AN ATM-Based Platform for Rapid Generation of Multimedia Applications

Eric R. Beyler
Olivier B. Clarisse
Edward A. Clark
Y. H. Levendel
Robert E. Richardson

The Software Assembly Workbench (SAW) approach is used to assemble software by combining reusable building blocks, much in the way some commodity hardware is being designed. SAW requires a network execution platform that performs network functions necessary to execute typical services and provides access to telecommunications and computing functionality. SAW is based on two adjacent software layers, the upper service layer and the intermediate component layer. Both require the implementation of the lower capability layer, which resides on the network execution platform. This building block approach enables the rapid creation and customization of software that can be dependably executed in a distributed telecommunications network to create both narrowband and multimedia broadband services.

Introduction

The entry of foreign and domestic competitors into both the previously protected U.S. and global markets is creating strong pressures on the pricing policies and response times of telecommunications equipment manufacturers. At the same time, deregulation is creating the need for interworking between diverse products. The semi-monopoly held by local service providers and the large research and development expenses required to develop modern telecommunications equipment make it difficult to meet these growing needs.

An analogy to this situation can be found in the computer industry. Early on, computer vendors, operating as semi-monopolies, took advantage of their position and urged customers toward large proprietary mainframes that for a while perpetuated their monopolies by excluding smaller competitors. Yet in less than a decade, the era of mainframes and of market monopolization has succumbed to low-cost processing and has given way to the market trends of distributed computing and market diversity.

Even though the large capital investments[1] needed to change the telecommunications infrastructure are slowing down these market trends, one still can expect, by the turn of this century, a similar opening up of the telecommunications industry and a proliferation of less expensive, distributed network solutions. For example, as the cost-to-performance ratio continues to go down, computers now used primarily as peripheral equipment will play a larger role in delivering services.

In effect, the world-wide balkanization of the telecommunications market is around the corner. The main question that arises is "What technological solution will best position the telecommunications industry to face these new challenges?" From a business viewpoint, this question becomes "How can we speed up the delivery of proprietary products and services in an open environment?" At the present time, software is the main bottleneck in reducing both cost and time to market, and it is essential to improve the software production process. To address this business need, technology must be developed that has two complementary attributes:

– Speeding up of software production, and
– Differentiating software by customization.

The next section of this paper examines the basic principles that are likely to

speed up software delivery through reuse. These principles are embodied in the system architecture of an application development environment, the Software Assembly Workbench (SAW), and an execution environment, the network platform, which are both described in the section "System Architecture." Since the implementation of these techniques is strongly domain dependent, the section "Rapid Creation of Multimedia Applications" shows how they can be applied to deliver multimedia services. Finally, pragmatic considerations related to an ATM-based distributed network are discussed in the section "A Few Fundamental Issues."

Panel 1. Abbreviations, Acronyms, and Terms

ACE — application creation environment
API — applications programming interface
ATM — asynchronous transfer mode
CPE — customer premises equipment
CRT — cathode ray tube
FPGA — field-programmable gate array
GTA — general terminal adapter
NTSC — National Television Systems Committee
PAL — phase alternate lines
PC — personal computer
SAW — Software Assembly Workbench

Speeding Up Software Construction

Constructing Software Like Hardware. The computer hardware industry has succeeded in defining reusable components that have allowed hardware designers to be more effective in their work, both in speeding up design and improving its quality. A significant portion of the hardware logic design has been reduced to establishing the connectivity of predefined components and to verifying the correctness of the assembly. By using a catalog of hardware components, the design process is greatly simplified and many of its phases can be automated.

This paper demonstrates a software construction method similar to hardware design methodologies in that it is based on component reuse to speed up software assembly. Of course, there still is a need to customize some hardware components, such as field-programmable gate arrays (FPGAs) and application specific integrated circuits (ASICs), to provide specification flexibility, but effective tooling has even been developed to speed up that customization. Similarly, there is significant room in software to develop custom components and for design aids to facilitate their development.

Software Reuse. In its most general definition, software reuse is practically unattainable, although it may be an attractive goal, and in that sense, it has proved to be an elusive goal. In very specific instances, however, software reuse has achieved high benefits. LOTUS 1-2-3* is the simplest example of reuse by end users, while MS-DOS* and more so UNIX* offer reusability of command level functions to create larger, more powerful scripts. In Microsoft Office,* parts of one essential application, such a Word for Windows,* can be reused in other applications. In all such successful cases, several essential issues stand out.

