe the capabilities of new technologies, protocols or interfaces in agreed-upon ways. These innovations can then be deployed on any application server that conforms to the XTML interface specification.

The goal of XTML is to define an open, unified, and extensible enhanced services architecture that can be broadly adopted by vendors in the enhanced services market, thereby accelerating market growth and delivering significantly increased value to service providers:

- With XTML, service providers are not locked into a single-vendor solution. With an open XTML standard, a service provider could buy an XTML application server from one vendor, enhanced service applications from a second vendor, and an XTML development environment from a third.
- With XTML, solution providers have a much larger market to sell to. A solution provider that develops a compelling new application can deliver it to a broader market of open platforms rather than to a single, proprietary platform. Similarly, a solution provider that develops an XTML server extension can deliver it to any conforming XTML application server.

In short, XTML is designed to bring the power of Internet-style market forces to the enhanced telecommunications market. The standardization of a common service description and delivery framework that can meet all needs and support product innovation will revolutionize the enhanced services market and push it to the next stage of growth.

2.2 Comparison to Related XML-Based Vocabularies

Several efforts have already been made to define XML vocabularies for telephony-related application domains. Most notable among these vocabularies are VoiceXML, a vocabulary for describing speech-enabled interactive voice response dialogs, and the IETF's IPTEL Call Processing Language (CPL), a vocabulary aimed at describing Internet telephony services. This section of the document provides a brief overview of each of these XML vocabularies and describes why they are not suitable for solving the service provider challenge identified earlier.

VoiceXML is more narrowly focused on the problem domain of voice response applications. Its goal primarily is to bring internet-style tools to bear on the challenge of providing speech-enabled access to information content. It has limited ability to be extended to provide additional functionality outside of this target application domain. In particular, it does not provide the type of third-party call control facilities needed to build applications like calling services, conferencing, or IP Centrex. It therefore addresses only a small percentage of the next-generation communications services that service providers are looking to deliver.

The IETF's proposed call processing language framework (CPL) is focused on the problem domain of describing call control applications within internet telephony networks. As described in the related IETF drafts, CPL has a heavy bias towards SIP (Session Invitation Protocol)-based call control signaling. While SIP is an important emerging standard for IP telephony, it represents only one of many signaling protocols that carriers will need to support in convergent networks. As currently defined, CPL can not be used to describe other services that use other call control protocols, such as MGCP (Media Gateway Control Protocol), SS7, and H.323. Since CPL is focused purely on call control applications, and provides no

extensibility features, it also cannot be used to describe other types of applications, such as interactive voice response or messaging services.

Furthermore, both VoiceXML and CPL share a major underlying design flaw. That flaw stems from the approach of attempting to define a static language to describe a very dynamic problem domain. To be an effective form of expression, language must evolve as quickly as the problem domain it is attempting to describe. As a practical point, that is why in the real world different groups of people facing different day-to-day realities develop different dialects, even within a common language. In the problem domain of next-generation communication services operating over convergent networks, the pace of change is incredibly rapid. New protocols are being defined, new forms of technologies are emerging, and new ideas for services are being generated, all at a rapid-fire rate. A committee-based process for authorizing language enhancements would never be able to keep pace.

XTML takes a different approach for defining language. It explicitly recognizes the need to define a core language, while allowing informal "dialects" or extensions to be dynamically defined in response to the changing problem domain. It clearly defines and separates the aspects of service description that are relatively static and common to all network-based personal communication services, from the more dynamic aspects that correspond to specific technologies, protocols, or application domains. It allows third parties to fundamentally extend the dynamic aspects of the language, while ensuring that these extensions are fully described and can be implemented on any open XTML server.

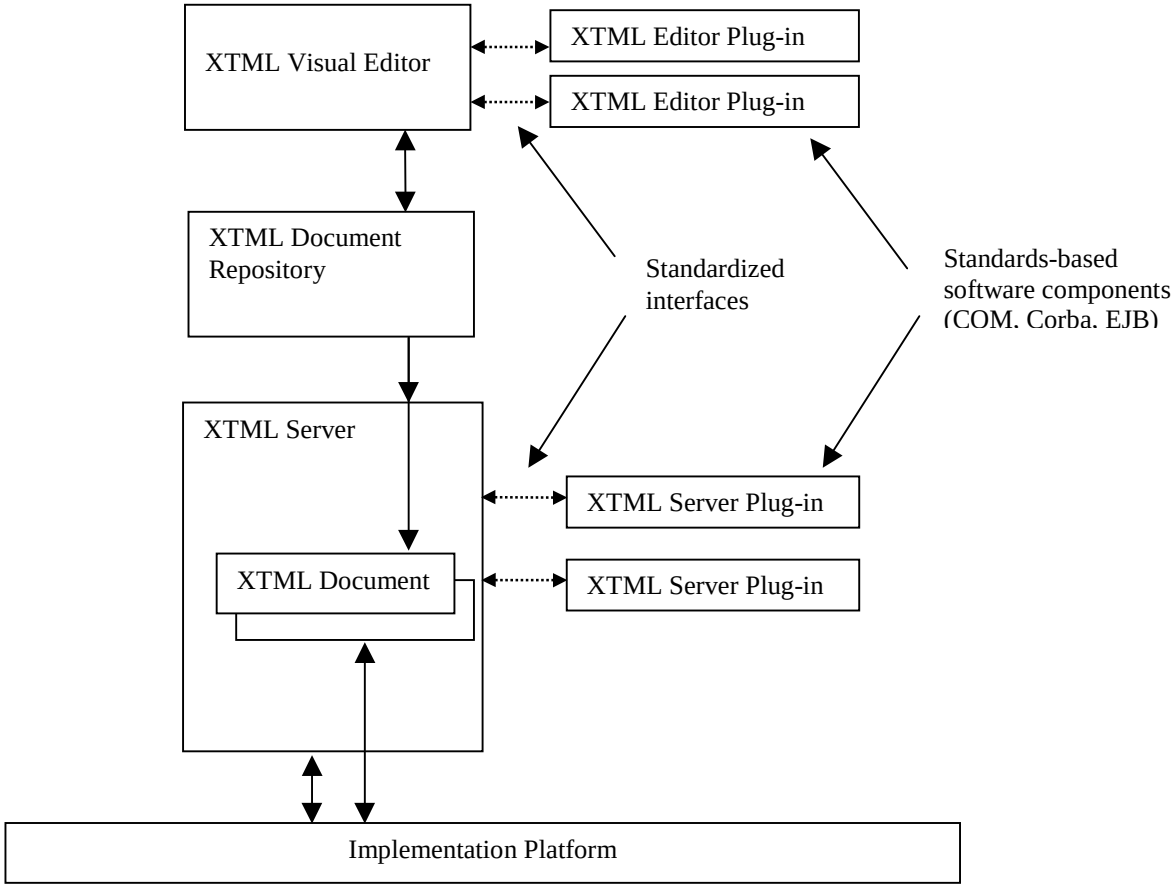
XTML therefore addresses the service provider challenge in a way that VoiceXML and CPL can not:

- it supports the full range of services that providers wish to offer,
- it enables service providers a choice of open service delivery platforms, and
- it can be easily extended to provide new features and leverage new technologies and market opportunities.

