Specification and implementation of an ISO session layer

by A. Fleischmann
S. T. Chin
W. Effelsberg

This paper describes a novel technique for the specification and implementation of layered communication software. The technique is called Parallel Activity Specification Scheme (PASS) and is based on an extended-state machine model of protocol automata. It allows a convenient description of the communication behavior of concurrent systems and semiautomatic generation of programming language code from the specification. The first large-scale experience gained with this technique was in the specification and implementation of an ISO session layer. The code generation process and the embedding of the session code into a portable OSI operating system environment are described in detail.

In recent years, the International Organization for Standardization (ISO) Reference Model for Open Systems Interconnection[1] has gained increasing importance. It defines a framework for protocol standards, allowing different systems from different manufacturers to communicate with one another. Systems meeting the standards can exchange information with one another and are said to be "open" for communication with other systems.

In the ISO Reference Model for Open Systems Interconnection, complex communication protocols are structured in layers, and each layer is further structured in entities. An entity uses the services of the layer below for communicating with another entity in the same layer but in a different system. This communication follows strict rules, called a *protocol.* An important part of a protocol standard is the exact specification of the allowable sequences of events in time. For example, in a connection-oriented protocol, a data transfer request is allowed only after a connection has been established. An attempt to send data before establishing a connection results in an error message.

In an ISO or Comité Consultatif International Télégraphique et Téléphonique standards document, the allowed sequences of events are usually described in the form of a state/event table. Translating this table into code is a significant undertaking. The state/event table in standards is only a semiformal description of the protocol machine. Thus, a tool to assist the programmer in translating a standards document into code would be useful, and has been developed at the IBM European Networking Center (ENC). ISO is currently considering the formal description languages LOTOS and ESTELLE for this purpose. This tool, called the Parallel Activity Specification Scheme (PASS),[2] can be used for specifying any system of parallel processes (e.g., communication systems, process control systems, etc.). The first practical experience using PASS was gained with the implementation of an ISO Session Layer (Layer 5 in the Reference Model).

This paper describes the PASS technique and its use in the session layer implementation. We introduce the basic concepts of PASS, give an overview of the ISO session layer service and protocol, and describe the specification and implementation of the session layer with PASS. An operating system environment for OSI software is then introduced, and it is shown in detail how the generated session layer code can be embedded into this environment.

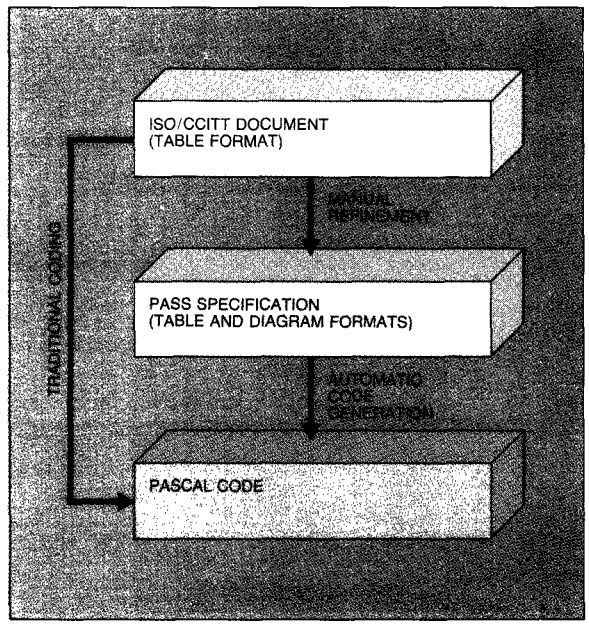
PASS: A specification technique for parallel processes

PASS is based on extended finite-state machines. *Basic finite-state machines* can only store status information implicitly, as a consequence of being in a particular state. *Extended finite-state machines* allow the declaration of variables to store information. For example, the sequence number of a current synchronization point or the number of unacknowledged frames could be kept in a variable. This extension is essential for the practical use of state machines for protocol description.

When PASS is used for OSI software, it helps the programmer to produce code from a standard document in a disciplined way. Without PASS, coding would be done directly from the ISO or CCITT document (traditional coding). With PASS, an intermediate step is introduced. The PASS description of a protocol is more detailed than the original standard document. All ambiguities have been removed, and intermediate states have been introduced that are not visible at the level of abstraction of the state/event table in the standard. The PASS specification contains enough detail to produce code semiautomatically. A code generator can be written for sequential or parallel programming languages. Our implementation of the code generator produces Pascal code. The role of PASS in the OSI software development process is shown in Figure 1.

The PASS graph refinements contain the local variables of a process, the operations, and the functions defined on the local variables.

Figure 1 The role of PASS in the OSI software development process



Outline of PASS. A system described in PASS consists of a set of processes communicating with one another via messages. Each process has a unique name, and the number of processes in a system is static. The messages have names (message name) and parameters (message parameters). Messages with the same name are of the same message type. In PASS, the description of a process consists of a PASS graph and a PASS graph refinement.

The *PASS graph* describes the sequences in which a process sends messages, receives messages, and performs internal functions and operations. The PASS graph is described by *nodes* and *edges* between the nodes. The nodes correspond to the *main states* and the edges to the possible *transitions*. The arcs (edges) point from the starting state (node) to the successor state. The edges are marked with the event causing the transition.

The PASS graph refinements contain the local variables of a process, the operations, and the functions defined on the local variables. (*Operations* change

the values of local variables, and *functions* leave them unchanged.) There are four different types of functions and operations. For each message type that can be received by a process, there exists a *receive-message specification* that describes the message parameters and the effect of a received message on the values of the local variables. For each message type that is sent by a process, there is a *send-message specification* that describes the message parameters and how the values of the message parameters are determined from the values of the local variables.

An *internal operation specification* describes how an internal operation changes the values of the local variables. Besides these changes, an internal operation can yield different results. For example, a PUSH operation on a stack of limited depth can have the result *done* or *stack full*. The internal operation specification also contains the possible results. An *internal function specification* shows the possible results of the internal functions in dependence of the values of the internal variables. The values of all local variables define the *local state*.

The execution of transitions from one main state to another can have four reasons. A transition can be triggered by sending messages, receiving messages, results of internal functions, and results of internal operations. Accordingly, in a PASS graph, four types of nodes can be distinguished. There are two types of communication nodes—send nodes and receive nodes—and there are two types of internal nodes—internal function and internal operation. The graphics representation of the four node types is shown in Figure 2. A complete PASS diagram (a PASS graph) describes the state-transition behavior of a process and all of its external interactions, i.e., all the messages it will send or receive. Figure 3 shows an example of a PASS graph.

A *send node* corresponds to a main state where one process wants to send a message to another process. The send operation can be *synchronous*; that is, the sending process can continue only when the receiving process is in a receive main state where it accepts a message of this type. Otherwise the send operation is an *asynchronous* one.