Software Components Must Expand. In the last few decades, software abstractions have increased in scope so that each language *construct*, such as a statement or program, spans a larger *content*. In other words, the assembly language equivalent of each construct has been growing. The *expansion factor* is a measure that expresses the content of a language construct. In that sense, a line of C++ code has a larger expansion factor than a line of C code. Empirical evidence has shown that software productivity is in direct relation to the expansion factor of the language used.[2]

Reusable Components Must Be Specialized. In all disciplines, modern large-scale industrialization, which began on Henry Ford's automotive assembly line, has led to a high degree of specialization, which has resulted in the intensive reuse of interchangeable parts to create many different products. In software as well, basic objects, such as a finite number of elementary language constructs, are being reused to produce larger entities, or programs.

In a way, software production is almost 100 years behind Mr. Ford's hardware industrialization. The main problem with reusable software constructs is that, in general, they are not specialized enough and, therefore, capture no data after solving a problem, called *problem solution knowledge*, that is reusable in other applications. Thus, a large expansion factor alone cannot move software production into a new software "industrial age." However, in combination with high specialization, it is

likely that we can begin to overtake Mr. Ford.

Reusability is Often Traded for Functionality. With time, the balance between expansion and specialization can change. For instance, in the late seventies, General Motors reduced its production costs by replacing the engines in its high-end Oldsmobile cars with mid-range Chevrolet engines, to the chagrin of some of its customers but to the satisfaction of most of its stockholders. Of course, while designing reusable software components that have a large expansion and high specialization to provide higher productivity, one is likely to face the question: "What if a component cannot provide specific behaviors—such as horsepower, whether for a car or a computer—that may be needed in a given situation?"

In such a case, four solutions are possible:

1. Forego the new, specific required behavior as long as the reusable component still has enough residual merit.
2. Modify the component to adjust its behavior as long as other functionality is not affected.
3. Create new components that combine lower level elements of a "standard" set, such as a macro capability.
4. Forgo the benefits of this, and other reusable components, and design the new, required software as usual.

Obviously, the first solution is the cheapest one, as long as the revenue from the unchanged component is high enough. In this case, one is trading higher reusability for lower functionality. Unless this tradeoff is realistic, there can be no long-range value to a software assembly methodology based on reusable components. Fortunately, experimental data in computing and telecommunications has shown that, in most applications, a small percentage of functionality is being used the most.[3] This is the old "80 - 20" issue—80 percent use of 20 percent of the assets—which raises the hope of designing components that can cost-effectively cover most evolving needs. These, indeed, will be robust components.

Reuse Requires Agreement. For one person to reuse a component designed by someone else, an agreement is necessary between the component designer and its user about the component's functionality. In the most general case, agreements are almost impossible to achieve. However, a broad agreement is possible when the economic benefits of such an agreement outweigh the loss of freedom, as it is the case for major operating systems, environments, and tools.

In most other cases, one has to be able to design frameworks of moderate size where agreements are possible and can be maintained. Such frameworks of limited scope can help fill the gap between the two extreme situations mentioned above. In our case, we use the SAW design environment tools to define an area where agreement is possible.

Domain Analysis is the Key to Reuse. In most industries, successful reuse requires a careful analysis of how and where the application is to be used, that is, it's *domain*, and breaking down the application into a set of robust components representing that domain. This set of reusable components is then consolidated into a "*palette*" that can be inserted into the SAW for reuse in other applications' domains. On one hand, domain analysis—in this case, multimedia services—allows the development of specialized, reusable software. On the other hand, domain analysis requires the solution of a new set of problems that are particular to that domain—such as bandwidth requirements and computer-to-computer communication. Although we have acquired some experience in analyzing telecommunications applications by developing diverse application domains, domain analysis of software products still remains more an art than a science.

Quality Up Front. Although formal methods have often been advocated for improving the quality of software, we have not found these methods to be cost effective. In the absence of practical, more robust methods, constructing software from reusable components contributes to improving the upfront quality of the software by relying on better component quality. Indeed, with time, multiple use of a component tends to improve its quality and that of the higher-level assemblies using it.

On a pragmatic level, reuse will yield more dependable software. The basic principles described above were put in practice a few months ago for a narrowband application, and a significant gain in productivity was observed. In the first three months of use, the reuse technique yielded a productivity gain of 3:1 without any decline in quality, in comparison to the traditional programming of service software. As in the case of any introduction, the method and the software production process need to be tuned, which will likely provide additional productivity gains. Our target is to reach a 10:1 improvement.

Having stated the broad principles underlying the approach, we now will demonstrate how to carry it out for a particular domain, that of multimedia applications.

