SERVICE CREATION TECHNOLOGIES FOR THE INTELLIGENT NETWORK

Michael J. Morgan, Michael J. Cosky, Thomas M. Gruenenfelder, T. Curtis Holmes, Jr., and Gerald A. Raack

***Michael J. Morgan, Michael J. Cosky, Thomas M. Gruenenfelder, T. Curtis Holmes, Jr.**, and **Gerald A. Raack** all work for AT&T Bell Laboratories. Mr. Raack is on temporary assignment at AT&T Network Systems (United Kingdom). Mr. Cosky is a supervisor in the Advanced Technology Laboratory, located at Indian Hill, Naperville, Illinois. In that capacity, he is responsible for resolving user-interface issues for new network-based services that employ emerging technologies. He joined AT&T in 1981, after receiving a B.A. from the University of Detroit, Michigan, and an M.A. and Ph.D. from the University of Texas, Austin, all in the field of psychology. Mr. Gruenenfelder is a distinguished member of technical staff in the Switching and Network Administration Systems Department in Holmdel, New Jersey.*
(continued on page 71)

An intelligent network lets a switch query a network node for help in completing a call. Such a node, for example, can translate an 800 number into a true network address. The network service provider often creates the logic in an intelligent network, but frequently that logic is written by sophisticated subscribers (business customers). Since subscribers can create their own service logic, the service creation system must ensure that such customized logic will not cause the network node to fail or in any way affect the service of other subscribers. Service creation technologies are the tools and technical infrastructure designed specifically to help create and update services. AT&T has created software tools to support application-oriented languages. Its service creation technologies are evolving to meet service provider and service subscriber demands for more control over the services they provide and use.

Introduction

Service creation technologies are the tools and technical infrastructure designed specifically to help create and update services.[1] These include current services, such as Advanced 800 (Freephone), televoting, and private virtual networks (which allow a subscriber to create a set of telephone numbers for the company's various locations); and upcoming services, such as Fax Mail and the Who's Calling service. The Who's Calling service uses text-to-speech technology to announce the name of the calling party to the called party without requiring anything but plain old telephone service (POTS).

The tools used for service creation technologies consist of editors, compilers, debuggers, and execution environments, the same types of tools the industry has used to create the existing base of switching software. AT&T has taken this traditional set of tools, however, and tailored them to meet specific service creation needs by following an application-oriented language[2] philosophy. These tools constitute a service creation environment, which enables the network operator and/or the service subscriber to create and develop new intelligent network services.

Languages such as C, Pascal, and assembly provide a set of general-purpose programming constructs, such as data declarations, assignment, compare/branch, and function calls. The data declarations normally include standard types, such as integers, real numbers, and characters. They frequently let the programmer build new data types based on existing ones. Since these constructs make no assumptions about the problem, a programmer can use them for different types of tasks.

Application-oriented languages incorporate an application model and programming constructs geared to that model. Text-processing languages assume a world containing such items as fonts, paragraphs, section headers, footnotes, and references, and they provide constructs for manipulating them. Some general-purposes languages make application assumptions. Fortran, for example, is used for number manipulation and Snobol for string manipulation.

An application-oriented language increases productivity by supplying high-level, application-oriented commands and data structures that a programmer would otherwise have to write. Although a programmer could write a document using the C language, most people would prefer to use a text-processing language. AT&T has applied this concept to service creation and designed languages that assume a telephony model, geared to call processing and operations.

Running a program written in a general-purpose language usually requires only a compiler and a target computer. Application-oriented languages most often run on a virtual machine, a software system that executes the instructions of a program written in an application-oriented language. A compiler, for instance, cannot easily map into machine language such directives as "connect party A to party B" or "ring party C for no more than 10 seconds." The compiler may provide part of the virtual machine by breaking high-level commands into multiple lower-level commands, but these latter commands may be executed by a software platform and/or hardware designed to run on the target machine.

Panel 1. Abbreviations, Acronyms, and Terms

1NCP	AT&T's No. 1 Network Control Point system
ASCII	American standard code for information interchange
CCITT	International Telegraph and Telephone Consultative Committee
DSDC	direct-service-dialing capabilities
ETSI	European Telecommunications Standards Institute
Fortran	formula translator (language)
OAM&P	operations, administration, maintenance and provisioning
Snobol	string-oriented symbolic language
SQL	structured query language

Service Creation Evolution

Call scripts and decision graphs are application-oriented languages that let programmers sequence high-level telephony primitives, such as "play announcement *x*," or "connect to destination *y*." Although they may offer the same capability, call scripts present the capability linearly, and decision graphs format it as a tree. Because decision graphs impose structure on programs, they tend to be understood and verified more easily than call scripts.

The direct-services-dialing capabilities (DSDC) provided the AT&T network's first service creation platform using application-oriented-language technology designed in a decision graph format. AT&T introduced DSDC in 1983 to give customers more flexibility in defining their services and faster response to their changing needs.[3] Within this framework is the service management system, which allows service administrations and their service subscribers to create and maintain customized routing plans that define exactly how an intelligent network will handle calls for each subscriber. The service management system offers two layers of application-oriented languages: a high-level, decision graph language for service providers and subscribers that it translates into a lower-level, decision graph, application-oriented language. DSDC runs on AT&T's No. 1 Network Control Point (1NCP) system, a service control point,[4] and provides the virtual machine for the lower-level application-oriented language. The service control point is a real-time database system which, based on a query from the service switching point, executes subscriber-specific logic and returns instructions to the service switching point on how to continue the call processing.