For asynchronous sending, PASS provides a buffer mechanism (i.e., input pools). The finite size of these pools must be declared explicitly, and the maximum number of buffered messages must be declared. If the pool of the receiving process is full, the sending process is blocked until at least one slot in the input

Figure 2 The four node types of PASS

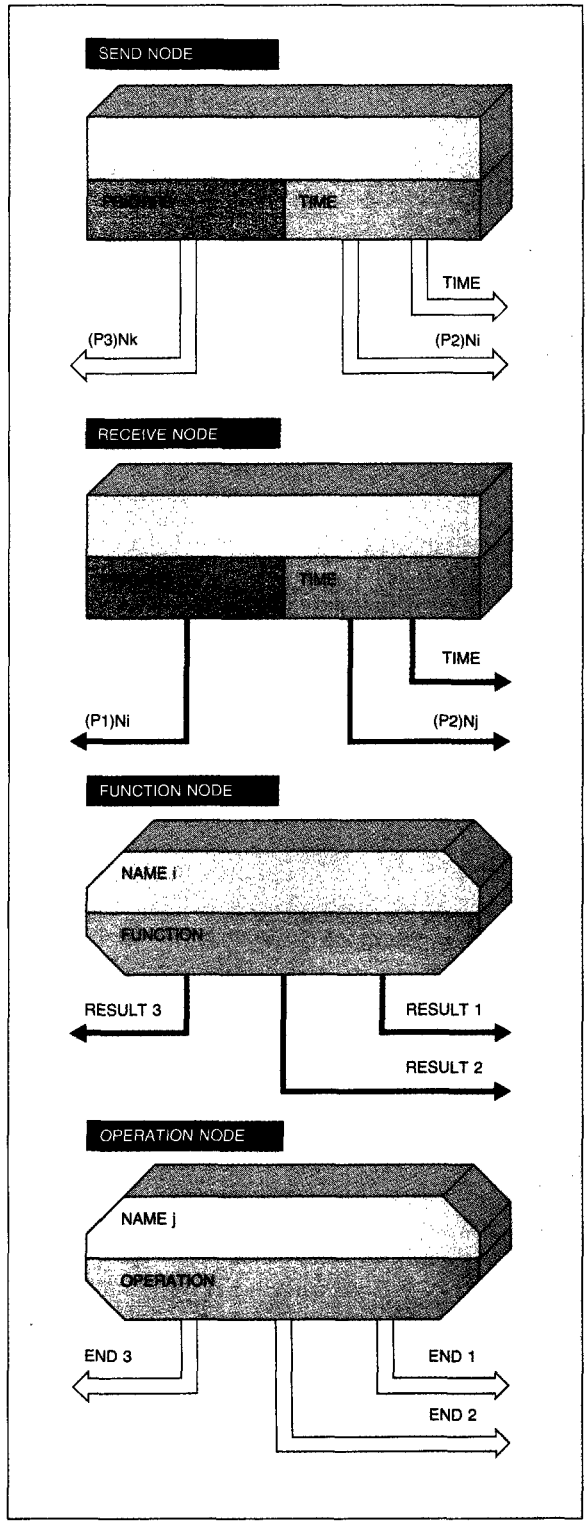
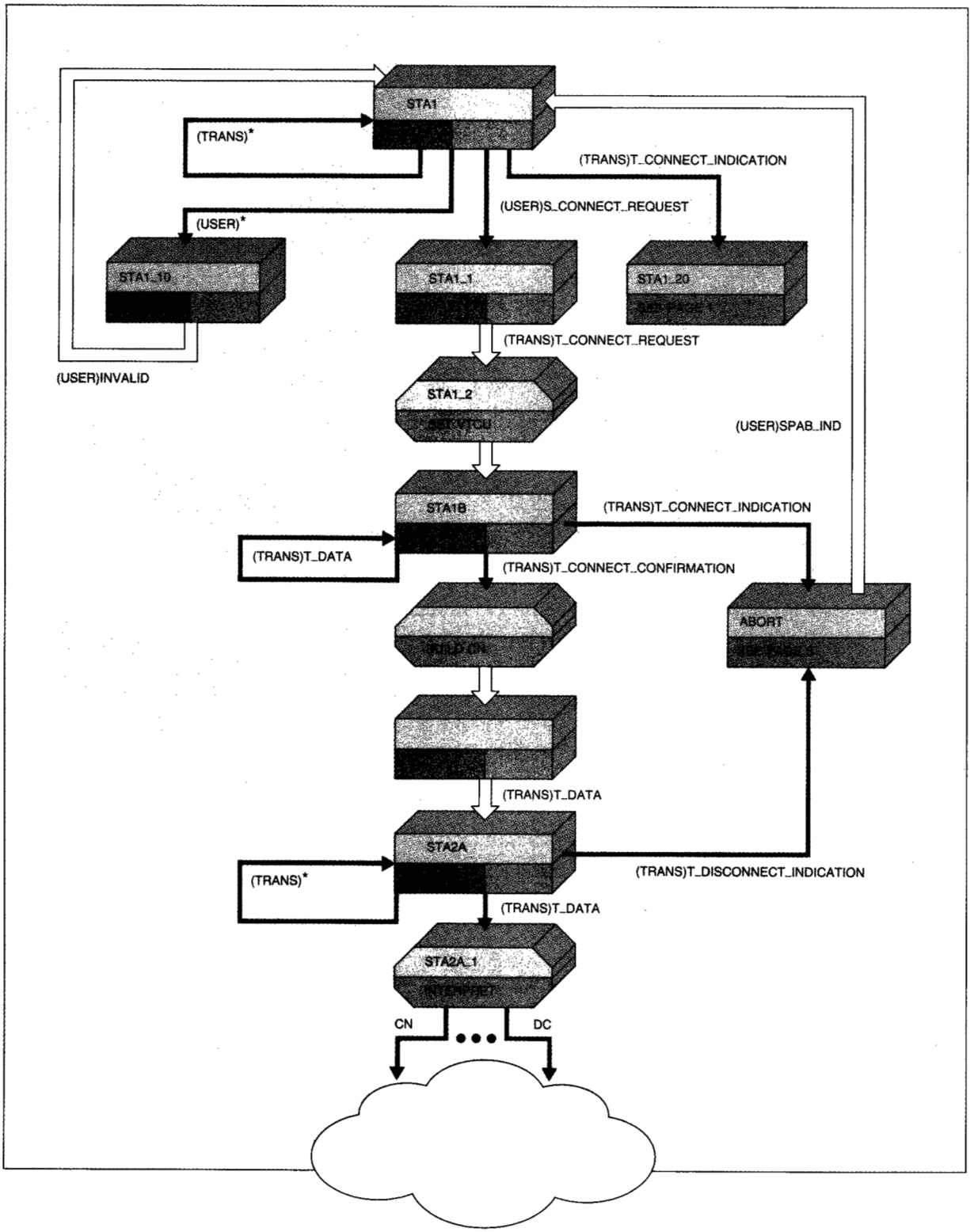


Figure 3 An example of a PASS graph



pool becomes free. The size of the input pool is a property of the receiving process. Therefore, the receiving process determines whether a message is sent synchronously or asynchronously.

If a process wants to send different messages alternatively, send nodes can have multiple outgoing edges that are marked with the message type and the name of the process that will receive this message. Depending on the message to be sent, the process performs the corresponding transition. If a send node has two or more outgoing edges, and two or more messages can be sent, one of the possible transitions has to be chosen. For this purpose, a priority list can be defined. The transition with the highest priority is then executed. If a process cannot send any message, and infinite blocking must be avoided, a time-out edge can be provided. If a process cannot send, the time-out transition is executed.

The graphics representation of a send node is a box with heavy-lined arrows for the transitions. A priority list can be entered in the lower left half of the box. A time-out transition is represented by an edge marked "time," and the timer value is entered in the lower right half of the box. In the PASS graph shown in Figure 3 state STA1_1 is a send state. In this state, the corresponding process wants to send the message T_Connect_Request to the process Trans.

A *receive node* corresponds to a state where a process expects a message from another process. If a process has an input pool and if the expected message is in the input pool, the transition to the next state is executed. If a process has no input pool but the expected message is offered by the corresponding process, the transition to the successor state can be performed. If the expected message is not in the input pool (receiving process with input pool) or if it is not offered by the corresponding process (receiving process without input pool), the receiving process will be blocked. The receiving of different messages—possibly from different processes—is allowed in one receive state and a corresponding number of edges leave the state. The outgoing edges are marked with the message type and the name of the sending process from which the message is expected.

If a receive node has two or more outgoing edges, and two or more messages can be received, one of the allowable transitions has to be chosen. For this purpose, a priority list can be defined, and the transition with the highest priority is then executed.