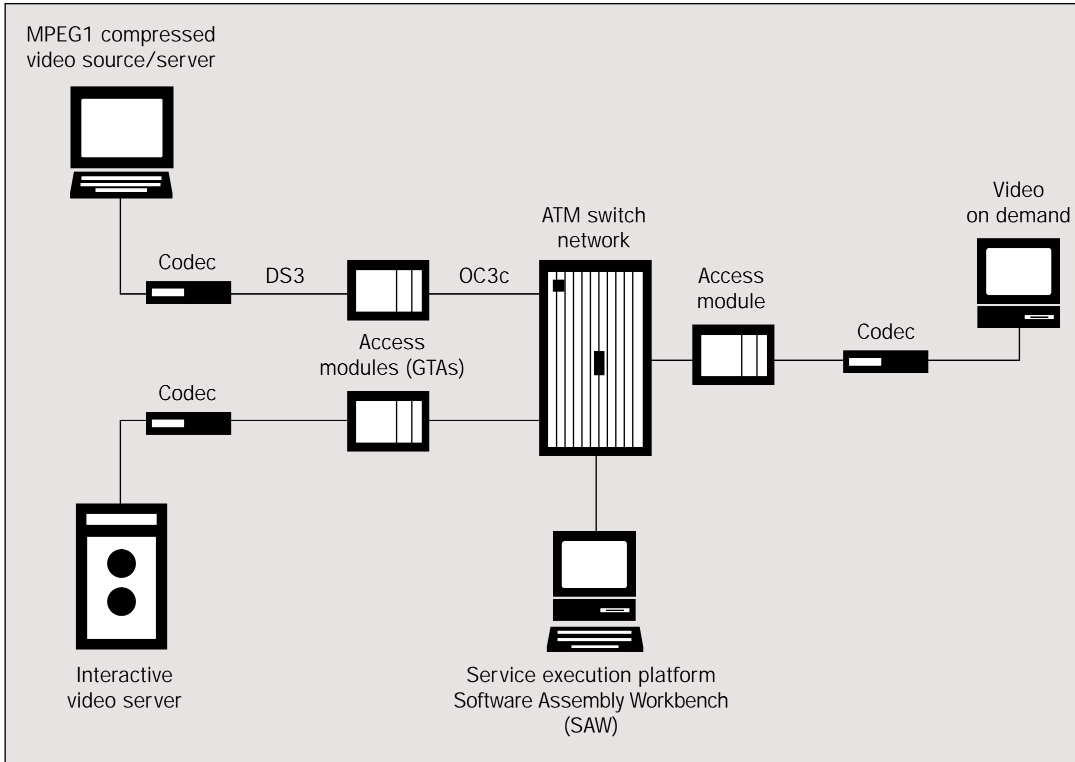

Figure 1. The hardware environment for multimedia application includes several elements: an asynchronous transfer mode (ATM) switch fabric, multimedia personal computers and workstations, general terminal adapters (GTAs) for converting various media to and from ATM protocols, a PC-based video server, and workstations to provide control environments for the network, as well as for the service execution platform, the Software Assembly Workbench.

System Architecture

A Platform for Easy Programmability. The hardware environment for our multimedia application includes several elements: an asynchronous transfer mode (ATM) switch fabric, multimedia personal computers (PCs) and workstations on the customer's premises, general terminal adapters (GTAs) for converting various media to and from ATM protocols, a PC-based video server, and workstations to provide control environments for the network, as well as for the application execution (Figure 1).

The software environment is composed of four parts: customer premises equipment (CPE) control software, network control software, the SAW application creation environment (ACE), and the application execution environment.

When designing a system for providing telecommunications services, it is important to minimize the coupling between the various layers, particularly between the *service environment* and the *hardware environment*. This helps to allow for the independent evolution of the individual layers.

It is almost a given that software written today for one telecommunications hardware platform will be executing in ten years on a different platform. In our prototyping platform, minimizing such coupling has allowed us to replace the switching fabric we were using, and its supporting software, without changing any service-level code. The first version of ACE ran on a baseband video switch, which was replaced with a prototype ATM switch, together with the GTAs. Today, the switch is being changed yet again, moving from the prototype ATM to the AT&T GlobeView® 2000 ATM switch. The GTAs also are evolving without affecting the application software.

In addition to the need to layer the system vertically, it also is important to realize that telecommunications services also are partitioned horizontally. The primary partitioning is between services that execute within a network and services that execute in CPE at the edge of the network. This partitioning is a reflection of the fact that different types of resources are used in the middle of the network—such as switching fabrics, trunking, adapters, and bridges—than at the edge of the net-

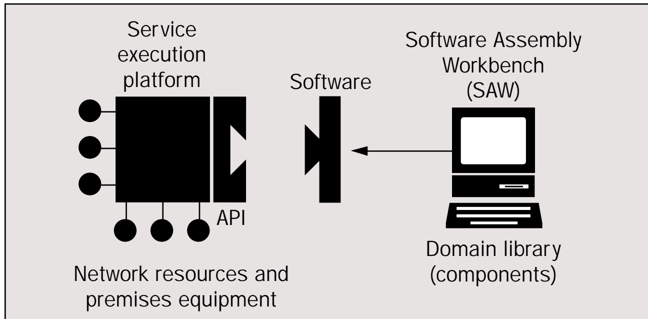

Figure 2. The execution system has a client-server architecture composed of network resources (the servers) and the execution platform (the clients). Each network resource provides specific functionality—such as voice systems, computers, and databases—which can be distributed in the network and is used to execute a service. On the basis of the existence of the capability layer (the applications programming interface, or API), two additional layers can be derived using domain analysis: the component layer and the service layer (see Table 1).

work—such as CPE personal computers and workstations, servers, microphones, speakers, cameras, CRT screens, and control buttons.