Since the introduction of DSDC, AT&T has also introduced call-script-based service creation languages on its Conversant® speech processor system [5] and Definity® communications system.[6]

AT&T's initial efforts emphasized how to create a new service quickly by making changes at the service management system, while the capabilities at the service control point and service switching point were held more or less constant. Thus, the focus was on moving the programmability into the service management system and making it available to subscribers. Decision graphs were an effective means to simplify service construction.[3]

The level of service creation described earlier was on the leading edge of technology until 1987, when exploration at a different level began. Network service providers wanted to create their own services without contracting through an equipment or intelligent network vendor. They wanted to use a service creation environment to program services for a service control point, as well as for other intelligent network nodes. AT&T Bell Laboratories conducted extensive research on this topic near Chicago. The work resulted in an evolution of the decision-graph application-oriented language approach. It also introduced a finite-state-machine application-oriented language to address the event-driven, asynchronous nature of general call processing. The two languages can interact to form a network programming language, described later in this paper. AT&T is continuing its efforts to solve the problems and extend its offerings of service creation within the intelligent network, in both the domestic and international markets.

The definition of service creation is evolving as the industry develops ambitious plans and expands its frontiers. Several standards bodies have formally addressed service creation in an attempt to standardize the definition and concepts. As an example, a multivendor interaction (MVI) forum was convened early in 1989.[7] It had several working committees, consisting of industry representatives, who addressed not only network architecture, but also call model, service creation, service-logic-execution environment, billing, and OAM&P (operations, administration, maintenance, and provisioning).

Technologies in Use

Today, AT&T products support service creation by using decision graph programming languages. NETSTAR is a service management system that allows users to create services for international intelligent networks; the 1NCP system is the execution platform that acts as the service control point. The next two sections elaborate on this technology and explain why decision graphs are especially good structures for service programming.

Service Creation for Intelligent Networks. Service creation technology is being used in the international market to offer services such as Advanced Freephone, calling card, televoting, and private virtual network. NETSTAR supports the creation, customization, validation, and administration of the subscriber's call-routing plans. Service creation technology gives service providers and service subscribers (business customers) the flexibility to customize and create services. Service subscribers of routing-control service can create their own call-routing plans.[8]

International telecommunication network providers and service subscribers customize their services with an ASCII-based user interface that conforms to enhanced Man-Machine Language. Using a set of predefined building blocks called user nodes, service creators can construct the call-processing logic for different services. Branches stem from these user nodes in a directed, graph-like structure. The user nodes fall into three categories: decision nodes, action nodes, and terminating nodes.

Decision nodes. Service creators can define services based on decision nodes, such as time of day, day of week, command routing, call prompter, holiday, and percentage distribution of calls, which we will describe here. By selecting and sequencing these nodes, a service provider or subscriber can customize services to meet the subscriber's business needs. Decision nodes have several possible outcomes, depending on the type of decision. Each possible outcome is a "call branch."

Time of day nodes route calls along call branches, depending on what time of day the call originates. For example, we might use a time node to route calls to one location during normal business hours and to another location after business hours.

Day of week nodes route calls along call branches, depending on which day of the week the call originates. A day node might route calls to one location on weekdays and another location on weekends.

Holiday nodes route calls according to the specific national holiday on which the call originates. For example, if January 1 is a national holiday and all business locations are closed, a service subscriber can route the call to an announcement that asks the caller to try again during normal business hours. The holiday node must be defined by the network service provider.

Command-routing nodes route calls if unpredictable events occur, such as equipment failure. When calls must be handled quickly, a branch on this node can be activated in less than five minutes with a near-real-time update.

Call-allocator (distribution) nodes route calls along branches, according to percentages that are set for each branch. The distribution node matches call volume at a location to the number of people available to answer those calls. For example, using this node, a service subscriber could route 60 percent of the calls to one location and 40 percent to another location.

Call-prompter (select) nodes route calls to a location chosen from a list of alternatives. The caller then hears an announcement, which directs him or her to enter certain digits based on where the call should be routed. For example, the caller might be asked to enter 1 to reach the sales department, or 2 to reach the service department.

Action nodes. Using action nodes, service creators can specify various actions as they move through a decision tree. Action nodes include sample, goto, busy, and queue nodes.

- *Sample* nodes collect information about area of origin, day, time, and assigned terminating location for a specified percentage of calls passing through the nodes.
- *Goto* nodes avoid duplicating complex structures in plans by directing calls to a specified node. For example, if two sections of the plan direct calls to the set of nodes X, Y, and Z, then that set of nodes need only appear in one section. "Goto" instructions can direct the system to the set of nodes in the earlier section of the plan.
- *Busy* nodes increase the likelihood that a call will be completed when all lines at the primary destination are busy. These nodes sequentially search an ordered list of call destinations and route calls to the first available location identified in the search.
- *Queue* nodes place calls arriving at a particular location into a first-in, first-out holding queue if all lines at the primary destination are busy. This maximizes the likelihood that every call will be answered.