When none of the expected messages arrives, the receiving process is blocked. In order to avoid permanent blocking, a time-out mechanism is provided in PASS. A time-out transition is specified with its own edge similarly to time-out edges in send states.

The graphics representation of a receive node is a box with single-line arrows as the outgoing edges. The type of the expected message and the name of the sending process are marked on each edge. A priority list can be entered in the lower left half of the box. A time-out transition is represented by an edge marked "time," and the timer value is entered in the lower right half of the box.

In the PASS graph shown in Figure 3, state STA1 is a receive state. If in this state the process receives the message S_Connect_Request from the process User, the transition to state STA1_1 is executed. If the process receives any other message from the process User (the symbol * in Figure 3 means any other message), the transition to state STA1_10 is performed. In state STA1 the message T_Connect_Indication from the process Trans causes the transition to state STA1_20. All other messages from the process Trans are thrown away, i.e., the transition marked with (Trans)* is executed.

An *internal function* corresponds to a main state where a process evaluates local variables. Because PASS is based on extended finite-state machines, it must be possible to evaluate the status of local variables and make transitions based on their values. For example, process execution can depend on the contents of an arriving message. Because a receive node distinguishes only types of arriving messages, not their contents, message contents are evaluated in a subsequent internal function node. Another example for an internal evaluation is a counter for the number of unacknowledged frames. Because the result of an internal evaluation is deterministic, exactly one transition will be executed. There are no priorities, and there is no time-out.

The graphics representation of an internal function is an oval with single-line arrows for the transitions. The lower half of the oval contains the name of the internal function. The upper half contains the optional state name. Each outgoing edge is marked with the result of the variable evaluation that leads to this transition. In the example, state STA2A_1 is an internal function state. In this state, the internal function Interpret is executed. This function can

have several results, depending on the state of the internal variables. The results CN and DC and the corresponding transitions are shown in Figure 3.

An *internal operation* corresponds to a state where a process assigns new values to local variables. Depending on the computed new values, different transitions to successor states can be made. As for internal functions, the internal computation is deterministic. Therefore, there are no priorities and there is no time-out.

The graphics representation of an internal operation is an oval with heavy-lined arrows for the transitions. The lower half of the oval contains the name of the internal operation. The upper half contains an optional state name. Each outgoing edge is marked with the outcome of the internal operation that leads to this transition. A common case for an internal operation is to provide for an error exit.

In the example, state STA1_2 is an internal operation state. In this state, the internal operation Set_Vtcu is executed. If an internal operation can always be performed and always has the same effect on the values of the local variables, then an internal operation state has exactly one successor state. The transition to the successor state means that the internal operation is performed with the only possible effect. In such a case, the marking for the corresponding arc can be omitted.

In the current version of PASS, the graph refinements are described in the Pascal language.

In addition to specifying the communication behavior of a process, it is necessary to describe the functions and operations in a formal language. Because functions and operations are purely sequential, any formal language for sequential processes may be used. This part of a PASS specification is called *PASS graph refinement*. In the current version of PASS, the graph refinements are described in the Pascal language, which was chosen for two reasons. The language is very widely used, and the current version of the code generator generates Pascal code for the PASS graph part. Thus it is very straightforward to integrate the PASS graph refinements with the code generated for the PASS graph.

PASS is a specification language for parallel processes.

A comparison of PASS with ESTELLE and LOTOS. PASS is a specification language for parallel processes. Communication protocols are only one example of its use. For the specific purpose of describing communication protocols, other specification languages have been developed. In particular, ISO is currently working on the standardization of two formal description techniques for protocols, ESTELLE[3] and LOTOS.[4] In this section, some aspects of these ISO languages are described and compared with PASS.

In all three specification techniques (ESTELLE, LOTOS, and PASS), a system consists of communicating processes. In each of these specification techniques a process[5] can be defined in the following two steps:

- Communication mechanisms, through which the processes communicate with one another
- Communication behavior, which is the relationship between inputs and outputs in terms of the order in which inputs and outputs may occur, and their value dependencies

Communication mechanisms. In ESTELLE, processes are a class of *modules*, and the abstract mechanism defined for communication between processes is an *interaction*. Interactions are exchanged via interaction points. A queue (FIFO buffer) is associated with each interaction point. The queue stores interactions received from processes. Because these queues have an infinite length, a process is never blocked when it sends an interaction (i.e., a message). This kind of

message exchange is called *asynchronous*, which implies that the receiver cannot prevent the sender from putting data in the queue. Consequently, in ESTELLE, a formal description of back pressure is not possible without introducing an additional message.[5] With this message, the process tells its partners how many messages they can deposit in its queues (*credit*). After sending the maximum number of messages, they have to wait for new credit.

Because of the use of FIFO message queues, additional interaction points (*queues*) have to be introduced in an ESTELLE specification to allow priority messages to pass the messages in the normal queue.

In LOTOS, a similar interaction concept is used. The atomic form of a LOTOS interaction is an *event*, which is a unit of synchronized communication that may exist between two processes. An event will only occur if both involved processes are prepared to engage in the event by making the appropriate event offer (synchronous communication). Thus communicating processes are coupled tightly. Without introducing additional processes to execute buffering functions, asynchronous communication is not possible. An example of a buffering process can be found in Reference 6.

In LOTOS, priorities for messages are not allowed. If more than one event is eligible for execution, one is chosen at random. Thus, one event cannot be described as having a higher priority, which is an important deficiency in process control applications. For example, an alarm message might be more important than all other offered events.

In PASS, both synchronous and asynchronous communication are possible.

In PASS, both synchronous and asynchronous communication are possible, as described earlier. An input pool size of zero corresponds to the synchronous message exchange. The finite size of an input pool makes it easy to describe back-pressure policies. The possibility of input-pool structures[2] allows one to define for each message individually whether it is to be sent/received synchronously or asynchronously.

PASS provides message priorities that make it easy to describe the behavior of processes in a conflict. Because the priorities of messages can be different in different states (priority field), the priorities can be used flexibly.

Communication behavior. To model the dependency of outputs upon inputs, ESTELLE uses an extended finite-state-machine model. Starting from a predetermined initial state, a process makes transitions from one state to the next. A transition is normally triggered by an input, and outputs may or may not be produced during a transition. In ESTELLE, the state space is spanned by the values of local variables. One of these variables is called STATE, or the *major state variable*, and the others are sometimes called *minor state variables*, or *context* variables. All variables are typed according to Pascal conventions, thereby allowing the use of conventional Pascal operations on these types. In ESTELLE, the operations executed during a transition are independent of the triggering input. This means that the same message accepted in different major states can have different effects on the context variables. The description of the communication behavior is spread out over the specification document. This makes it difficult to determine the allowable input/output sequences.

LOTOS defines a set of temporal operators to model the order in which events may occur and their value dependencies. LOTOS is based on Milner's Calculus of Communicating Systems;[7] examples of processes these operators allow include

- Sequential composition
- Nondeterministic choice
- Parallel composition of processes
- Execution disruption of one process by another

In LOTOS, the communication behavior is described with *behavior expressions* that describe observable sequences of events. The behavior expressions can be parameterized, and recursion extends the model to transition systems where the number of states may be infinite. In LOTOS, states do not exist explicitly. Instead of sequences of states (as in ESTELLE), sequences of transitions are considered.

The model used for PASS is similar to the model used for ESTELLE in that the contents of the variable STATE in ESTELLE correspond to the nodes in the PASS graph, and the context variables in ESTELLE correspond to the local variables in the PASS graph refinements. In ESTELLE, a transition cannot be triggered by an output. PASS allows inputs (receive messages) and outputs (send messages) for triggering transitions, which is similar to the procedure in LOTOS.