The basic construction of ACE is independent of where it executes. By allowing for easy customization, ACE can create services that control resources within the network or services that control resources at the edge of the network.

Execution System: Client-Server Architecture

As shown in Figure 2, the execution system has a client-server architecture composed of network resources (the servers) and the execution platform (the clients). Each network resource provides specific functionality—such as voice systems, computers, and databases—that can be distributed in the network and used to execute a service.

In the context of multimedia services, resources can be video-servers and video-conferencing bridges, which are connected to the execution platform through a telecommunications network. In Figure 2, the SAW and the service execution platform are shown as separate computing environments for conceptual ease, but this is not a fundamental issue. In fact, we have implemented a version where both functions are performed by the same platform.

Software Assembly Workbench. The SAW is a software design environment with the following ingredients:
- Graphical user interface;
- Visual representation of a basic set, or "palette," of components;
- Mechanism to parametrize the components;
- Visual software construction paradigm to combine components;
- Incremental editing capabilities;
- Simulation mechanism to verify the assembled software; and a
- Mechanism to produce executable software.

These ingredients are necessary to facilitate the construction of executable software by application programmers.

Rapid Creation of Multimedia Applications

Software Layering. On the basis of the existence of the capability layer (the applications programming interface or API in Figure 2), two additional layers can be derived using domain analysis: the component layer and the service layer (see Table 1). The higher the layer, the easier the programming should be and the larger the revenue potential. In this multilayered software architecture, each layer is insulated from the layer below and is used to construct the layer above it. The right column of Table 1 represents the location of this element during the software assembly. However, during execution, all of the software will be shipped to the execution platform and executed there.

Resource Programmability. The resources are programmed with two layers of programming: the *lower layer primitives* and the *upper layer capabilities*. The decision to have two layers was largely accidental and due to the fact that, in our experience, the resources used were already equipped with an "operating system" providing "native" primitives. Somewhat arbitrarily, the capabilities were defined as an additional layer, creating more compact operations that would minimize the message load between client and servers. Sample server capabilities for interactive video-on-demand are given in Table 2.

Different capabilities will be necessary for additional application domains. For instance, several application domains will require data functionality that can be covered in the following capabilities:
- Business databases,

Table 1. Software layers

Layer elements	Used to construct:	Element's location
Services		Software Assembly Workbench
Components	Services	Software Assembly Workbench
Capabilities	Components	Execution platform
Primitives	Capabilities	Execution platform

Table 2. Interactive video-on-demand capabilities

Request video titles	Request list of videos from video server
Request video	Request video title to be played
Release video	Release video title
Video status	Request status of video title
VCR command	Control playback of video title

- Electronic files,
- Electronic mail,
- Facsimile,
- Modem communication, and
- Directories.

In general, the capability definition will be driven by domain definition, will be grouped into resources, and will execute on resource servers. The resources being a combination of telecommunications and computing resources, our proposed software and hardware architecture is a good mechanism to wed computing and telecommunications to provide new service revenues for the industry.[4]

The Software Assembly Workbench. Establishing a framework to capture domain analysis—the SAW—opens up opportunities to change the way software is constructed. Now, there is room for a new form of programming, *software assembly*, performed by individuals less versed in software design but more qualified in application domains. In telecommunications, this allows an agent of the service provider, the operator of a SAW, to "retail" services to the end customers. The service software can now run either on equipment belonging to the service provider or to the end customer.

The resources necessary to support a service can belong to the service provider, such as telephone directory database; to the end customer, such as customer orders; or to a third party, such as computer databases and processing nodes. These are the principles used in delivering telecommunications services, and in particular, video-on-demand over a broadband infrastructure.

Software Assembly Paradigm. To construct applications in a given domain, it is assumed that a vocabulary and a grammar are defined for the domain. Application domain software is designed by constructing sentences from chosen elements of the vocabulary, organized according to the grammar rules.

The domain vocabulary is represented using a palette of icons (upper left side of the window in Figure 3) that instantiate, or change, into components when placed on the design area. This example of domain vocabulary—that is, the components used to construct services—includes menu execution, play, stop, rewind, and customer authentication. Other domains may be included in different palettes.