Terminating nodes. Terminating nodes (line, signal, and courtesy) allow users to specify where various types of calls will be terminated.

- *Line* nodes terminate calls at specific locations whose line numbers are listed in data files.

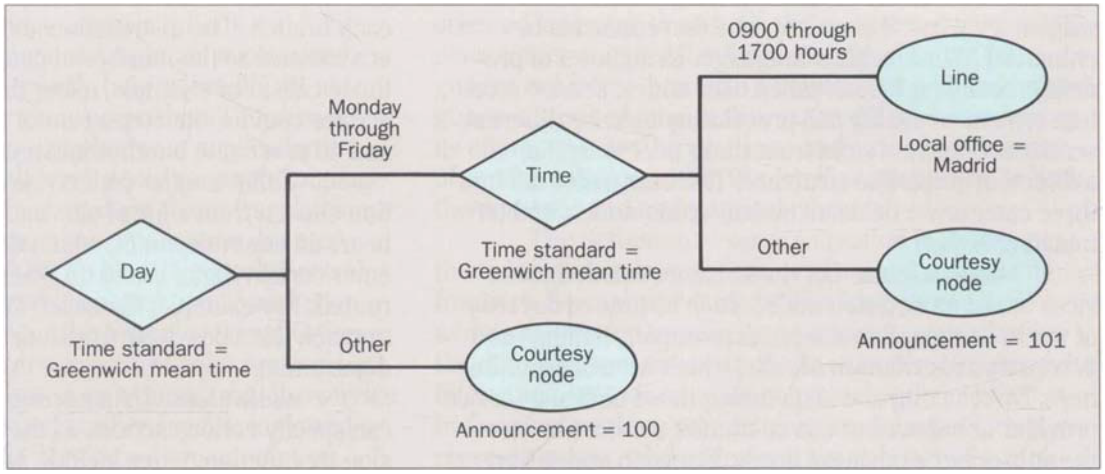

Figure 1. A sample call-routing plan created by service subscribers using day- and time-decision nodes. The service programmer specifies the day node and its branches, as shown. The time node routes all specified calls to either a line node or a courtesy node. In this example, all calls that originate Monday through Friday, between the hours of 9 a.m. and 5 p.m., are routed to the primary office in Madrid. Weekend and nonworking hour calls are answered by an appropriate announcement from a courtesy node.

- *Signal* nodes play one of several specified signals (tones, announcements, or silence) to the caller, but they do not route the call further.
- *Courtesy* nodes play a customized announcement to callers and then terminate the call, particularly when the call cannot be answered otherwise.

Sample programs. Figure 1 shows an example of an Advanced Freephone service created using the day- and time-decision nodes. NETSTAR prompts the service programmer to enter a node. After the programmer has specified the day node, NETSTAR requests the first branch entry. The programmer specifies one path for Monday through Friday and enters that information. NETSTAR then prompts the programmer to account for the remaining days of the week. He or she specifies "Other" to account for the remaining days.

To base weekday call-routing decisions on the time of day, the programmer selects a time node. To obtain certain terminating treatment during normal working hours, 0900-1700, and other treatment during nonworking hours, the programmer enters "0900-1700" on the first branch of the node and "Other" on the second branch.

NETSTAR then prompts the programmer to enter another node. At this point, the programmer can use the line node to terminate calls made during normal working hours at the primary office in Madrid. However, during nonworking hours, the programmer wants to play a courtesy announcement to all callers. Therefore, he or she enters the courtesy announcement node.

For security reasons, the service subscriber cannot directly enter the line number, but instead must include it from a data file referenced by this particular call-processing plan. This security mechanism helps prevent fraudulent or inadvertent activities that could result in misrouting of calls. Service providers have sole responsibility for entering this type of data.

Figure 2 depicts how a service provider or

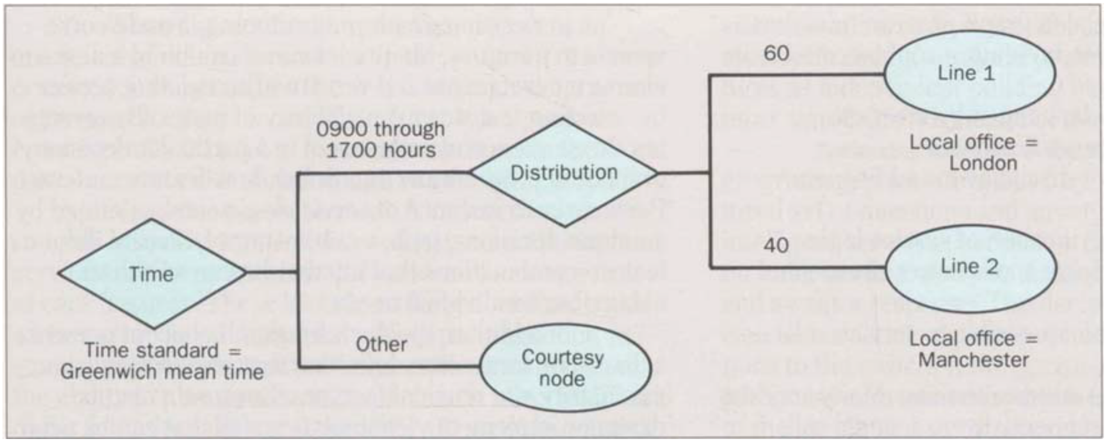

Figure 2. A sample call-routing plan using a distribution node. After creating the time-decision node, the programmer creates a distribution node. All calls made between 9 a.m. and 5 p.m. go subsequently to the distribution node, which routes 60 percent of the calls to Line 1, in London, and 40 percent to Line 2, in Manchester. This call distribution corresponds to the staff distribution available to take the calls. Calls made during nonworking hours are routed to an appropriate announcement from a courtesy node.