2.3 XTML Architectural Model

XTML itself is simply a vocabulary for describing services. In order for these services to be executed, they must exist within a service delivery framework. In order to support the extensibility features of the XTML language, XTML poses some requirements for this service delivery framework. These requirements can be described in terms of an XTML architectural model, which is pictured below:

Figure 1: XTML Architectural Model

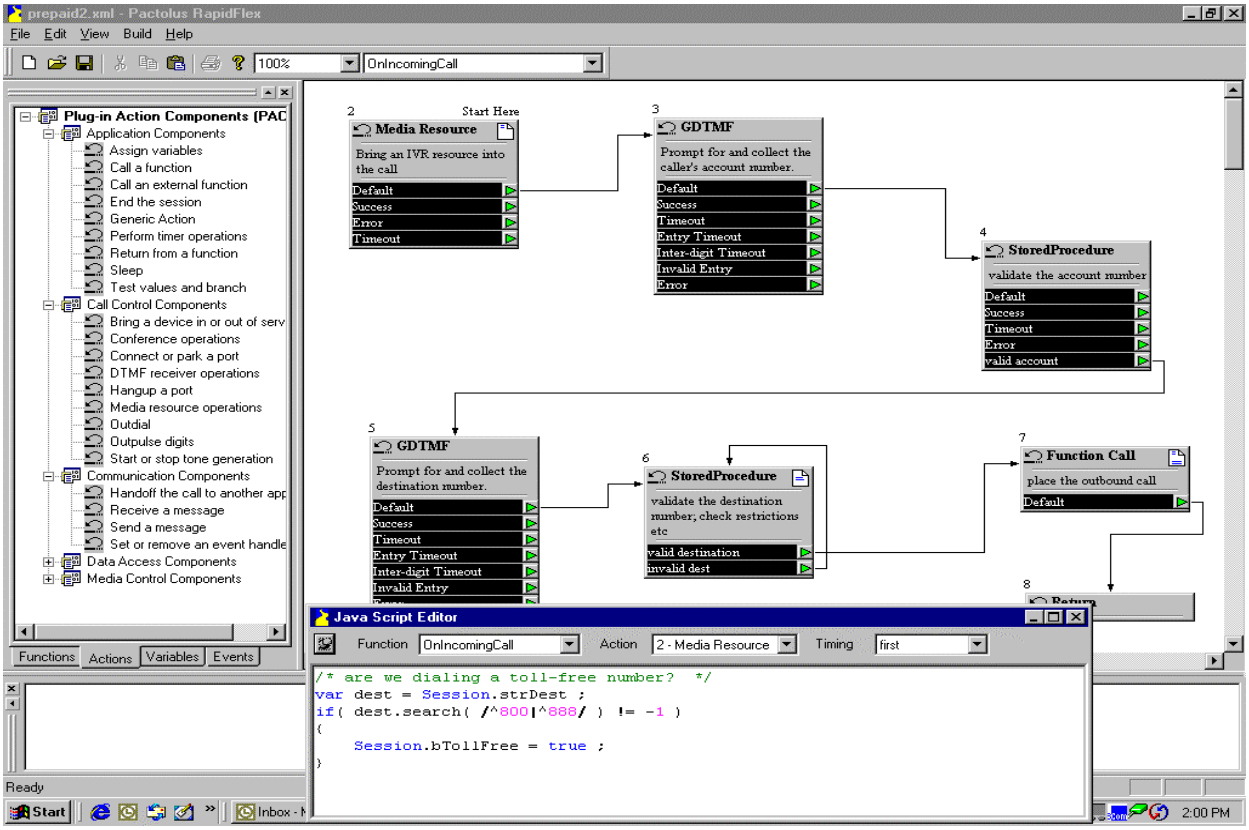


An *XTML Visual Editor* is graphical design tool for creating XTML documents. Since XTML is an XML-based language, standard XML browsers or even text editors can be used to create XTML documents. However, a visual design tool can be a significant aid in creating complex call-flow applications that involve a large amount of state machine-type logic and conditional branching. An XTML Visual Editor is a graphical tool that allows a designer to drag, drop, and configure service building blocks into a flowchart-like visual representation of the service, making it much easier to quickly create and modify services. Since the XTML Visual Editor generates and loads XTML syntax, services can be designed in an XTML editor and then modified in a pure XML editor or text editor, or vice versa. An example screenshot from Pactolus' XTML visual editor is displayed on the next page.

Because the XTML language is extensible and allows third parties to add new service description actions and events, XTML Editors (as well as XTML Servers) must be extensible as well, in order to support the configuration and execution of these new language constructs. That is, in order to make use of an XTML language extension, a designer must be able to configure information about the extension at design time, and the XTML server must be able to execute the service logic associated with the language extension at runtime. Thus, the XTML architectural model specifies a component-based architecture. As the language

is extended to incorporate new actions, XTML editors and servers can be similarly extended via a component architecture so that services can be designed and executed using these new language extensions. The XTML architectural model does not however mandate the use of any particular component framework; a variety of component solutions are available, including COM, CORBA, and Java/Enterprise Java Beans (EJB). In addition, Pactolus has developed an lightweight Portable Component Architecture (PCA) that allows high-performance XTML server extensions to be developed in pure C++ and deployed to both UNIX and Windows NT XTML application servers without requiring additional third-party software.

Figure 1: Sample Screenshot from Visual XTML Editor



An *XTML Document* is a text file containing the description of service using the XTML vocabulary. These documents are generated using XTML Visual Editors or XML editors, and are stored in an *XTML Document Repository*, which is typically accessed via a web server.

An *XTML Server* retrieves XTML documents from the XTML document repository and executes the service logic described by the document. The XTML Server provides concurrency support, allowing a single service simultaneously on behalf of multiple independent subscribers as well executing multiple different services concurrently. XTML servers are designed to be able to handle thousands of sessions executing simultaneously.

XTML Server Plug-ins are independent software components containing domain-specific service logic that extend the XTML server to support new XTML language extensions and features. An XTML document references the XTML language extensions it uses, and the XTML Server loads and initializes the plug-ins corresponding to these extensions when the XTML service is loaded.

The *Implementation Platform* provides the interface to the outside world, in terms of the specific hardware devices and software platforms that running services interact with and make use of. The implementation platform is controlled by the XTML server, which acts on behalf of the running services that it manages. The implementation platform also generates events in response to changes in the outside world. These

events are handled by the XTML server and routed to the specific service and subscriber instance that needs to know about this event.