In PASS, receiving a message always has the same effect on the local variables and is independent of the state in which it is received. In ESTELLE, the effect of a message (interaction) depends on the state in which it is received. If the same type of message can be received in different ESTELLE states, different effects can be specified. In LOTOS, receiving a message only has the effect of copying the values of the message parameters into local variables. In PASS, the communication behavior is described separately (i.e., by a PASS graph) from the effect of a transition on the local variables (receive) or message parameters (send).

It was easy to implement a generator that produces Pascal code for the PASS graph.

In ESTELLE, Pascal statements are used to describe the transitions. This is very close to an implementation. In LOTOS, an abstract data-type language called ACT ONE[8] is employed as a sublanguage to describe the data-value domains and operations on variables. For the PASS graph refinements, any appropriate specification technique can be used. Depending on the stage of a specification, natural language, a formal technique, or a programming language can be employed.

The semantics of PASS graphs are simple. Therefore, it was easy to implement a generator that produces Pascal code for the PASS graph. If the PASS graph refinements are also described in Pascal, the two parts can be combined to form a complete implementation of a process. This is discussed in the next section. Even though ESTELLE and LOTOS are both intended for international standardization, we believe there is a place for PASS because PASS is easy to understand and use, and because PASS is suitable for automatic code generation.

The code for a PASS process consists of the code for the communication behavior (PASS graph) and the code for the transitions (PASS graph refinements). This makes the code easy to understand and to debug.

Translating PASS into Pascal. A PASS specification is a formal description of the behavior of a process, including its communication control (message passing and finite-state machine) and its internal sequential actions. For the automatic generation of programming language code, each of these parts is considered in turn.

The message-passing operations for a certain message are translated into corresponding procedure calls. If an input pool is specified in PASS, the corresponding procedures will provide a buffer for the required number of messages. For each send/receive edge for a specific message type, a Pascal procedure call for the corresponding procedure is generated. The final mapping of the send and receive operations to operating system primitives depends largely on the operating system and network environment. Therefore, the complete body of these procedures cannot be generated automatically.

For the internal sequential actions, no code generation is required because those actions are already specified in Pascal.

The generation of code for the finite-state machine can be done in two ways: Either the state/event table (a sparse matrix) is stored in some internal format

and interpreted at run time, or the states and transitions are compiled into sequential pieces of code for which, typically, each piece of code has a label and corresponds to a state/event combination. After the action in a node has been performed, a GOTO to the successor piece of code is computed dynamically. In the current version, the code generator is of the second type; that is, it produces pieces of sequential Pascal code for each state/event combination, a point that is discussed in more detail later in this paper. We are studying a generator of the first type (state/event table that is interpreted at run time) in order to compare the code produced by the two approaches.

For the internal sequential actions, no code generation is required because those actions are already specified in Pascal. The generator simply copies them as procedures into the Pascal output file.

At the IBM European Networking Center in Heidelberg, a complete tool set is being developed for PASS. In addition to the code generator for Pascal, there is a code generator for Modula-2 on the PC and an interactive, screen-oriented PASS editor for easy data entry and modification of PASS specifications. There is also a set of consistency-checking routines for analyzing a PASS graph. Thus, unreachable states can be detected in the early design phase.

Having introduced PASS, we now discuss briefly the ISO session layer and the use of PASS for the session layer implementation.

The ISO session layer protocol

Protocols are rules that govern the exchange of data in computer networks. Layered protocols result from the separation of functions, wherein each protocol layer uses the services provided by the layer below to provide its service to the layer above. A framework for layered communication protocols was developed by the International Organization for Standardization; it is known as the *Reference Model for Open Systems Interconnection.*[1] Figure 4 shows the seven protocol layers of the ISO/OSI Reference Model and their main functionality. The session layer, which is layer five in the Reference Model, provides services for synchronizing the communication of session users.[9,10] The session layer provides means to perform the following functions:

- Establish a connection with another session user, exchange data with that user in a synchronized manner, and release the connection in an orderly manner
- Negotiate for the use of tokens to exchange data, synchronize and release the connection, and arrange for half- or full-duplex data exchange
- Establish synchronization points within a dialogue and—in the event of errors—resume the dialogue from an agreed-upon synchronization point
- Interrupt a dialogue and resume it later at a prearranged point

The services of the session layer are separated into *functional units*, which are logical groupings of related services.

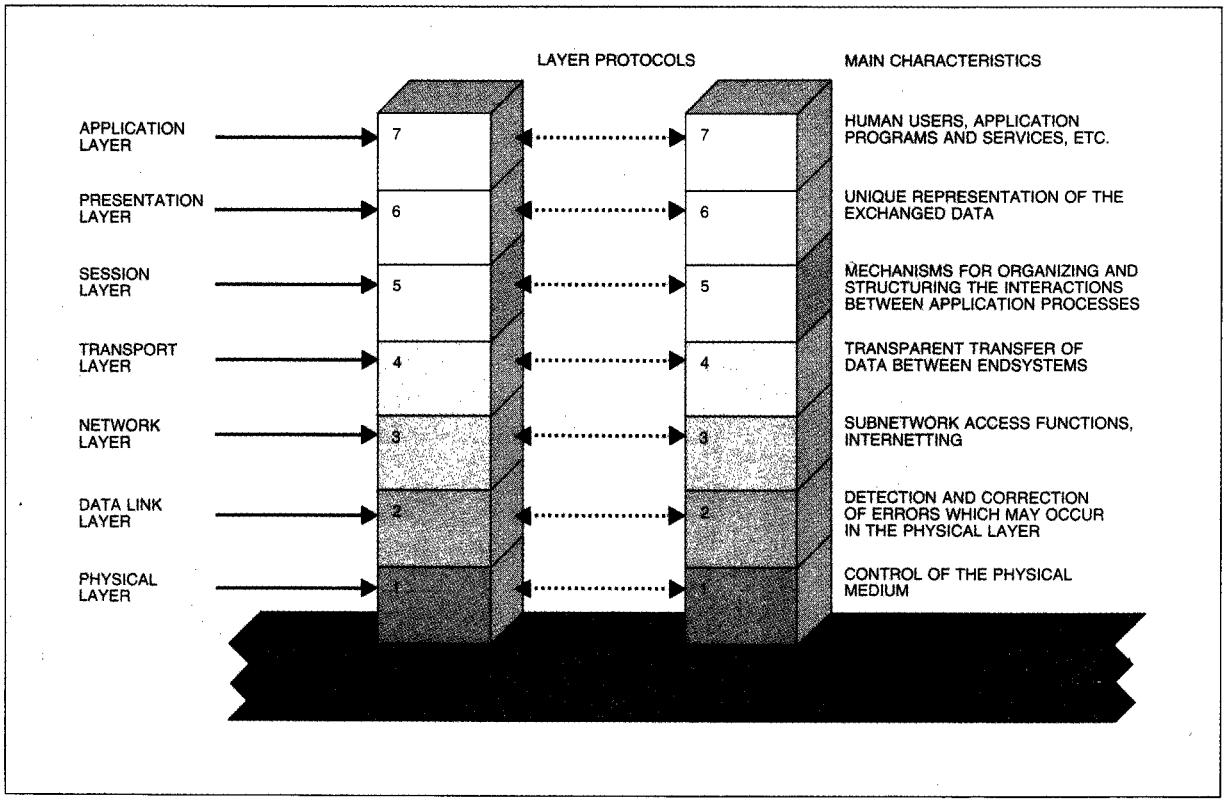
The kernel functional unit supports the basic session services required to establish a session connection, transfer normal data, and release the session connection. This functional unit must be available in each session implementation. Optional functional units relate to the following aspects of services:

- Negotiated release
- Half-duplex
- Duplex
- Typed data
- Expedited data
- Capability data exchange
- Minor synchronization
- Major synchronization
- Resynchronization
- Exceptions
- Activity

During the connection establishment phase (service primitive S-Connect-request), the communication partners negotiate the functional units to be used during the connection. Therefore, it is possible for session entities with different sets of functional units to communicate with one another, provided they can agree on a common subset of session functional units they both support.