subscriber can provide customized services based on percentage distribution of calls. In this scenario, after creating a time-decision node, service programmers create a distribution-decision node. As branches to the distribution node, the programmer enters 60 and 40, which routes 60 percent of the calls to Line 1, in London, and 40 percent of the calls to Line 2, in Manchester. Subscribers can thereby match expected call volume with the percentage of staff available at particular locations.

Other capabilities. The current generation of service creation technology can also:
- Provide billing data
- Generate reports
- Translate data from a format readable by users to one needed by the service control points
- Distribute data to various service control points.

NETSTAR can bill service subscribers for disk space used by the call-routing plans, for the number of nodes of each type in the call-routing plan at the service control points, and for the time it takes the subscriber to interact with NETSTAR to create or modify a plan.

NETSTAR generates numerous customized reports to support service creation capabilities. These include service management reports used by service providers and service subscribers, troubleshooting reports, monitoring and management reports, and network administration reports. From a service management perspective, service subscribers can obtain data on how the busy and queue nodes are being used. These data will help them determine if they need alternate routes, different services, and/or additional staff.

NETSTAR also distributes call-routing plans to one or more service control points.

Advantages of Decision Graph Programming. Decision graph programming was one of the first service creation technologies available. In 1983, when decision graph programming was introduced, alternative approaches were only able to define service logic using traditional, general-purpose programming languages and to support service customization through simple

parameter changes. Decision graph programming has several advantages over today's more sophisticated tools because it:
- Enables service providers and subscribers to sequence service features flexibly
- Provides a structure or discipline for adding new features to services
- Supports automatic verification of service logic
- Uses the same underlying application software for more than one service
- Makes service creation accessible to service subscribers.

Flexible sequencing of service features. Many services invoke several features sequentially on a single call. From the perspective of decision graph programming, a node in a graph represents a feature, and a path through a decision graph specifies the sequence in which the individual features will be invoked. Decision graph programming allows the service creator — either the service subscriber or the service provider — to specify the node sequence, on a subscriber-by-subscriber basis. This characteristic has two advantages:
- First, decision graph programming determines the feature sequence when the service is provided, rather than when the underlying application software is developed. Therefore, the designer of the application software need not pinpoint the optimal sequence of features. As the number of features that can be used on a call increases, the number of decisions also increases multiplicatively.
- Second, the sequence of features can affect the user's perspective of the service if the user is either a caller or a service provisioner. Decision graph programming allows a subscriber the flexibility to sequence features, thereby giving its users the best service possible.

Adding new features. As services are enhanced with new features, determining how new features interact with old ones is of paramount importance. The service design team must make these decisions early, because they will affect some subscribers positively and others negatively.

In decision graph programming, a node corresponds to a feature; the programmer can build a new service or a new customized version of an existing service by selecting features from a library of nodes. By arranging the sequence of nodes used in a particular decision graph, the programmer can define how features interact. Because each instance of service logic can be defined by a unique decision graph, a subscriber can create the feature combinations that interact best to suit that subscriber's individual needs.

In addition, the decision graph concept presents a discipline for feature definition that encourages true modularity and reuse of features. Because a feature designer implements features as nodes that can be used in many combinations with other nodes, he or she can implement each feature as an independent building block that performs one function well in any application.

Verification of service logic. Automatic program verification, in which one program determines the accuracy of another program, is a notoriously difficult area that still relies on good judgment as much as on a set of well-defined techniques. The structure of decision graph programs, however, lends itself to the definition of simple, yet robust, verification rules. A call-processing decision graph program must be able to route any call that it receives.

Pragmatically, all paths through the decision graph must end in a terminating node; all decision nodes must have an outcome for each possible value of the decision parameter. For example, a node that makes a decision according to the day of the year must have an outcome specified for each of the 366 possible days of the year. For a given decision node, each value of the decision parameter must have only one outcome assigned to it. The graph also cannot contain infinite loops. Given the structure imposed by the decision graph paradigm, algorithms for implementing these checks are straightforward and, consequently, can verify the logical consistency of a call-processing decision graph program simply and automatically.

When the verification software uncovers an error, it also determines where in the program that error occurred; it then notifies the decision graph programmer of its exact location. In a decision graph program, an error typically is isolated to a single node or, in the worst case, to a particular path through the decision graph.

Support for multiple services. A variety of what can roughly be called translation, routing, and authorization services involves repeated applications of a common set of core features. These include making decisions based on time, the dialed digits, the calling-party number, personal identification numbers, and authorization codes; the ability to play an announcement and collect additional digits from the caller; and the ability to provide network-routing instructions to a switch. Area-number calling, incoming-call screening, outgoing-call management, and virtual network services all perform these functions.