As an example of the interaction between an XTML server and an implementation platform, the XTML server might command the implementation platform to perform an outdial in response to a service request for a specific subscriber. Once the outdial is complete, the implementation platform will generate a response to the XTML server indicating the success or failure of the request. Later, the implementation platform might asynchronously send an event to the XTML server indicating that a hangup event has been detected on a specific telephony port. The XTML server would route this event to the particular session of the running service which owns that port, which in turn would execute the service logic that the associated XTML document has defined for this event.

3 XTML Language Overview

This section provides a high-level overview of the XTML language syntax. It reviews the conceptual underpinning of the language, describes the distinction between the core language schema and extensions, and shows a simple example.

3.1 The XTML Philosophy

The XTML language specification consists of a base schema plus any number of extension schemas. The XTML base schema focuses on the service-independent aspects of the language—constructs such as services, sessions, variables, functions, events, and transitions. These constructs will be a part of every service, because they are the fundamental building actions that are required to describe any finite state machine. Since they are service-independent, they are therefore relatively static.

However, many of the other constructs that appear in a service description tend to be the service-specific or application domain-specific actions and events required to implement the service (e.g., send/receive a message, proxy or redirect a sip request, write a billing record to a database, locate a subscriber, send a screen pop to an operator workstation, create an audio or video conference, etc). These constructs tend to be dynamic, because they change as rapidly as application domains, market requirements, and enabling technologies change.

In a dynamic, multi-application environment it would be foolish for one organization to try to define a static language that can describe all possible services that a service provider might want to deploy. Such a language would never be broad enough to describe all possible services that providers will want to deliver, and it would also quickly become outdated shortly after it was released, due to the ongoing pace of change in the market. Instead, the XTML approach is to define a core language that is relatively static and common to all services, and then supply a mechanism for vendors to augment the language via domain-specific extensions that can be described and delivered in such a way that these solutions can made broadly available to the market. This will allow groups of service and solution providers that are focused on a specific application domain to work together to more rapidly evolve the functionality in that area, while ensuring that the resulting solutions are available to the broader market and interoperable with each other.

To summarize, the XTML philosophy is:

- *Capture the service-independent language concepts in a base schema that can be widely agreed upon and managed through an open standards process.*
- *Allow domain-specific language extensions to be developed independently but linked explicitly into XTML documents through language features in the base schema.*
- *Specify the interface between an XTML application server and the plug-in software components, thereby allowing support for new language extensions to be dynamically added to any conforming XTML application server.*

3.2 XTML Base Schema Overview

An XTML document describes a next-generation enhanced communication service. Such a service fundamentally consists of a series of interactions between subscribers, network and computing devices, and information content. These interactions are governed by the state machine logic that is defined in an XTML document. The major constructs of the XTML language, therefore, are the elements that are needed to describe state information, actions, events, and transitions in a highly concurrent and asynchronous environment. These concepts are covered below.

3.2.1 Services

A *Service* is a pre-programmed series of interactions between subscribers, network and computing devices, and information content. Services are described using XTML and are executed by an XTML Server

operating in conjunction with an implementation platform. The terms “service” and “application” are used interchangeably within this document.

3.2.2 Sessions

A *Session* is a single, independent, logical thread of execution of a specific service that executes within an XTML Server. It is independent in the sense that each session maintains its own state information and execution context. Typically, a session executes on behalf of a specific subscriber. For example, if an XTML Server was concurrently running 1,000 instances of a prepaid calling card application, each session could be handling a different subscriber, could have different values for subscriber account information, and could be at a different point in the prepaid call flow state machine. The fundamental responsibility of the XTML server is to create, manage, and terminate sessions.

3.2.3 Functions

The service logic that describes a service can be modularized into *Functions* within an XTML document. Functions provide all of the familiar constructs of 3GL programming languages: the ability to centralize and reuse standard logic, the ability to pass parameters by value or reference, the ability to return results, and so forth.

Example XTML Syntax:

```
<function name="OnIncomingCall" start=”1” returns="void">
```

3.2.4 Events

Events are generated by the implementation platform, or by the XTML server itself, and are handled by sessions. Event support is necessary to develop services that are designed for the asynchronous nature of telecommunications applications. The XTML server dispatches events to specific sessions for event-handling logic to be performed. An XTML document specifies the events that it is prepared to handle and indicates for each one a function that will act as the event handler. When an event is received (or generated) by the XTML server, it interrupts the logical thread of execution of the session receiving the event and calls the event handler function for that session. Once the event handler function has returned, the previous thread of execution is resumed. An XTML document can also specify that events should be disabled during specific states of the call flow. In this situation, events will be queued by the XTML server until the session re-enables event handling, at which points the queued events will be dispatched to the session for handling.

Example XTML Syntax:

```
<events>
        <event name="Pactolus.OnIncomingCall.1" handler="OnIncomingCall" />
</events>
```

3.2.5 Variables

An XTML document defines *Variables* that are used to maintain service-specific state information for a running session. A broad range of variable types are supported, including strings, integers, floating points, booleans, dates, arrays, and structures. Variables can be scoped at 3 levels: local variables are defined within a function and scoped to that function, global variables are accessible from any function within a session, and shared variables are accessible from any session within a given service. The distinction between global and shared variables is that each session has its own independent copy of global variable values, while there is only a single copy of shared variable values that are shared by all running sessions. Shared variables are useful for maintaining information that is maintained for all running sessions, such as configuration-related information (e.g., the default trunk group to outdial on after hours, the SIP server to proxy for certain types of requests, etc).

Example XTML Syntax:

```
<vars>
        <var name="nPromptID" type="i2">1001</var>
</vars>
```

3.2.6 Actions

An *Action* is a large-grained instruction that is issued by a running session. A function consists of a set of actions that are executed in the sequence specified by the XTML document. The service logic for each of these actions is not contained in the XTML server itself, but is provided via XTML server plug-ins. This enables new instructions to be easily added to an XTML server without requiring modifications or upgrades of the XTML server itself.

The action element provides the critical link between the base schema and extension schemas. The plug-in attribute of the action element specifies the run-time component that the XTML server should load to perform this action. The action element has an open XML model, so that additional elements that are not part of the base schema are allowed to appear as child elements of the action element. It is the plug-in's responsibility to parse any additional XTML language extension elements at service load time, and carry out the action-specific logic at service execution time.

Example XTML Syntax:

```
<action id="1" plug-in="Vendor.Announce.1">
        <announcements xmlns="x-schema:http://www.vendor.com/my-extensions.xml>
                <announcement>nPromptID</announcement>
        </announcements>
        <results>
                <result name="Next" link="2" />
        </results>
</action>
```

3.2.7 Results

The execution of an action has a *Result*, or outcome, which determines the transition to the next action. Results are identified by symbolic name (e.g., "Success"), and each type of action can have a unique set of possible results. For example, a Call action might have results such as "Success", "Busy", "No Answer", etc. In addition to the predefined results for a given action, service designers can add custom results at design time by building a conditional statement to be evaluated at run time (e.g., "if retries > 3 then result = 'Hang up'"). See above for an example.