For synchronizing the communication between two session users, the session provides the ability to separate the data stream into logical units. Activities are one kind of logical unit. When a user starts an activity, a *name* is assigned to it, and the activity can be discarded, interrupted and continued later, or finished without interruption. The name of an activity is important for continuing an interrupted activity, because the user must identify which activity should be continued.

Figure 4 The ISO Reference Model for Open Systems Interconnection



Major and *minor synchronization* points can also be used for separating a data stream. The main difference between major and minor synchronization points is that an entity stops sending data until a requested major synchronization point is confirmed, whereas an entity can continue sending data after a minor synchronization point has been requested. If a rollback is necessary, the data retransmission starts at a previously confirmed major or minor synchronization point.

Another important concept in the session layer is that of *tokens*. A communication entity can perform certain service requests only if it owns the corresponding tokens. The following four tokens have been defined:

- Data token
- Release token
- Synchronized minor token
- Major/activity token

Service primitives are provided to request and give tokens. For example, consider the data token. This token exists only if the session uses the half-duplex functional unit, and only the entity owning the data token can send data. If a user wants to send data and his session entity does not own the data token, he can ask his partner for the data token by using a *please-token service*. A user can give a token to his partner via a *give-token service*. During the connection establishment phase, the initial distribution of the tokens is negotiated.

Implementation of an ISO session layer with PASS

Because communication software is system software, it depends to a large extent on the operating system environment in which it is executing. For example, interrupts from the network must be passed on to the communication software, and hardware timers are used for time-outs. On the other hand, it is

desirable to write communication software in a portable way. In addition to reducing the coding effort, portable code guarantees compatibility of the protocol implementations on different systems. Also, conformance testing need be done only once. The inherent problem is to write portable communication software for computer systems whose hardware and operating system are typically very different.

This can be done by providing an operating system environment for the specific needs of OSI software. Its functionality includes buffer management, timer management, terminal I/O interface, etc. Instead of using the real operating system functions directly, the OSI layers use these services only indirectly through this environment. The code of all OSI layers thus becomes independent of the operating system. Only the environment must be ported to the various real operating systems and hardware architectures.

Pascal was chosen as the implementation language for the session layer.

Our session layer implementation is based on this approach. The PASS specification and code generation are applied to the session layer functionality only. The result is a session protocol machine (SPM) that is independent of the operating system. The PASS language and tools were developed at the IBM European Networking Center in Heidelberg, and the experimental operating system environment for OSI software and a session interface process (SIP) for it were designed and implemented at the IBM communication software development laboratory in Palo Alto, California.

The session protocol machine is specified as one PASS process. A PASS specification of a process consists of a PASS graph and the PASS graph refinements. These are also the main parts of the session protocol machine implementation.

Implementation of a PASS graph. Pascal was chosen as the implementation language for the session layer. Therefore, this paper discusses only the transformation of a PASS graph into Pascal. However, a PASS graph may be transformed into other types of programming languages.[2] Because a complete mapping of the PASS semantics into Pascal is very difficult, the following restrictions were made:

- Each process has an input pool so that there are no synchronous message exchanges.
- Each process sends only one message in a send state so that there is no alternative sending.

In the special case of communication protocols, these restrictions have no impact because processes with input pools and send states with only one possible message are the most common cases.

There are two main considerations in transforming the PASS graph into a Pascal program: (1) the implementation of message passing (send/receive), and (2) the simulation of processes in a sequential programming language.

The implementation of message passing. Because of our restriction that only processes with input pools are allowed, message passing is implemented via a buffer with the following two functions associated with it: (1) A *write function* is invoked by a process that wants to send a message to the owner of that input buffer. The write function deposits a message into the buffer, and a return parameter shows whether it was possible to put the message into the buffer. (2) A *read function* is invoked by the process that owns the input buffer. A parameter of the function contains the type and sender of an expected message. If this expected message is in the input buffer, the message is transferred to the reading process and is removed from the buffer. If the message is not in the buffer, a corresponding code is returned to the reading process.

The simulation of processes in a sequential language. The PASS graph of the session corresponds to one Pascal procedure. The invocation of this procedure means that the session process gets control. In the procedure, each send state corresponds to a piece of code similar to the one shown in Figure 5. Each receive state corresponds to a piece of code similar to the one shown in Figure 6.

A full session layer at run time consists of a number of processes, each corresponding to an active session

Figure 5 Code structure for a PASS send node

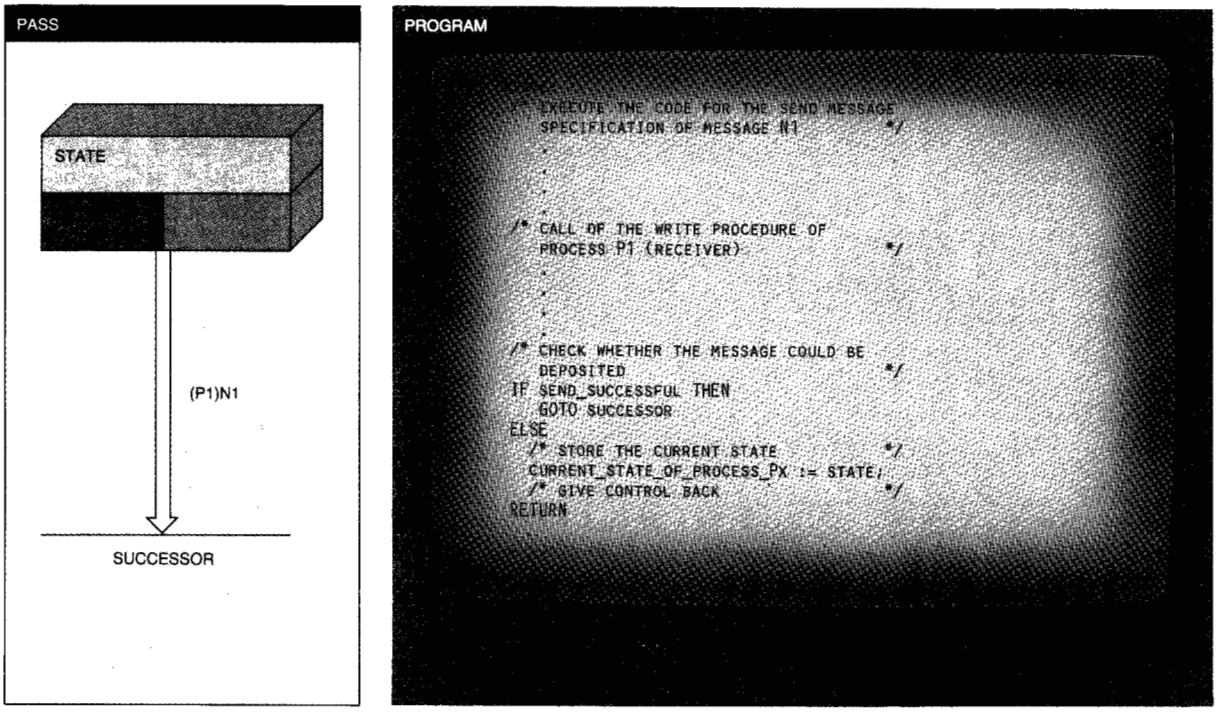
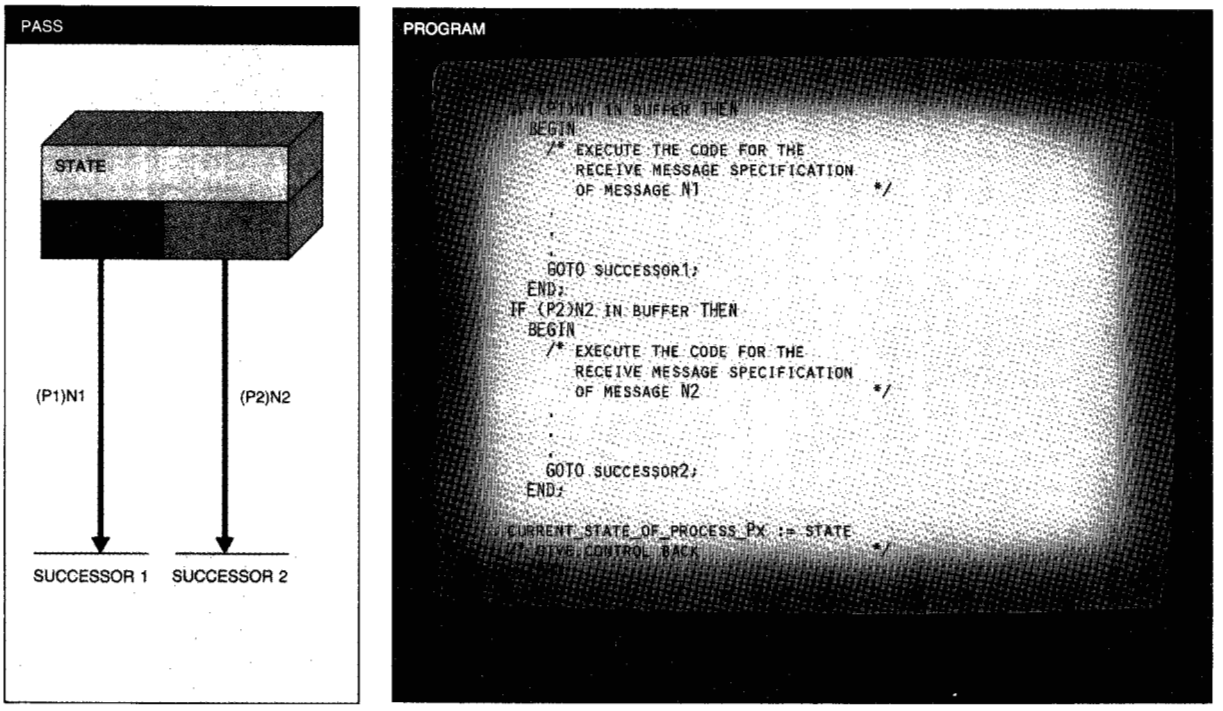