In a decision graph programming environment, the underlying software interprets the decision graph and executes each type of node encountered in the graph. The decision graph environment implements each node as an independent building block. The programmer can write the decision graph software in a generic way that makes no references to the particulars of a given service. Once a node is implemented for one service, it can be used immediately by all services within the decision graph paradigm. Obviously, all services may not need a particular node, but most services can support a wide variety of services using a core set of nodes.

Service creation by service subscribers. Many sophisticated service creation technologies still require new software applications to undergo extensive laboratory verification before they are used. The expense of laboratory verification, along with associated security issues, makes it difficult for network services to offer these tools directly to service subscribers. It is relatively easy to develop software that can verify the logical consistency and completeness of decision graph programs. In addition, with only minimal training, service subscribers can readily comprehend and construct decision graphs. Service subscribers can write decision graph programs with minimal training, verify them simply, and implement features as independent building blocks. This is perhaps the most significant advantage of the decision graph paradigm.

Enhancing Decision Graph Programs. Decision graph programming was developed by AT&T in 1980 as a structured set of questions and answers. When a call requires intelligent network processing, the switch places the call on hold, sends a query to the decision graph program, and awaits a response. The decision graph program then executes a serial sequence of nodes and returns instructions to the switch. If the decision graph program needs additional information from the caller, it issues a request to the switch and stops processing the call until that request is answered.

This mode of operation has been used successfully in a wide variety of network services. However, certain services, such as multiway calling and call transfer, require the service logic to respond to unexpected (asynchronous) events, such as button presses and switchhook flashes. To address such services, AT&T is developing additional application-oriented languages, modeled after finite-state machines.

A-I-Net™ Technologies

In 1987, AT&T began to develop new tools and platforms to support service creation for intelligent network nodes. Based on the premise that specially designed application-oriented languages enhance programmer productivity, this effort expanded on the already successful decision graph paradigm and integrated it with a new finite-state-machine language. A new A-I-Net service creation environment incorporating these results will work initially with the A-I-Net service circuit node[9] and the A-I-Net service control point, two intelligent network nodes built on a common platform specifically designed to support service creation.

The sections that follow describe the service creation environment and the design-by-use process that develops it.

Developing New Technologies. While developing application-oriented languages for the service creation environment, AT&T recognized that an iterative, rapid prototyping method was most suited to the process. The service-creation-environment project identified two critical elements of a successful application-oriented language — its "usefulness" (in handling the constructs required to support the intelligent-network service domain) and its "usability" (to a service developer designing and writing code). Accordingly, AT&T adopted a design-by-use process that could iterate on application-oriented-language designs and empirically test them with users. The design-by-use process consisted of three major iterative stages: capturing expert knowledge, designing application-oriented languages, and empirically testing these languages.

Capturing expert knowledge. In this process, language designers interviewed experienced and expert software developers who had contributed to a number of AT&T switching products. These experts were experienced in developing and maintaining services and had worked on more than one phase of the software life cycle. They helped the project members develop a basic view and framework of some of the essential components of the service creation environment and its application-oriented languages. The project members also studied existing service creation literature and participated in emerging standards activities to ensure that the views they formed were consistent with those of the larger telecommunications community.

Designing application-oriented languages. In this stage, the project's designers developed a "first-cut" design of the functionality and form of the application-oriented languages. They decided that an application-oriented language should be kept simple; thus, the initial language included the smallest possible set of language capabilities for handling a wide array of intelligent network services. They designed the basic framework of the language to be flexible enough to handle additional capabilities as the language was exposed to daily use.

Empirical testing of application-oriented languages. Armed with an initial design of application-oriented languages and a service-creation-environment prototype, the designers worked closely with users in a customer company to evaluate systematically whether the languages were useful and usable for particular intelligent networks. Some key components of each experiment included a *language tutorial*, in which users were taught the new language, followed by extensive "hands-on experience" with the language, such as writing and debugging services, and reading and maintaining code written by others. The designers also measured how quickly and easily users could learn the application-oriented language, how well they could remember it, how easy the resulting code was to comprehend and maintain, and how consistent it was with the developers' knowledge base and its application.

The designers used the results of the empirical tests to refine and improve the language. They then subjected the revised version of the application-oriented language to the same empirical test to identify further improvements.

Designers working on the service-creation-environment project found the design-by-use approach to be extremely effective and powerful for developing application-oriented languages. They now understand user requirements empirically and can integrate new software technology, expert knowledge, intelligent-network service demands, and feedback from users who are also potential customers of an service-creation-environment product.

A Service Creation Environment. The A-I-Net service creation environment consists of a set of tools that runs on UNIX® workstations and platform software that runs on the service execution vehicle (e.g., the A-I-Net service circuit node). (UNIX is a registered trademark of UNIX Systems Laboratories, Inc.) The tools comprise a high-level user interface, a network-service-execution environment, and a set of support tools for the network programming language, including editors, compilers, and an integrated debugger.

The platform software is essentially a telephony "operating system" with "system calls," such as accept call, open path, collect digits, and diagnose announcement circuit. These service-independent building blocks implement the functionality that Bellcore refers to as the service logic execution environment (SLEE).[10] It provides the virtual machine that executes the network-programming-language software.