3.2.8 Links

A *Link* represents a unidirectional path between two actions that is taken when a specific result occurs as the outcome of the first action. A link therefore defines a transition in the state machine logic described by an XTML document. See above for an example.

3.2.9 Scripts

A *Script* is a programming script that is be executed at runtime by the XTML server. Javascript is currently the only scripting language supported, although others are possible in the future. Scripts can be declared globally, or as part of an action. A global script typically contains function definitions that are called from action scripts. Action scripts are executed in concert with a specific action plug-in.

A script can be executed either just before the action plug-in (timing="first"), just after the action plug-in is executed but before custom results are evaluated (timing="middle"), or after the custom results are evaluated but before transitioning to the next action (timing="last"). It is acceptable to have up to three different action scripts associated with a single action, each executed with a different timing sequence selection.

The XTML application server exposes several predefined objects that scripts can use to interact with the running service. The 'Session' object allows the script to access or update any variables that have been

defined in the XTML document by referring to them as an attribute of the Session object (i.e., Session.variable). The 'Service' object allows the script to access information about the service, such as the service name and the current number of running sessions. Finally, the 'Server' object allows the script to retrieve information about the XTML application server, as well as perform actions such as writing information to the system log, etc. An example of a simple action script might be as follows:

Example XTML Syntax:

```
<action id="1" plug-in="Vendor.CreateConnection.1">
<script language=”Javascript” timing=”middle”>
<![CDATA[
if( ++Session.retries > 3 )
{
     Session.bForwardToOperator=true;
}
]]>
</script>
</action>
```

In addition to these elements, XTML documents can also contain XML processing instructions for the XTML editor. Processing instructions are an XML construct that are used to pass information that is unrelated to the content of an XML document through to an XML application that is processing the document. Two processing instructions are supported: one that informs the XTML editor where to visually position a action within a flowchart-like representation of a function, and one that indicates that a specific result should be hidden in the editor. Examples of processing instructions are shown below:

```
<!—position the announce action at coordinates (47,33) in the editor -->
<action id="1" plug-in="Pactolus.Announce.1">
       <?xtml-editor x=47 y=33 ?>

<!—show the “User Located” result but hide the “User Not Found” result-->
<results>
    <result name="User Located" link="2" />
    <result name="User Not Found" />
       <?xtml-editor visible=”no” ?>
</results>
```

3.3 Simple XTML Example

With the preceding information as background, we can now illustrate XTML syntax using a simple example. The XTML document below describes a service that answers an incoming call, plays an announcement, and then hangs up the call.

Figure 2 Simple XTML Service – Answer, Play Announcement, Hangup

```
1   <?xml version="1.0" ?>
2       <xtml name= "announce" xmlns="x-schema:http://www.pactolus.com/xtml-base-schema.xml">
3           <events>
4               <event name="Pactolus.OnIncomingCall.1" handler="OnIncomingCall" />
5           </events>
6           <functions>
7               <function name="OnIncomingCall" start="1" returns="void">
8                   <parameters>
9                       <parameter name="port_number" type="ui4"/>
10                      <parameter name="dnis" type="string"/>
11                      <parameter name="cli" type="string"/>
12                  </parameters>
13                  <local-vars>
14                      <var name="nPromptID" type="i2">1001</var>
15                  </local-vars>
15                  <actions>
16                      <action id="1" plug-in="Pactolus.Announce.1">
17                          <?xtml-editor x=47 y=33 ?>
18                          <announcements xmlns="x-schema:http://www.pactolus.com/ps-call-extensions.xml>
19                              <announcement>nPromptID</announcement>
20                          </announcements>
21                          <results>
22                              <result name="Next" link="2" />
23                          </results>
24                      </action>
25                      <action id="2" plug-in="Pactolus.Hangup.1">
26                          <?xtml-editor x=103 y=42 ?>
27                          <hangup xmlns="x-schema:http://www.pactolus.com/ps-call-extensions.xml
28                              port="port_number" />
29                          <results>
30                              <result name="Success" link="3" />
31                              <result name="Failure" link="3" />
32                          </results>
33                      </action>
34                      <action id="3" plug-in="Standard.EndSession.1">
35                          <?xtml-editor x=260 y=66 ?>
36                      </action>
37                  </actions>
38              </function>
39          </functions>
40      </xtml>
```

The line numbers displayed above are not part of the XTML document, but are shown here to help illustrate some of key features of XTML, described in the table that follows.

Line	Comment
2	Declares the root element, XTML, of all XTML documents, and also specifies the base xml schema that XTML documents conform to.
4	Specifies the function that should act as the handler for a given event. Notice that events are identified by a symbolic name conforming to the convention "Vendor.Event.Version"[1]. This allows different vendors working independently to extend the XTML language by defining new events without concern for clashing identifiers
7	Defines a function and specifies the first action that is to be executed when this function is called (by means of the start attribute).
8-12	Defines the parameters that are passed to this function. In this case, the function takes three parameters: the port the call arrived on (an unsigned long), the dialed number information service (a string), and the caller line identification (a string). By default, these parameters are all passed by value.
13-15	Defines a local variable to represent the prompt to play, and initializes the variable with a value of 1001.
16	Defines an action and specifies the XTML server plug-in that contains the execution logic for this action. The XTML server will load this plug-in when the service is initially loaded, and will execute the service logic in this plug-in when this action is transitioned to during service execution. The id associated with the action is used in the link elements below to identify the next action to navigate to. The convention for defining XTML server plug-ins is similar to events, i.e., Vendor.Plug-in.Version[2]. An XTML Server must be able to instantiate and load a server plug-in based on this symbolic name, and an XTML Editor must be able to instantiate and load a editor plug-in based on this symbolic name.
17	A processing instruction to an XTML Visual Editor indicating the visual coordinates at which to position this action when displaying the service in an XTML editor.
18	The announcements element, and its child element, are not part of the XTML base schema. However, the implementer of the announcement plug-in has extended the XTML base schema by defining an extension schema that describes the additional elements and attributes that pertain to the announcement plug-in. In this case, the announcement instruction requires information about which announcements to play. The xmlns attribute defines a namespace for this element and its children that should override the base schema which was identified in the xml namespace attribute in the document (XTML) element defined earlier in the document.
21-23	Defines the transition processing from this action based on the possible results of this action. In this case, there is only one outcome, which transitions to the action identified with the id of 2.
34	Defines an instruction to end the session. When the execution reaches this action, the XTML server will trigger a Standard.EndSession.1 event for this session, call any associated handler

[1] There is also a "Standard" namespace for events, which defines the core set of events that all XTML servers are required to support. These are: Standard.AppLoad.1, Standard.AppUnload.1, Standard.SessionStart.1, Standard.SessionEnd.1, and Standard.Timer.1. These are all of the events that are generated by the XCPL Server itself, as opposed to the external implementation platform. All conforming XTML Servers must implement support for these events.