Figure 6 Code structure for a PASS receive node



connection. A new process instance is created when a new connection is established. Each process executes the PASS graph procedure code. The invocation of such a *process procedure* means that it must resume execution where it last gave up control. (This is the concept of *coroutines*.) In the context of a finite-state machine, the PASS graph process must be continued in the current state, which is stored in a variable called "current_state_of_Px" in the examples. Therefore, the first statement in a process is a branch to the code piece corresponding to the current state. Figure 7 shows the structure of this process procedure.

The main program, which gives control to the different process procedures, contains the scheduling strategy and is part of the OSI-specific operating system environment, not the session protocol machine.

We have illustrated the basic principle for converting the PASS graph into a program. For the session implementation, this technique was improved in order to optimize performance, but the principle remains the same.

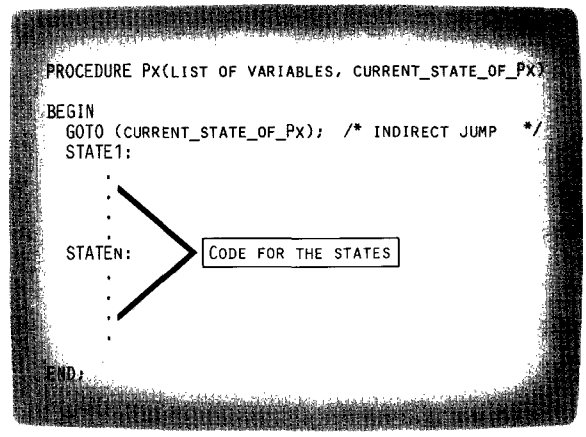
We chose Pascal for portability and easy integration with the code for the PASS graph.

Implementation of the PASS graph refinements. We have mentioned that the PASS graph refinements can be specified in any language for sequential programs. We chose Pascal for portability and easy integration with the code for the PASS graph.

The internal functions and operations of the PASS graph refinements for the session protocol machine can be categorized as follows:

- *Predicates* are conditions based on internal variables, such as "synch point number reaches 999999"; predicates are specified in the session standard.
- *Actions* are protocol actions, such as "increment synch point number," and are also specified in the session standard.

Figure 7 The structure of a process procedure



- *Conversion routines* convert service element parameters into protocol data unit parameters, and vice versa.

These functions are specified in the form of Pascal subroutine procedures.

The code for the entire protocol machine is produced automatically by the generator. The generator reads the PASS graph (in a table format) and also reads the PASS graph refinements (already written in Pascal) as an input and produces a complete Pascal program for the session protocol machine as an output.

The generator is also able to produce a session protocol machine containing any subset of session functional units, and it accepts the names of the functional units as a parameter. The generator produces an internal PASS graph corresponding exactly to these given functional units, and then uses this reduced PASS graph to generate code. Thus, it is easy to tailor the session protocol machine to specific applications.

Embedding the session layer into an operating system environment

In the previous sections, we have shown that the OSI session standard protocol may be specified unambiguously by using the PASS technique. We have also seen how high-level language code may be generated directly by automatic means from the PASS description.

In this section, we introduce an OSI implementation environment that supports the development of layered systems of communication protocols. We identify the implementation-dependent issues in designing an OSI system and describe how these issues are solved in this environment. We then describe the embedding of the session layer into this environment.

An OSI implementation environment. In our OSI implementation environment, a layer implementing an OSI standard is supported by many system components protecting it from the particulars of a native operating system. In the following, this OSI implementation environment is called the *Base*. The Base provides the layers with a set of operating system services that constitute the kernel of a modern layered communication system. These services include work-request management, control-block management, buffer management, timer management, and message-log service. The Application Program Interface (API) provides an interface between the native operating system application and layers supported by the Base. The Network Interface Sub-Layer (NISL) provides an interface between the native operating system network device handlers and layers supported by the Base. Systems Management provides to the layers the system- or network-management-related services such as the directory service. Figure 8 shows the structure of the experimental OSI environment implemented at the IBM laboratory in Palo Alto, California. The figure shows only a transport and a session layer, but any number of layers may be included in the system.

System control block structure. The experimental OSI implementation architecture defines the following control block structure, which closely resembles the ISO/OSI Reference Model: A layer is controlled by a Layer Control Block (LCB); a service access point is controlled by a Service Access Point (SAP) Control Block; and a connection end point is controlled by a Connection Control Block (CCB).