User interface. The user interface is a programmable graphical interface that uses the standard X-Window graphics protocol originally developed by the Massachusetts Institute of Technology. It logs the service creator into the system and provides access to the service creation environment's various tools. Users of the service creation environment are typically the service providers or subscribers.

Since one would expect users of the service creation environment to find new tools that they would like to use within the environment, the user interface is programmable. A service-creation-environment programmer can tell the system about the new tool, how to access it, and how to prompt and interact with the programmer. For tools that were not designed to run in a window environment, the user interface can supply a window and optional custom, application-specific buttons.

The administrator of the service creation environment sets up a default tool set for the users. Each service creator is free to modify these defaults. The service creator also can restore the configuration to the system defaults at any time by selecting the appropriate item from a menu.

In the same spirit, the user interface is not tied to any specific source-control tool. The administrator can tell it how to interact with the team's favorite source management system.

Network execution environment. The network execution environment is a somewhat simplified, workstation-based version of the platform (service circuit node and service control point) software. It lets a service creator test the service software on his or her workstation. It also incorporates software to simulate some functionality of class 5 switches (e.g., 5ESS® switches). A class 5 "office," for example, lets a tester set triggers for its various "lines." Service creators, therefore, can compile their services and test them in a network environment.

A network programming language. The network programming language is an integrated suite of several application-oriented languages. Figure 3 depicts a three-layer programming model. This three-layer model provides a programming domain for each provider or subscriber. It enhances productivity because:

- Each programmer has a restricted domain.
- Each layer of programming can create many services without relying on a lower level for more functionality.

The bottom layer represents the relatively static platform software. The top two layers represent the service logic written using a network programming language.

The middle layer contains software written in an application-oriented language of the network programming language's finite-state machine, called the service-logic language. The service creation environment enables the user to organize this software into *service packages*, independent combinations of service software, each of which serves its own set of subscribers. Subscribers need not be people. A service package that supports televoting, for example, might support voting on a given election by dialing a special telephone number. In this case, each different election that the package supports would be a different "subscriber."

A service package may support multiple, possibly interacting, services or features appropriate for its subscribers. For example, a subscriber may have a personal-locator service that routes a subscriber's call to a home, office, automobile, or other location. The location, specified in a default plan by the subscriber, usually depends on such factors as the day of week or time of day. The subscriber may also want the *Who's Calling* service, which announces the name of the calling party. A service package that supplied both services would let

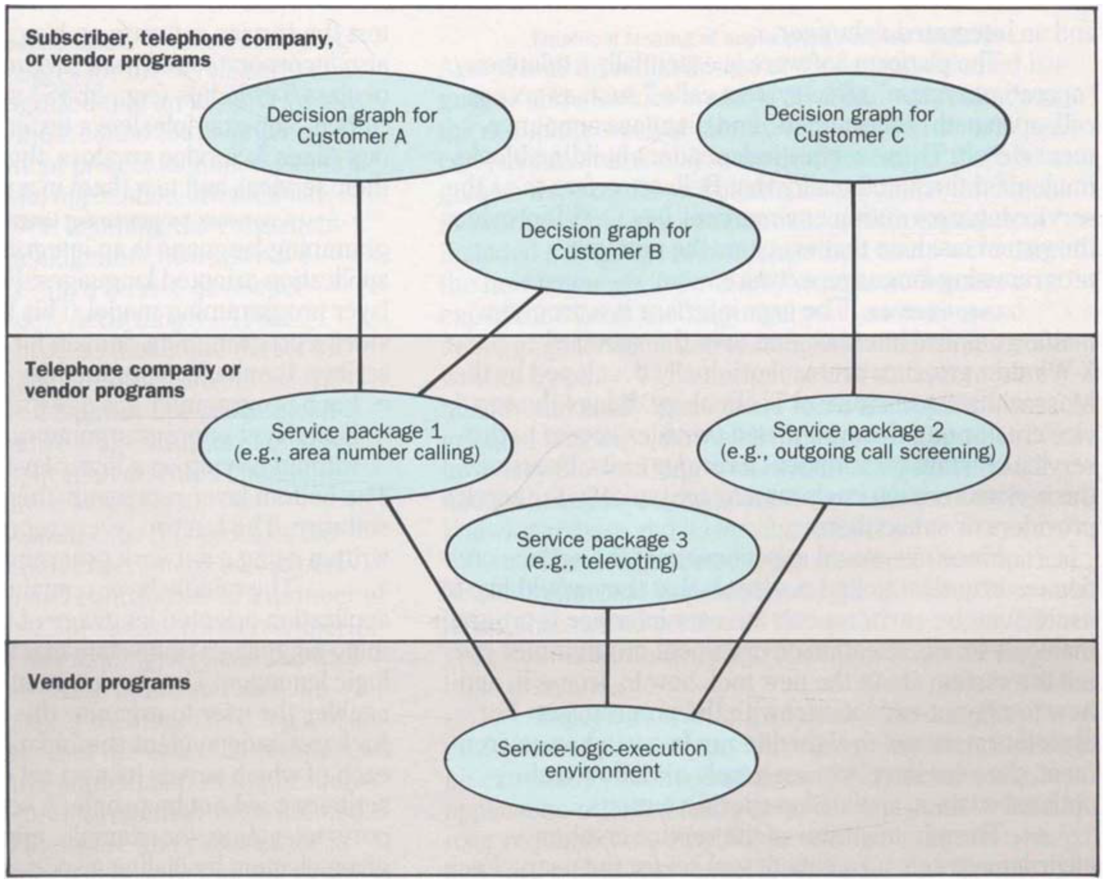
Subscriber, telephone company, or vendor programs
Decision graph for Customer A
Decision graph for Customer C
Decision graph for Customer B
Telephone company or vendor programs
Service package 1 (e.g., area number calling)
Service package 2 (e.g., outgoing call screening)
Service package 3 (e.g., televoting)
Vendor programs
Service-logic-execution environment