[2] There is also a "Standard" namespace for plug-ins, which defines the core set of plug-ins that all XTML servers are required to support. These are: Standard.FunctionCall.1, Standard.FunctionReturn.1, Standard.EndSession.1, Standard.Timer.1, and Standard.Load.1. These are the plug-ins necessary to call and return from functions, end a session, set timers, and load a service. The information model associated with each of these plug-ins are defined as part of the base XTML schema. All conforming XTML Servers must implement support for these plug-ins.

Line	Comment
	function for this event, and then terminate and restart the session. Note that this plug-in is in the "Standard" namespace, as described in footnote #2 earlier, and that no results are provided since there can be no outward path from this action to any other action.

3.4 XTML Service Execution

In the previous section we introduced the concept of sessions, and identified the XTML's major responsibility to be creating and managing sessions, but we have not yet described exactly how and when sessions get created and destroyed. In this section we'll review the lifecycle of sessions and describe the various ways in which services get loaded and sessions get created and destroyed.

A service can be loaded in one of two ways:

1. An external process, such as a service management system, requests the XTML server to load a specific service; indicating the service name, the number of sessions to create, and the conditions under which the XTML server should unload the service. Note that an external process can also be used to provide schedule-driven service availability (e.g., load 500 sessions of a message delivery service at 5pm daily, and unload the service after each session has completed one iteration).
2. A running session can issue a command to the XTML server to load a specific service, similarly specifying the service name, session count, and unload conditions. The XTML server responds by obtaining the XTML document and creating the specified number of sessions of the named service. The requesting session continues executing its own service logic, i.e., the two services are now running concurrently and independently in the same XTML server.

The details of the interaction in (1) above between a service management system and an XTML server are outside the scope of this document. The mechanism described in (2) above is accomplished by means of the Load instruction, which is part of the core XTML schema.

When the XTML server loads a service, it first triggers a 'ServiceLoad' event, which the service can handle if there is a need to perform any one-time initialization to prepare the implementation environment for the service.

Once the XTML server has loaded the service and executed any application load handler, it creates the requested number of sessions and triggers a 'SessionStart' event for each session. An XTML document will provide a handler for this event in order to execute any service logic that should be triggered once a session has started. Typically, an XTML document will handle this event by either launching directly into some service logic, or by indicating to the implementation platform an interest in receiving specified event types. For instance a prepaid calling application would register to receive incoming call events and would then wait to receive an incoming call event.

Once the session has started executing, it will continue executing actions until either the session returns from a function with no next action to execute or an 'EndSession' action is encountered. In the former case, the session will wait for an event that it has a handler for to occur (e.g., perhaps a hangup event or a timer event if a stable call is in progress). In the latter case, the XTML server will trigger an 'EndSession' event, in order to allow the session to perform any necessary session cleanup logic. Once the 'EndSession' handler has completed executing the XTML Server will reinitialize the session and then trigger another SessionStart event, which begins the session running again. This cycle will continue until the XTML server detects that the service unload conditions have been met, or is commanded to unload the service by an external process such as a service management system. At this point, the XTML server will trigger an 'ServiceUnload' event, to allow the service to perform any one-time cleanup logic, and will then unload the service and free all associated resources.

4 Extending XTML

The section above identified the major elements that are part of the XTML base schema. These elements provide the basic constructs needed to support concurrency, event handling, state information, and execution control. To this we need to add support for domain-specific events and actions in order to build real services. These constructs are added via extensions to the XTML base schema. This section describes how this is done.

Adding child elements to the <action> element is the primary mechanism for extending XTML. The XTML base schema defines the action element to have an open model, which means that it can be extended by the addition of new elements and attributes. The XTML base schema identifies only a single sub-element for the action element, and that is the <results> element that defines the result processing for a action. To this, a vendor can add additional elements that contain the configuration data required by the XTML server plug-in associated with this action.

The data needed for the execution of the plug-in at run time drives the design of extension elements. For example, an "Announce" plug-in will need information such as a list of announcements to play, and the number of times to repeat the announcement, whether to terminate the announcement if the subscriber speaks or presses a dtmf key on their phone, etc. On the other hand, a "Sleep" plug-in might need only information about the number of seconds to sleep. Therefore, the schema extensions for these two different plug-ins might result in elements such as below:

Figure 3: Announce XTML schema extensions

```
<action id="1" plug-in="Vendor.Announce.1">
    <announcements repeat=”3” xmlns=”x-schema:http://www.vendor.com/extensions-1.xml”>
        <announcement>1001</announcement>
        <announcement>1002</announcement>
    </announcements>
</action>
```

Figure 4: Sleep XTML schema extensions

```
<action id="1" plug-in="Vendor.Sleep.1">
    <sleep duration=”20” xmlns=”x-schema:http://www.vendor.com/extensions-2.xml” />
</action>
```

Notice that the new elements define a namespace for the attributes and sub-elements they contain, and reference an xml schema document that specifies the syntax for these new elements and attributes.

When the service is loaded, the XTML server will parse and load the XTML document. As it encounters extension elements that are sub-elements of the <action> element, it will validate them against the referenced schema, if provided, and then hand the xml text for these elements to the XTML server plug-in for the current action. The plug-in can then use the information provided via the xml text to configure itself for execution.

This allows arbitrary extensions to be made to the XTML languages without requiring any updates to an XTML server. The extension is released as a package containing the corresponding XTML server plug-in, the new xml schema document that defines the language extensions and, optionally, an XTML editor plug-in to make it easier for designers to create services that incorporate the new capabilities. With this package, services using the new language extensions can be deployed on any conforming XTML server the uses the same component framework as the plug-in.

5 XTML Application Server Overview

The previous section described how vendors can extend the XTML language to provide additional features. An important part of implementing these extensions is the creation of an XTML server plug-in that parses the xml syntax associated with these extensions and implements the service logic required. To this point in the document we have not discussed the component architecture requirements in much detail, other than to say that the architecture is agnostic about the specific choice of component framework (e.g., COM, Corba, EJB, etc). This section provides more details on the requirements that the XTML architectural model places on XTML Servers and the component interface definition.

The goal of the component architecture is that as one vendor or interest group extends XTML to provide additional functionality in a specific application domain, those extensions can be executed on any conforming XTML server. The intent is that the investment made to develop solutions in application-specific domains can be delivered to the broad market of all service providers interested in servicing that domain, not just a single vendor's platform.

In order to accomplish this, the definition of the interface between the XTML server and an XTML server plug-in must be standardized. Furthermore, in order to support multiple component technologies, several different interface definitions may be required, one per component technology. Although Interface Definition Language (IDL) is designed to be programming language-independent, there are differences between Microsoft's IDL and CORBA IDL, for example. Therefore, a specific interface will be defined for each of the following component solutions:

- COM (interface defined in MS IDL)
- CORBA (interface defined in CORBA IDL)
- Java/EJB (interface defined in Java interface signatures)
- Pactolus portable component architecture, PCA (interface defined in C++ method signatures)

Although each component solution will use a different interface definition approach, the basic functionality provided by an XTML plug-in component will be similar. The table below provides a high-level logical description of the interfaces that a component must support in order to plug-in to an XTML application server.