A diagram comprises a set of operators and a set of connections representing associations between output events and input events. An example of such a diagram, presented in Figure 3, resembles a logic circuit diagram. However, each connection between operators does not represent an absolute point-to-point "hardwired" connection but, instead, a *potential* path for

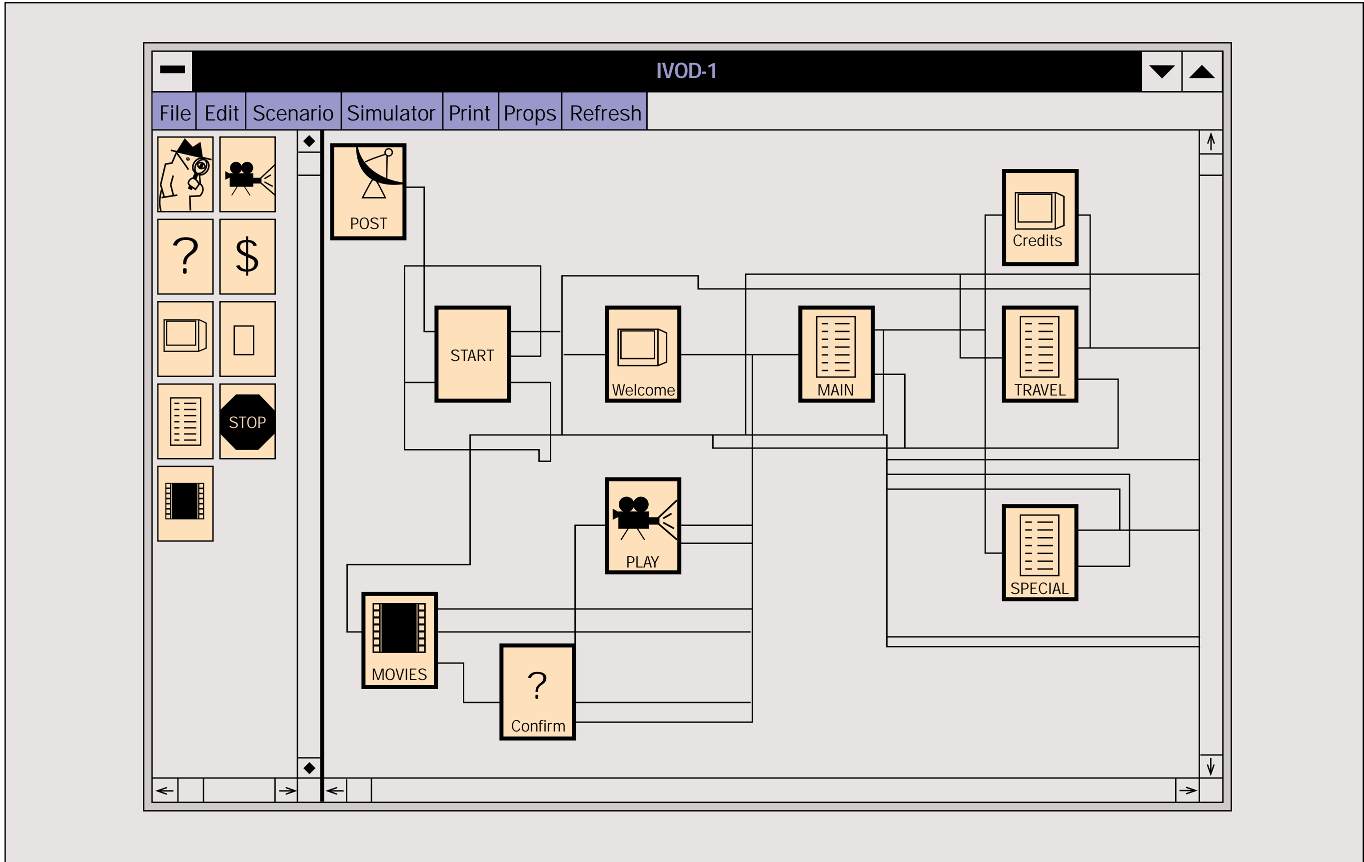

Figure 3. The domain vocabulary is represented using a palette of icons (upper left side of window) that instantiate, or change, into components when placed on the design area. Although the diagram resembles a logic circuit diagram, the "wires" do not represent absolute point-to-point "hard-wired" connections but, instead, potential paths for transmitting packets of information.

transmitting packets of information (event and data) between two or more operators. Each operator, implemented as a component, may itself be constructed from a set of lower-level operators.

Obviously, the components embodied in this example of the palette define the specific domain of video-on-demand. However, the palette for a different multimedia domain, such as video-messaging, will contain a certain number of the same components as video-on-demand, plus other components which differentiate this domain. However, at the platform level, the components will invoke the same capabilities, making this programming approach very versatile.