Figure 3. The bottom layer of this three-layer programming model represents the relatively static platform software. The top two layers depict the service logic, written using a network programming language.

the Who's Calling service follow the subscriber around, according to the subscriber's routing plan.

When a service programmer uses the service logic language to create services, the language lets the programmer package parts of the services (features) as decision graph nodes. The programmer can then combine these nodes into decision graph programs that provide subscriber-specific algorithms, such as the personal-locator plan mentioned above. The top layer of Figure 3 corresponds to this decision graph programming layer, supported by the network programming language's decision graph language. There is no restriction on the number of decision graphs that can be used for a service. The service logic language programmer can design a service to use as many decision graphs as necessary. Presumably, however, multiple decision graphs would be used for answering different questions. For example, one graph might tell the service package how to route the call for a particular subscriber, whereas another graph might tell it how to interact with the party called.

Although no restrictions exist in the three-layer model regarding who can program any particular layer, usually the vendor would supply the platform software, and the service provider would concentrate its programming at the service-package layer. The service provider might then offer its subscribers the opportunity to customize services using decision graphs.

The service creation environment provides two additional application-oriented languages that are associated with the layered model. These languages let a service creator specify how to lay out forms for database entry and templates for service-measurement reports.

Service logic language. Call processing is generally event driven in nature: the software *responds* to events, such as incoming calls, Touch-Tone digits, and disconnects. These events are frequently *asynchronous*; they can occur independently, and the software cannot necessarily finish processing the first event before it starts to process the second. The finite-state-machine programming model is well suited to address such an environment. It provides a structure for specifying an *event handler*, a piece of software the system can invoke when the given event occurs. The software can set the "state" of the program as a way of recording which events have been fully or partially processed. When a new event arrives, the system can choose which event handler to execute, depending on both the event and the current state.

The service logic language allows the service creators to write event-driven, finite-state-machine programs. It interacts with the underlying platform software by making requests called *actions*. A program, for example, could use the "accept call" action to ask the system to answer an incoming telephone call. The program views the results of an action as events. These events are also asynchronous, so a program can issue multiple outstanding action requests. It does not have to wait for an action to be completed before it issues a second request.

Because the language must be extensible in a changing intelligent network environment, its interface to the SLEE, the set of legal actions and events, is a library. There is no need for the language compiler to be changed in order to change this set. The administrator of the service creation environment can make new platform actions or events available to the service logic language by updating a file that specifies their legal syntax.

The service logic language also lets the programmer declare and manipulate data structures that are especially appropriate to telephony, such as a telephone number. For each service package, the service logic language compiler produces executable C++ software, structured query language (SQL) commands, default forms to populate and update those relations, default-report templates to generate reports that summarize measurements taken by the service package, and decision-graph-node information for each node/feature that the service package supports. The SQL commands are used to create database relations for provisionable data.

The language assumes that the service provider or subscriber normally supplies provisioning data using a craft or service-management-system interface. However, it provides a mechanism that lets a service package update this data, based, for example, on a caller interacting with Touch-Tone service.

Measurements are an integral part of the language, and the service creator must explicitly declare them and associate them in natural groupings that appear in the default reports. The system usually collects these measurements over multiple calls and sends them periodically to a measurements database. A billing measurement is a special type of measurement taken on a per call basis. The language provides special billing data structures, and the service programmer makes a specific request to the system to send the data to the billing system.

Decision graph language. The network programming language's decision graph language evolved from the decision graph language described earlier. It shares the same concepts of decision, action, and terminating nodes, although it does not implement the same set of nodes. The language is graphical and runs on top of the X-Window system. It is a dynamic language that it can rapidly incorporate new nodes specific to a given service package, based on the node information generated by the service-logic-language compiler.

Form and report languages. The service-logic-language compiler produces default forms for provisioning a service package's data and default templates to generate measurement reports. These defaults use form and report languages associated with the underlying database. A service creator can edit these forms and templates to modify the default layouts. The service creator also can provide provisioning and report access to the craft or, if desired, directly to subscribers.

Network-programming-language debugger. The network programming language has an integrated debugger that can work with both service-logic-language and decision-graph-language programs. It can be attached to a running service package, which eliminates the need to start the debugger before starting the program to be tested. The debugger provides capabilities designed for the applications, such as:

- Injecting events, that is, making the system think that a new event has occurred, such as a new call arriving, or a Touch-Tone digit being pressed;
- Trapping on events, states, and lines, which involves setting a breakpoint to stop the program when the specified event occurs, when it enters the specified state, or when it reaches the specified line;
- Examining variables, that is, determining the value of a specified variable.