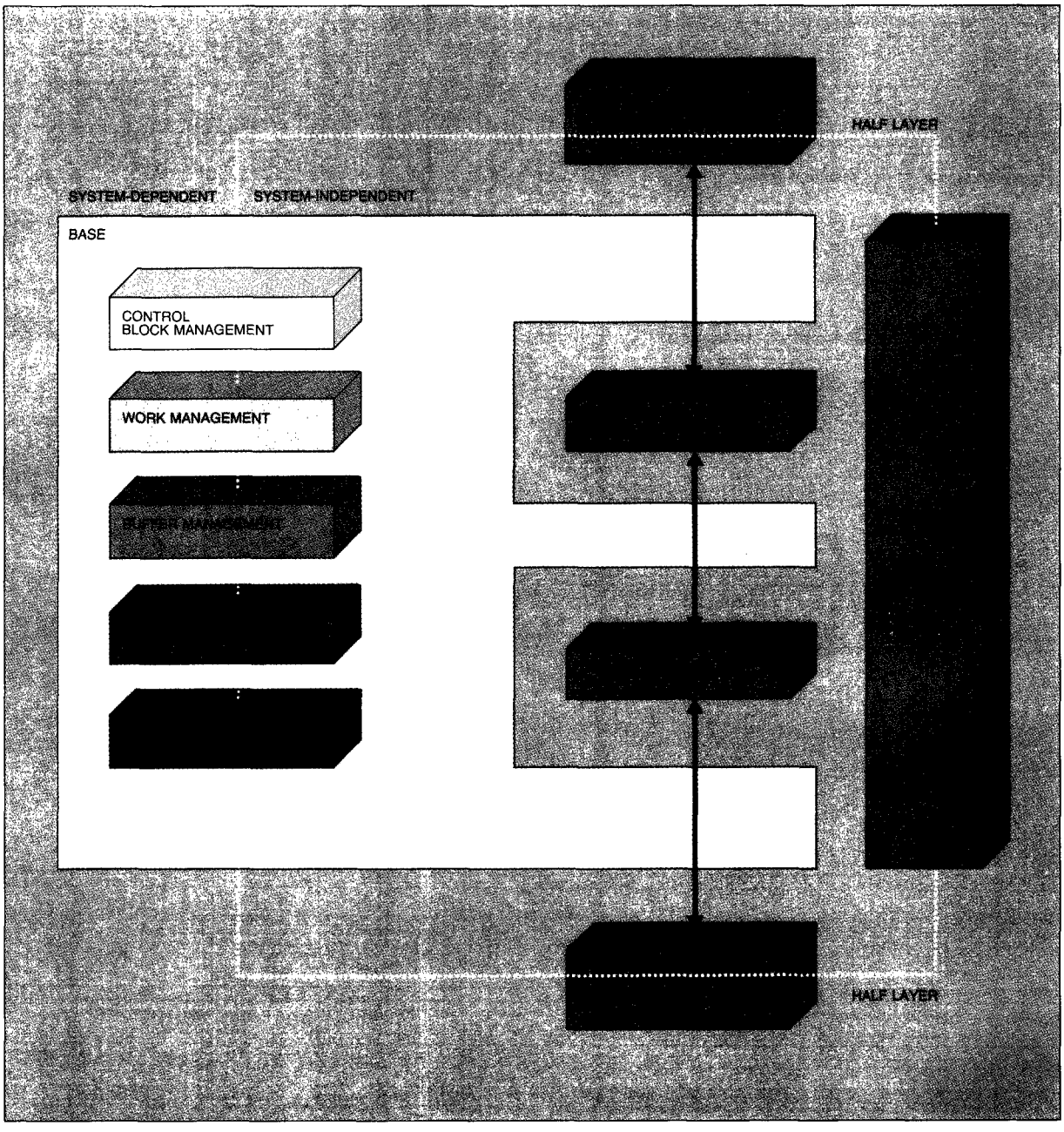
A SAP and a CCB exist at the interface of two layers, and one half of each block exists in each of the two layers. The half in the user layer represents the provider of the service to the user layer and thus is called a Provider SAP (PSAP) and a Provider CCB (PCCB). The half in the provider layer represents the user of the service to the provider layer and thus is called a User SAP (USAP) and a User CCB (UCCB). Each half block has its Base part and its Layer part. The Base parts are used to maintain the relationship between the two halves of a control block. The Layer parts are used to maintain layer-specific information. In particular, the Session Control Block (SCB) used by the Session Protocol Machine (SPM) is located in the User Connection Control Block in the session layer. Figure 9 shows an example of the control blocks.

One of the major differences in different implementations of layered communications standards is the means of achieving the interlayer communications within a system.

Interlayer communication. One of the major differences in different implementations of layered communications standards is the means of achieving the interlayer communications within a system. The experimental implementation architecture at Palo Alto that we have been discussing provides system-independent functions for interlayer communications. A layer requesting an information transfer prepares a work request, which is functionally equivalent to the Interface Data Unit (IDU) defined in the ISO/OSI Reference Model, and associates it with a SAP or a Connection Control Block that is shared with the layer accepting the information transfer. The accepting layer is then eventually called by the Base to perform the scheduled work request and return when done. If the accepting layer—in performing the scheduled work request—requires the services of another layer, that other layer produces work requests while it is invoked.

Execution environment. In a layered communication system, a work request originating at one end (top or bottom) of the layered system starts a sequence of related work requests that are executed in a synchronous manner. Such a sequence is called a thread of execution and is managed by a stack of work requests. The experimental implementation architecture allows multiple threads that may be concurrently served. This multithread environment is a

Figure 8 OSI implementation environment for the experimental system at IBM Palo Alto



highly efficient execution environment for a layered communication system because it takes advantage of the operating system multitasking feature (if available) and at the same time minimizes the task-switching overhead. Figure 10 shows an example of the task switch involved in processing an application program request and a network message.

Mapping and serialization. One function of the session layer is to map a session connection to a trans-