XTML Plug-in Interface (Major Methods)

Method	Description
Load	Called by the XTML server to deliver an element from an XTML document to the plug-in component. The XTML server passes through any elements below the <action> element that are not part of the base XTML server. It is the plug-in's responsibility to parse this xml text and use the information to configure itself accordingly. For instance, a plug-in that performs call connections would receive information describing the destination number to dial and the ring-no-answer timeout value to use.
Unload	Called by the XTML server when a service is being unloaded. This allows the plug-in to perform any necessary cleanup operations if it has some resources that need to be freed.
Execute	Called by the XTML server when this plug-in needs to be executed in the context of a running session. When the call flow transitions to a given action, the XTML server calls the Execute method on the plug-in that is associated with that action in the XTML document. This method is where the service logic for the plug-in resides. The plug-in can execute in either single-stage or multi-stage mode. In single-stage mode, the plug-in completes execution of its service logic and immediately returns a result to the XTML server as a return value of this method, allowing the server to transition to the next action. In

Method	Description
	multi-stage mode, the plug-in initiates some processing and then returns from this method, indicating to the XTML server that execution is ongoing. The XTML server will not transition to the next action, but will route incoming response messages to this plug-in as they are received, until the plug-in indicates that execution has completed.
ExecuteWithMessage	Called by the XTML server to route a message to a plug-in that is using multi-stage execution. The plug-in processes the message and returns, indicating to the XTML server whether its processing is now complete, or is still pending.
ExecutionTimedOut	Called by the XTML server to notify a plug-in that is using multi-stage execution that it has timed out. In the response to the initial Execute method, a multi-stage plug-in can optionally provide a timeout value in milliseconds that it wants the XTML server to enforce. Typically, this timeout value is provided in the xml text that the plug-in parses during the Load method. The XTML server calls the ExecutionTimedOut method if the requested timeout period has elapsed before the plug-in has completed processing. The plug-in should cancel any pending operations and indicate to the XTML server the timeout result path to use to transition to the next action.

The concept of single-stage versus multi-stage execution that is mentioned above bears a bit more discussion. Single-stage execution occurs when a plug-in can complete its processing within the Execute method callback. For example, a plug-in component that calculates a random number and assigns it to a session variable would use single-stage execution.

On the other hand, if a plug-in component needs to talk with external processes or network devices it might choose to use multi-stage execution. For example, a plug-in component that uses MGCP to create a connection would need to send an MGCP create connection command to an external process and receive a confirmation message in response. The latency between the command and response over a network connection might be 50 milliseconds. If the plug-in uses single stage execution to carry this out it will send the message and then wait within the Execute callback method for the confirmation response. From a performance standpoint, this has the undesirable side-effect of holding onto an operating system thread for the 50 milliseconds (or more) until the message returns or the operation is timed out. This would degrade scalability and performance at high call volumes where thousands of sessions might be trying to create connections simultaneously.

Instead, the plug-in component could employ multi-stage execution. It would send the MGCP create connection command in the Execute method. It would also register with the XTML server to receive any responses to that message and, optionally, request the XTML server to set a timeout value for the action. It would then return immediately from the Execute method, thereby returning its thread to the thread pool where it can be used by other sessions that are ready to perform some processing. When the XTML server later receives the confirming response message from the media gateway or softswitch, it will deliver the response message to the same plug-in by calling the ExecuteWithMessage method on that plug-in. The plug-in can then process the message and specify the result to follow for the next transition. Alternatively, if the message is not received within the specified timeout, the XTML server will call the ExecutionTimedOut method on the plug-in. This approach allocates units of work to threads more efficiently and supports greater scalability.

In addition, a reverse interface is also necessary. That is, an XTML server must expose an interface to the plug-in component so that the component can perform actions like querying and updating variable values, registering to receive response messages etc. Some of the major methods in this interface are described below

XTML Server interface (Major methods)

Method	Description
get_VariableValue	Called by the plug-in to retrieve the current value for a specified variable.
put_VariableValue	Called by the plug-in to update the value for a specified variable.
SendMsg	Called by the plug-in to ask the XTML server to send a message to an external process.
RegisterForMsg	Called by the plug-in to notify the XTML server of its desire to receive specific message types
WriteLogMsg	Called by the plug-in to ask the XTML server to write a message to the system log file.

These two interfaces standardize the communication between an XTML server and a plug-in extension component. With this background we can now specify the requirements for an XTML server. A fully-conforming XTML server must meet the following requirements:

- It must be able to load and execute XML files or text containing XTML-described services.
- It must be capable of hosting COM, CORBA, Java/EJB, and PCA plug-in components; i.e., implementing the appropriate XTML Server interface for each and making the appropriate calls on the XTML plug-in interface.
- It must provide support for the Standard package of events and plug-ins.

A partially-conforming XTML Server is an XTML Server that supports only a subset of the specified component models; e.g., a partially-conforming XTML Server may support COM only, or CORBA and PCA only, etc. A partially-conforming XTML Server would thus be restricted in terms of the which types of XTML plug-ins it could host.

6 XTML Editor Overview

An XTML Editor provides a highly visual way of designing and modifying XTML services. Although XTML services can be designed in any standard XML editor or text editing tool, the complex branching logic inherent in any sophisticated communications service can make it unwieldy to do so. An XTML editor provides a visual, direct manipulation method of arranging call flow paths in a two-dimensional drawing space that makes it much easier to design these applications.

An XTML editor also supports a plug-in architecture that allows vendors that extend XTML to provide an enhanced level of configuration support by supplying property pages that make it much configure the information required for each action. However, an XTML language extension can be released without an editor plug-in, and the service designer will still be able to design services by directly editing the xml text for the language extension within the XTML editor. In this case, a user would still get the benefits of the graphical tools to visually arrange the call flow or state machine transitions. An editor plug-in simply offers a greater degree of customization and ease of use in designing services by providing easy-to-use property pages and validation of XTML language extension syntax. Note that the XTML editor loads and saves the same xml text that the XTML server does; no additional files, configuration information, or transformations are necessary.

The example below illustrates how a simple prepaid calling application can be viewed and edited in an XTML editor. The prepaid application answers an incoming call, performs a database lookup to determine how much calling time the subscriber identified by the calling number has, sets a timer for the call duration, and then places the requested call. When the timer goes off, the service responds by hanging up on the caller and ending the session. If the caller hangs up before the timer goes off, then the session simply ends.