After considering several component composition approaches,[5,6] two types were prototyped: one using control flow diagrams, and the other using asynchronous logic diagrams. Both approaches lend themselves well to diagrammatic composition. Parallel composition constructs are more naturally represented in data flow diagrams[7] and enhanced state diagrams.[8] The asynchronous logic diagrams approach was chosen for our work because of the importance of asynchronism for multimedia and other telecommunications applications.

Some parallelism can be hidden from the users and processed during software synthesis. However, limited amounts of parallelism may have to be handled by the user. For instance, in the case of services that execute in a distributed environment, the designer may have to specify that the execution of three components may be performed

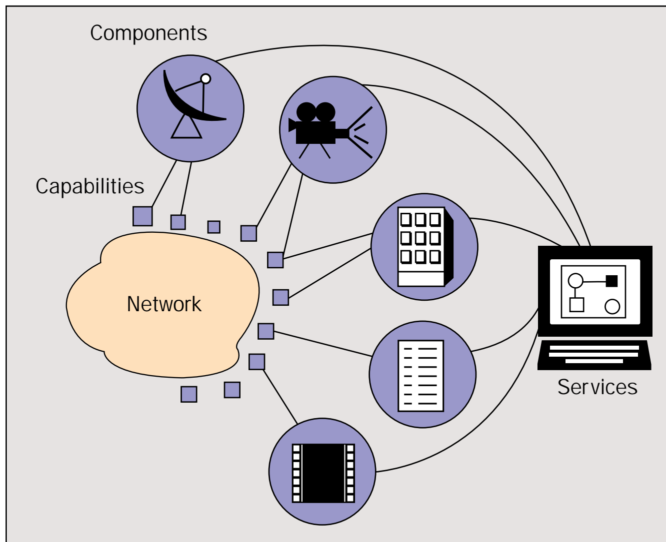

Figure 4. This network model shows the following functional mapping: network elements and resources that provide a set of capabilities and the basic functionality they provide to the network. Different capabilities can reside on separate physical network elements.

in any order but needs to be completed before the execution can continue to the fourth component. Asynchronous interruptability of a service by external events—such as a customer hang-up, incoming call signaling, or a conversation interrupt, which is another form of concurrency—can be entirely handled by the lower software layers, except that a SAW user might need to be aware of this.

A component accepts input events and generates output events upon the successful completion or failure of its tasks. A component may be realized using other components or it may internally use a control flow "diagram" to perform its tasks. In this approach there is no sequencing of actions: each operator processes events independently from the other operators involved in the service. Parallelism is inherent to the event-based interactions between operators.[7-10] This approach is well suited for voice, video, and multimedia domains where services frequently require the coordinated access to multiple resources from possibly disjointed network elements.

Building Block Mapping into the Network. The network model of Figure 4 shows the following functional mapping: network elements and resources that provide a set of capabilities and the basic functionality they provide to the network. Different capabilities can reside on separate physical network elements.

An intermediate level provides an abstract application domain set of functionality by combining into components capabilities from one or more network elements. This level is a *virtual functional domain*. Finally, services are constructed based on the component functionality as described in the assembly diagram.

Dynamic Software Architecture. The client-server architecture and the software layering provide the base for capturing the dynamic behavior of the software. The execution of the application software acts like a "conductor" who directs the sequencing of the resources necessary to perform the functions defined in the components. The component execution will, in turn, trigger the execution of the capabilities that compose it. In that sense, the visual software definition in the SAW is a representation of the dynamic software architecture of the system.

Software Simulation and Execution. An important characteristic of the SAW is the ability to seamlessly transition between software simulation and execution. This is a challenging goal that has no analogy in circuit design, and is possible in software—since software is present at all levels—with certain limitations and compromises.

In the SAW development environment, components can be independently simulated. The simulation of a service results from the composite and concurrent execution of small independent simulators that are "wired" according to the service layout diagrams.

Each simulator requires a private execution context, that is, an execution stack to run its virtual machine. This independent context insures that multiple instances of services can co-exist as independent processes competing for network resources. The resource contention is therefore handled at the lowest level in the components hierarchy and does not appear in the service assembly process.

When a service is requested, a database lookup is used to populate each required component with the corresponding data. In addition, the necessary connections are established between the low-level components and the requested communication ports to establish control of the network capabilities required by the service. Component connection occurs iteratively as part of the service initiation process. When all the components' data are populated, the service is ready to serve its customer.

Events are responsible for coordinating tasks and data exchange, or messages, between the constituent components of a service instance. Events are typically generated at component ports and are distributed, or dispatched, to subsequent components, where they are transformed and passed to the next layer. Component simulators are idle unless directly activated by an event. Component simulators process events concurrently and asynchronously.

The simulation and execution models are identical down to a certain level and diverge below it. The larger the common part, the larger the software reuse. However, this benefit must be balanced against the need to keep simulation efficient.