For decision graphs, the debugger can interact with the decision graph editor to display the path through the decision tree that the system traverses.

Platform software. The platform software must respond to high-level requests issued by the service logic. These requests may be oriented towards call processing, as in "collect digits," or towards OAM&P, as in "diagnose circuit" or "restore circuit." They are asynchronous, that is, the platform executes requests as it continues to process service logic and returns any result to the service logic as an event. The platform also presents unsolicited entries to the service logic as events. For call processing, these entries might correspond to service requests: a subscriber making a call that gets connected to the service circuit node or the service switching point request to an intelligent network node. An OAM&P request for service might originate at an operations system or a maintenance terminal.

Conclusion

Specialized service creation technologies are not new to AT&T. For more than ten years, the direct-services dialing capability has served as a platform for service creation based on decision graphs. But the growing complexity of network services and configurations inherent in the intelligent network places new demands on these technologies.

AT&T's A-I-Net service creation environment meets these demands by building on the successful history of decision graph programming and by adding new application-oriented languages to enhance programmer productivity. The network programming language evolved as both internal and external customers used it to create intelligent network service. By incorporating both the finite-state-machine and decision-graph-programming models, the service creation environment meets service providers' programming needs and lets subscribers customize their AT&T services. The service creation environment addresses both call processing and OAM&P. It automates the service-based provisioning and report generation processes and makes them accessible to subscribers.

Service creation needs will continue to change. The AT&T service creation environment addresses this issue by incorporating new tools and by abstracting the interface between the service creation environment and the service logic execution environment platform into easily updatable specification files. AT&T will continue its research into service creation technology and apply application-oriented language and other approaches to meet future service creation requirements.

References

1. R. E. Bright, M. J. Morgan, and E. J. Weiss, "Service Creation in an Intelligent Network," *Proceedings of Globecom '89*, Vol. 1, November 1989, pp. 137–140.
2. D. W. Brown et al., "Advanced Software Technology Supporting Customer Programmability," *AT&T Technical Journal*, Vol. 68, No. 2, March/April 1989, pp. 33–46.
3. G. A. Raack, E. G. Sable, and R. J. Stewart, "Customer Control of Network Services," *IEEE Communications Magazine*, Vol. 22, No. 10, October 1984, pp. 8–14.
4. E. G. Sable and H. W. Kettler, "Intelligent Network Directions," *AT&T Technical Journal*, Vol. 70, No. 3–4, Summer 1991, pp. 2–10.
5. S. A. Riederer, "*CONVERSANT*® VIS Means Business," *AT&T Technology*, Vol. 5, No. 4, 1990, pp. 14–18.
6. *DEFINITY*® *Communication System Generic 2 and System 85 Feature Description*, Vol. 1, AT&T Customer Information Center, Indianapolis, Indiana.
7. "Bellcore Multi-Vendor Interactions Compendium of 1989 Technical Results," SR-TSY-001629, Issue 1, March 1990.
8. V. P. Gupta et al., "End User Administration of an Intelligent Network," *Proceedings of the IEEE International Conference on Communications '89*, Boston, Massachusetts, June 11–14, 1989, Volume 3, pp. 1182–1187.
9. H. M. Hall et al., "The AT&T Service Circuit Node: A New Element for Providing Intelligent Network Services," *AT&T Technical Journal*, Vol. 70, No. 3–4, Summer 1991, pp. 72–84.
10. Bellcore, "Advanced Intelligent Network Release 1 Network and Operations Plan," *Special Report SR-NPL-001623*, Issue 1, June, 1990.

Biographies (continued)

Currently, he is working on service management systems, service creation capabilities, and end-user control of service logic. He earned an A.B. in psychology from the University of Notre Dame, Notre Dame, Indiana, and a Ph.D. in cognitive psychology from Indiana University, Bloomington. Mr. Gruenenfelder joined AT&T in 1980. Mr. Holmes is supervisor of the Service Management Systems Planning and Architecture Group in the Service Management Systems Department in Columbus, Ohio. He is responsible for developing an A-I-Net service management system architecture that can meet global needs. After earning a B.S. from Southern University, Baton Rouge, Louisiana, and an M.S. from Purdue University, Lafayette, Indiana, both in computer science, Mr. Holmes joined AT&T in 1984. Mr. Morgan is a supervisor in the Advanced Intelligent Network Department in Indian Hill, where he is developing an A-I-Net service creation environment to support AT&T's service circuit node and service control point. He joined AT&T in 1980, after earning an A.B. in applied mathematics from Harvard University, Cambridge, Massachusetts, and an M.S. in computer science from the University of California at Berkeley. Mr. Raack is located in Malmesbury, England, where he is a supervisor in the Applications Software Development and Systems Engineering Department. Currently, he is responsible for startup of development on the International Network Control Point in the United Kingdom. He joined AT&T in 1967, after receiving a B.S. in mathematics from Ohio State University, Columbus, and an M.S. in computer science from Purdue University.

(Manuscript received February 5, 1991)