Here's how the service looks in its raw XTML form:

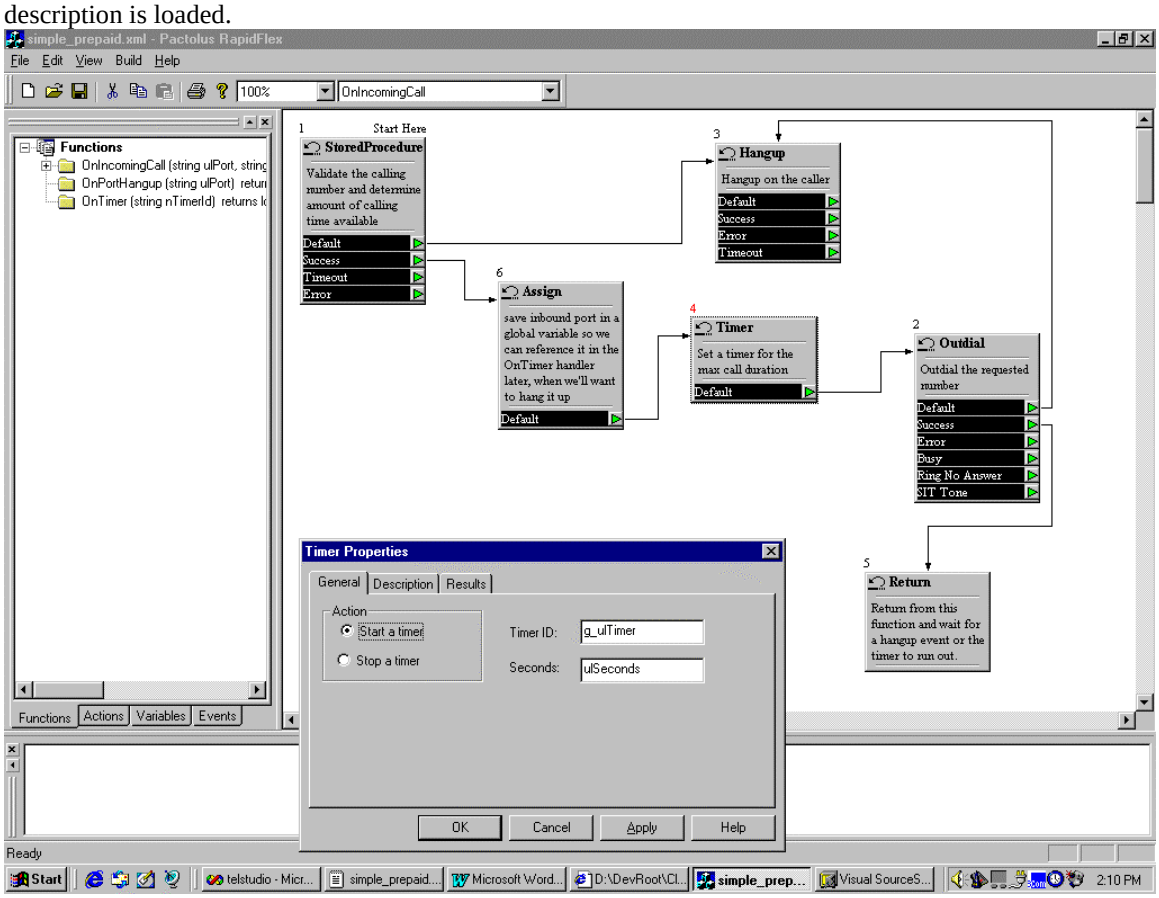
Figure 5: simple XTML prepaid calling application

```
<?xml version="1.0"?>
<xtml name="prepaid" xmlns="x-schema:http://www.pactolus.com/XTML-base-schema.xml">
    <events>
        <event name="Pactolus.IncomingCall.1" handler="OnIncomingCall" />
        <event name=" Pactolus.Incoming.1" handler="OnPortHangup" />
        <event name="Standard.OnTimer.1" handler="OnTimer" />
    </events>
    <global-vars>
        <var name="g_ulTimer" type="ui4">0</var>
        <var name="g_ulInboundPort" type="ui4">0</var>
    </global-vars>
    <functions>
        <function name="OnIncomingCall" start="1" returns="i4">
            <parameters>
                <parameter name="ulPort" type="string" />
                <parameter name="strDNIS" type="string"/>
                <parameter name="strCLI" type="string" />
                <parameter name="nMediaType" type="string" />
                <parameter name="lCallArriveTime" type="string" />
                <parameter name="strInfoDigits" type="string" />
            </parameters>
            <local-vars>
                <var name="ulSeconds" type="ui4">0</var>
            </local-vars>
            <actions>
                <action id="1" plug-in="Pactolus.SQLStoredProc.1"
                    xmlns="x-schema:http://www.pactolus.com/XTML-ps1-extensions.xml">
                    <?xtml-editor x=15 y=24 ?>
                    <!-- Validate the calling number and determine amount of calling time
                     available-->
                    <sql-stored-procedure timeout="" return-value=""
                    connection="ulConnection" statement=""retrieve_calling_time"">
                        <parameter type="input">strCLI</parameter>
```

```
            <parameter type=”output”>ulSeconds</parameter>
        </sql-stored-procedure>
        <results>
            <result name="Default" link="3" />
            <result name="Success" link="6" />
            <result name="Timeout" />
            <result name="Error" />
        </results>
    </action>
    <action id="2" plug-in=" Pactolus.PlaceCall.1"
        xmlns="x-schema:http://www.pactolus.com/XTML-ps1-extensions.xml">
        <?xtml-editor x=561 y=198 ?>
        <!-- Outdial the requested number-->
        <call dest=”8005551212” cli=””strCLI”” timeout=”10”/>
        <results>
            <result name="Default" link="3" />
            <result name="Success" link="5" />
            <result name="Error" />
            <result name="Busy" />
            <result name="Ring No Answer" />
            <result name="SIT Tone" />
        </results>
    </action>
    <action id="3" plug-in="Pactolus.Hangup.1"
        xmlns="x-schema:http://www.pactolus.com/XTML-ps1-extensions.xml">
        <?xtml-editor x=384 y=29 ?>
        <!-- Hangup on the caller-->
        <hangup port="ulPort" timeout="" return-value="" hangup-time="" />
        <results>
            <result name="Default" />
            <result name="Success" />
            <result name="Error" />
            <result name="Timeout" />
        </results>
    </action>
    <action id="4" plug-in="Standard.Timer.1">
        <?xtml-editor x=363 y=184 ?>
        <!-- Set a timer for the max call duration-->
        <timer start="1" id="g_ulTimer" duration="ulSeconds" />
        <results>
            <result name="Default" link="2" />
        </results>
    </action>
    <action id="5" plug-in="Standard.FunctionReturn.1">
        <?xtml-editor x=517 y=410 ?>
        <!-- Return from this function and wait for a hangup event or the timer
         to run out.-->
        <return value="" />
    </action>
    <action id="6" plug-in="Pactolus.Assign.1"
        xmlns="x-schema:http://www.pactolus.com/XTML-ps1-extensions.xml">
        <?xtml-editor x=191 y=152 ?>
        <!—save inbound port in a global variable so we can reference it in
        the OnTimer handler later, when we'll want to hang it up-->
        <assignments>
            <assignment>g_ulInboundPort = ulPort</assignment>
        </assignments>
        <results>
            <result name="Default" link="4" />
        </results>
        </action>
    </actions>
</function>
<function name="OnPortHangup" start="1" returns="i4">
<parameters>
    <parameter name="ulPort" type="string" pass="byval" />
</parameters>
<actions>
    <action id="1" plug-in="Standard.EndSession.1">
    <?xtml-editor x=96 y=87 ?>
```

```
            </action>
        </actions>
    </function>
    <function name="OnTimer" start="1" returns="i4">
        <parameters>
            <parameter name="nTimerId" type="string" pass="byval" />
        </parameters>
        <actions>
            <action id="1" plug-in="Pactolus.Hangup.1"
                xmlns="x-schema:http://www.pactolus.com/XTML-ps1-extensions.xml">
                <?xtml-editor x=69 y=47 ?>
                <!-- Time is up, so hang up on the caller.-->
                    <hangup port="g_ulInboundPort" timeout="" return-value=""
                    hangup-time="" />
                <results>
                    <result name="Default" link="2" />
                    <result name="Success" />
                    <result name="Error" />
                    <result name="Timeout" />
                </results>
            </action>
            <action id="2" plug-in="Standard.EndSession.1">
                <?xtml-editor x=264 y=61 ?>
            </action>
        </actions>
    </function>
</functions>
</xtml>
```