A Few Fundamental Issues

As mentioned earlier, the video-on-demand applications were constructed on top of a software platform that insulated the applications from the infrastructure. In our particular implementation, we chose to limit ourselves to a bounded offering—a bounded broadband network—because of the lack of ubiquity of ATM networks. However, as soon as broadband infrastructures are more widely deployed, it will be necessary to handle pragmatic issues related to two important characteristics of the infrastructure:

- Its distributed nature, and
- Its ATM nature.

The key issues are described below.

Resource Control. A number of resources must be controlled to provide a telecommunications service. Some of these resources reside in the CPE, while others are located in the network. The resources found in the CPE are generally oriented to the needs of a given user, and include such things as a speaker, a microphone, a mouse, and a screen with windows. Resources found in the network are generally oriented to meeting the requirement of interconnecting two or more users. These include such things as switch fabrics, transmission trunks, network announcements, and switch digit receivers.

A resource type might be located at either edge of the network or deep inside. Examples include bridges used to join two or more calls, storage devices to implement answering machines, and converters for adapting the format of user data, such as a National Television Systems Committee (NTSC) to or from the European standard for television, the phase alternate lines (PAL) format.

Besides physical resources such as these, there are also logical resources that have to be controlled. An important resource of this type is the address of the called party dialed by the calling party. This resource can be manipulated many times during the life of a call. It can be changed by services that provide speed calling, which might be located in the CPE or in the network, and it can be screened by services that block calls to a given set of addresses, again located in either the CPE or the network. It can also be used to route to more than one destination based on other factors, 800 numbers being an example.

An ACE has to be able to specify how a given service will control the wide variety of resources it uses to provide a service. And, to remain useful as technology changes, an ACE has to be able to add new resources to the list of resources it already knows how to control. For example, as multimedia becomes more ubiquitous and diverse, service creation environments will have to know how to control resources related to video, such as cameras and monitors.

In addition to providing the means to specify how a specific service will control the resources it needs to perform its function, a service creation environment—and the corresponding execution environment—also has to address the issue of how multiple services might interact while trying to control the same set of resources.

These interactions occur in many different contexts. They can arise between two or more services that execute within the context of the same call, such as a call forwarding service and a call waiting service, and they can arise within the context of the multiple threads of the same service executing simultaneously for different calls, such as a call distribution service that has to handle more than one active call.

In these two examples, the interactions occur as the result of requests from a single customer. Interactions also can occur when services execute simultaneously at the request of different customers. For example, two service execution threads might interact when trying to access the same resource, such as a bridge.

Impact of Asynchronous Transfer Mode. The introduction of ATM into the network impacts service software

in many significant ways. Sometimes this impact is positive in that it allows for the creation of new services, such as video-on-demand. But, this added flexibility of ATM comes at a price—it introduces additional complexity in the software controlling the service. Services now have to deal with issues that never occurred before, such as bandwidth, end-to-end delay, and jitter.

The issue of variability in the size of a connection is a good example of complexity that never had to be dealt with before, when all connections were of one size. With ATM, the size of a connection can change from one service to the next. And the size of a given connection can even change dynamically during the life of a call.

Service creators will be required to make new decisions about the size of a connection to be used, in addition to all the decisions they currently make. How a service determines the amount of bandwidth to request will depend on many factors, such as the cost of bandwidth given the time of day, the capabilities of each CPE on the call, and the quality of service desired by the users.

Additional complexity also occurs because one call will now be able to use more than one connection—or even no connections. Service creators will have to make decisions about adding and deleting connections dynamically. And they will have to make decisions about the correlation between connections, such as equalizing the end-to-end delay for two connections within a call.

In addition to the issue of resource management within a given call, there is also the issue of resource management for the entire system. Before ATM and multimedia, system level management was simplified by making the assumption that all calls roughly used resources in the same way, and system-wide resource management could be done just by looking at the number of calls.

With ATM and multimedia, this assumption no longer holds. Resource providers can no longer know whether they can support the next call by just looking at the number of existing calls. Instead, they will have to consider the level of resource usage, whether this usage is statistical or constant, and whether the existing set of calls, and their services, will change their level of resource usage. A service creation environment will have to support the ability to handle these new types of service decisions.

Conclusion

A method was presented for the construction of software using reusable components. The salient advantages of the method are:

- Software production speed is achieved by visual composition and simulation.
- Contrary to similar methods, software can be generated that controls multiple resources in a distributed network.
- Contrary to similar documented methods, the software generated is instrumented for various essential telecommunications tasks, such as dependability, revenue generation, and network measurements.

While the approach is promising for speeding up software design and has already been put in practice, several issues stand in the way of ubiquity:

- The ability to rapidly perform efficient domain analysis and derive the best set of components for a given domain.
- An opportunity to test in the field software techniques automatically provided to enhance system dependability.[11]
- A comprehensive platform that provides a generalized application programming interface.
- An efficient mechanism to instrument the software for security and revenue generation.
- A consistent approach to network and service management.