The image that follows shows the user interface that an XTML editor provides when the same service description is loaded.



Clearly, the XTML editor provides a much easier to use, graphical interface for designing XTML services. Users can drag actions from the actions palette and drop them into functions. Each function is displayed in a separate view, and users can easily navigate between functions using the dropdown list below the menu bar. Links between actions are easily created by clicking the mouse over the result indicator of one action and drawing a line to the next action that should be executed if this result is true. Actions can be easily configured by double-clicking on them to bring up a property page provided by the XTML editor plug-in for that component.

In the case above, an XTML editor plug-in was provided for the timer action and the associated property page is open, showing the information that needs to be configured for this action. The document can be checked for completeness and validity at any point by means of a menu option. The XTML editor also translates the service description into a self-documenting design document that can easily be printed, distributed, and viewed to show the call flow path and high-level logic.

Because of the general nature of the requirements, XTML editors can be developed in a variety of languages or tools, such as C++, Java, etc. Pactolus has developed the XTML editor pictured above for Windows NT/9x using C++ and COM as the plug-in extension framework.

This example demonstrates that although XTML services can be created using a variety of tools, an XTML editor provides the simplest and most powerful way of quickly designing complex new services. Because XTML editors generate pure XML files, services can be designed initially in an XTML editor, then modified in text editors or other tools, then brought back into the XTML editor, without loss of information.

7 XTML Schema

The XTML language is described below using the XML Schema language.

```
<schema targetNamespace="http:://www.pactolus.com/XTM"
        xmlns="http://www.w3.org/1999/XMLSchema"
        xmlns:xtml="http://www.pactolus.com/XTML">

<annotation>
    <documentation>
    XTML schema specification.  XTML stands for extensible telephony markup language.
    XTML is designed to describe voice and data personal communication services that will
    be delivered over convergent packet-switched networks.
    </documentation>
</annotation>

<!--The document, or root, element of an XTML document is the XTML element -->
<element name="xtml" type="xtml:xtml"/>

<complexType name="xtml">
    <group order="all">
        <element name="global-handlers" type="xtml:global-handlers"/>
        <element name="global-vars" type="xtml:global-vars"/>
        <element name="shared-vars" type="xtml:shared-vars"/>
        <element name="functions" type="xtml:functions"/>
        <element name="script" type="xtml:script"/>
    </group>
    <attribute name="version" type="decimal"/>
</complexType>

<complexType name="global-handlers">
    <element name="handler" minOccurs="0" maxOccurs="*">
        <complexType>
            <attribute name="event" type="string"/>
            <attribute name="function" type="string"/>
        </complexType>
    </element>
</complexType>
```

```
<complexType name="global-vars">
    <element name="var" type="xtml:var" minOccurs="0" maxOccurs="*" />
</complexType>

<complexType name="shared-vars">
    <element name="var" type="xtml:var" minOccurs="0" maxOccurs="*" />
</complexType>

<complexType name="var">
    <complexType base="string" derivedBy="extension">
        <attribute name="name" type="string"/>
        <attribute name="type" type="xtml:var-type"/>
    </complexType>
</complexType>

<simpleType name="var-type" base="string">
    <enumeration value="i2"/>
    <enumeration value="ui2"/>
    <enumeration value="i4"/>
    <enumeration value="ui4"/>
    <enumeration value="float"/>
    <enumeration value="string"/>
    <enumeration value="boolean"/>
</simpleType>

<complexType name="functions">
    <element name="function" type="xtml:function" minOccurs="0" maxOccurs="*" />
</complexType>

<complexType name="function">
    <group order="all">
        <element name="parameters" type="xtml:parameters"/>
        <element name="local-vars" type="xtml:local-vars"/>
        <element name="actions" type="xtml:actions"/>
    </group>
    <attribute name="name" type="string"/>
    <attribute name="start" type="unsigned-long"/>
    <attribute name="returns" type="xtml:var-type"/>
</complexType>

<complexType name="parameters">
    <element name="parameter" minOccurs="0" maxOccurs="*" type="xtml:var"/>
</complexType>

<complexType name="local-vars">
    <element name="var" type="xtml:var" minOccurs="0" maxOccurs="*"/>
</complexType>

<complexType name="actions">
    <element name="action" minOccurs="0" maxOccurs="*" model="open">
        <complexType>
            <group order="all">
                <element name="results" type="xtml:results"/>
                <element name="scripts" type="xtml:scripts"/>
            </group>
            <attribute name="id" type="unsigned-long"/>
            <attribute name="plug-in" type="string"/>
        </complexType>
    </element>
</complexType>

<complexType name="results">
    <element name="result" type="string" minOccurs="0" maxOccurs="*">
        <complexType>
            <attribute name="name" type="string"/>
            <attribute name="link" type="unsigned-long"/>
        </complexType>
    </element>
</complexType>
```

```
<complexType name=”scripts”>
    <element name=”script” type=”xtml:action-script” minOccurs=”0” maxOccurs=”3”/>
</complexType>

<complexType name=”script”>
    <complexType base=”string” derivedBy=”extension”>
        <attribute name=”language” type=”string” minOccurs=”0”/>
    </complexType>
</complexType>

<complexType name=”action-script”>
    <complexType base=”xtml:script” derivedBy=”extension”>
        <attribute name=”timing” type=”xtml:script-timing”/>
    </complexType>
</complexType>

<simpleType name=”script-timing” base=”string”>
    <enumeration value=”first”/>
    <enumeration value=”middle”/>
    <enumeration value=”last”/>
</simpleType>

</schema>
```