These issues are currently being addressed experimentally.

* Trademarks

LOTUS 1-2-3 is a registered trademark of Lotus Development Corporation.

MS-DOS is a registered trademark of Microsoft Corporation.

UNIX is a registered trademark of Novell in the United States and other countries, licensed exclusively through X/Open Company Limited.

Microsoft Office MS-DOS is a registered trademark of Microsoft Corporation.

Windows MS-DOS is a registered trademark of Microsoft Corporation.

References

1. J. Celentano, F. McInerney, and S. White, “Who’s Got the Money for these Multimedia Networks,” *Telephony*, Vol. 228, No. 22, May 29, 1995, pp. 32-36.

2. C. Jones, *Programming Productivity*, McGraw-Hill, Englewoord Cliffs, New Jersey, 1981.
3. Y. Levendel, "Reliability Models: Who Needs Them?" *Fourth Reliability Conference*, Invited Address, Denver-Colorado, 1991.
4. A. Reinardt, "The Networks with Smarts," *Byte*, Vol. 19, No. 10, October 1994, pp. 51-64.
5. B. A. Price, R. M. Baecker, and I. S. Small, "A Principled Taxonomy of Software Visualization," *Journal of Visual Languages and Computing*, Vol. 4, No. 3, September 1993, 211-266.
6. B. Selic, "An Efficient Object-Oriented Variation of the Statecharts Formalism for Distribute Real-Time Systems," *IFIP Conference on Hardware Description Languages and Their Applications*, April 26-28, 1993, Ottawa, Canada, pp. 335-366.
7. B. Lee, and A. R. Hurson, "Dataflow Architectures and Multithreading," *Computer*, Vol. 27, No. 8, August 1994, pp. 27-39.
8. D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, Vol. 8, North-Holland, 1987, pp. 231-274.
9. G. Agha, *Actors - A Model of Concurrent Computation in Distributed Systems*, The MIT Press, Cambridge, Massachusetts, 1986.
10. R. Chandra, A. Gupta, and J. L. Hennessy, "COOL: An Object Based Language for Parallel Programming," *Computer*, Vol. 27, No. 8, August 1994, pp. 13-26.
11. Y. Levendel. "Software Assembly Workbench: How to Construct Software Like Hardware?" *International Computer Dependability and Performance Symposium*, Erlangen, Germany, April 24-26, 1995, pp. 173-185.

(Manuscript approved July 1995)

__Eric R. Beyler__ is a technical manager of the Advanced Service Creation Platform Group in Network Systems Global Public Networks in Naperville, Illinois. His group is responsible for the development and application of new technologies for advanced network services. He joined the company in 1976. He has a B.S. degree in mathematics from Stanford University in Palo Alto, California, an M.A.T. degree in mathematics education from the University of Chicago, and an M.S. degree in computer science from Southern Illinois University in Carbondale.

__Olivier B. Clarisse__ is a member of technical staff in the Advanced Service Creation Platform Group in Network Systems Global Public Networks. He is responsible for visual creation, verification, and interactive simulation of multimedia services. He joined the company in 1986. He has an M.S.E.E. degree from Ecole Superieure d'Electricité in Paris, France, and a Ph.D. in electrical engineering from the Illinois Institute of Technology in Chicago.

__Edward A. Clark__ is a member of technical staff in the Advanced Switching Technology Group of Network Systems Global Public Networks, involved in investigating the impact of broadband multimedia services and protocols on call processing software. He joined the company in 1978. He has a B.S. degree in physics and a B.A. degree in mathematics from the University of Oklahoma in Norman and an M.S.E.E. degree from the California Institute of Technology in Pasadena.

__Y. H. Levendel__ is technology planning director of the Prototyping Platform Department in Network Systems Global Public Networks in Naperville, Illinois. He is responsible for the Global Public Networks Services Concept Center, which develops prototypes and customer pilot projects for broadband and narrowband telecommunications projects. He joined the company in 1976. He has a B.S.E.E. degree from Technion in Haifa, Israel, an M.S. degree in computer science from the Weitzmann Institute of Science in Rehovot, Israel, and a Ph.D. in computer engineering from the University of Southern California in Los Angeles.

__Robert E. Richardson__ is a technical manager of the B-ISDN Prototyping Platform Design Group in Network Systems Global Public Networks in Naperville, Illinois. His group is responsible for defining and prototyping advanced broadband multimedia control protocols and connection control and call control software. He joined the company in 1981. He has a B.A. degree in mathematics and economics from North Central College in Naperville, Illinois, and an M.S. degree in computer science from the University of Southern California in Los Angeles.