Figure 9 Example of system control blocks

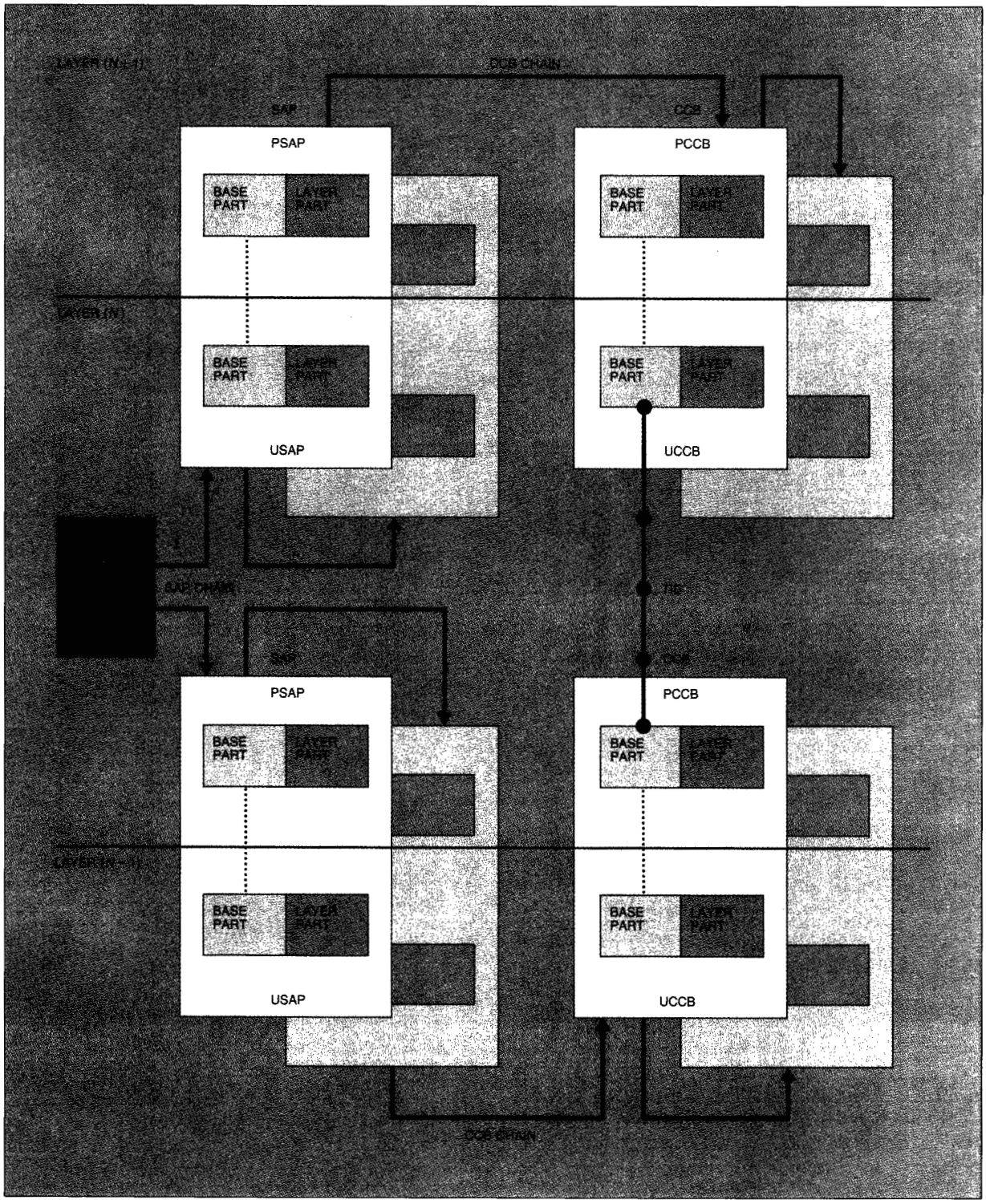
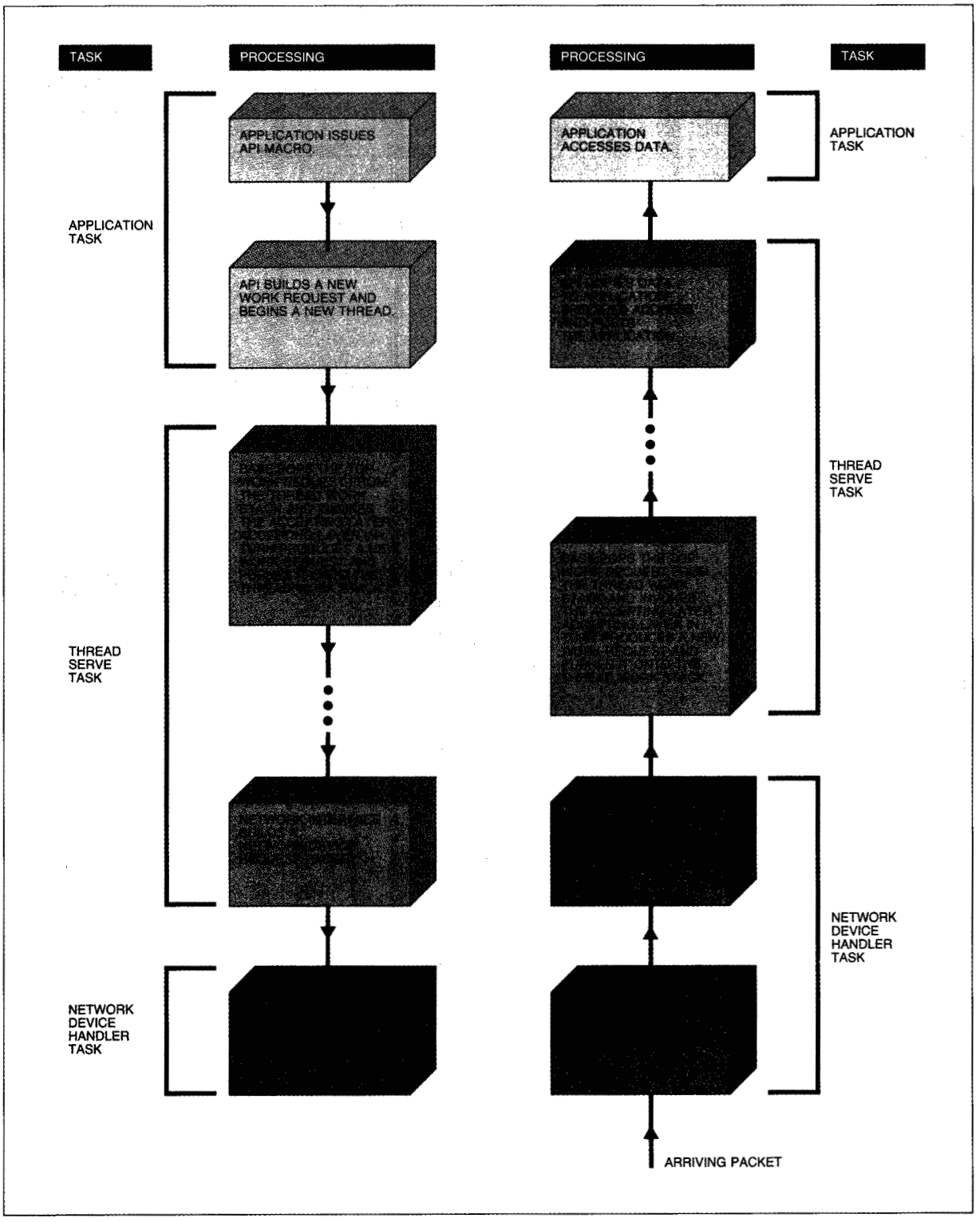


Figure 10 Execution environment



port connection. To achieve this, the session ties the UCCB representing the session connection with the PCCB representing the transport connection during the session connection establishment. This tie is not restricted to a one-to-one basis. Rather, the layers that support multiplexing may use many-to-one ties and the layers that support splitting may use one-to-many ties. Besides facilitating the mapping, the tie is used by the Base in a multitasking environment to serialize the access to the Connection Control Block. Serialization provides that, although there is an instance of a session layer invocation working on a UCCB or on the tied PCCB, no other invocation of the session layer is working on the same CCBs.

Implementation issues of the actual physical location or makeup of the buffer are divorced from the layer design.

Interlayer flow control. The ISO/OSI Reference Model leaves it as a local matter how one layer may exert back pressure to the adjacent layers so as to prevent flooding of requests into the layer. In the experimental OSI implementation, layers may use a special *autorecall* work request to prevent a flood of requests. For example, when API begins a new thread to send data to an application, it places an autorecall at the bottom of the thread work stack. API then may not accept any more send data requests from the application program until it is rescheduled with the autorecall work request, indicating that the previous application data have been sent to the network. The layer below API may suspend the thread so as to prevent API from being invoked with the autorecall work request.

Buffer management and timer management. The Base buffer management service allows a layer to perform logical rather than physical operations on a buffer. Thus the implementation issues of the actual physical location or makeup of the buffer are divorced from the layer design.

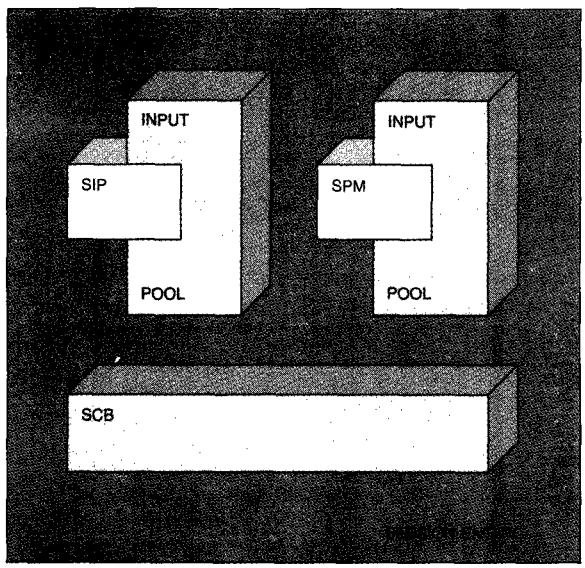
The Base timer management service allows a layer to start a timer. The Base manages the timer requests, and, when a timer expires, the Base schedules a timer-pop work request. The layer that requested the timer service is then invoked with the timer-pop work request and reacts according to the communication protocol.

Directory service. In the ISO/OSI Reference Model, the communicating session entities cooperate through Transport SAPs. Finding the remote Transport SAP address to which the responding session entity is attached and the local Transport SAP address to which the initiating session entity is attached is the function of the directory server. The organization and the means of accessing the directory are highly implementation-dependent. In our experimental implementation, the Systems Manager serves as the layer interface to the directory service. The directory service request and response work requests are exchanged via a SAP between a layer and the Systems Manager.

Session layer in the experimental OSI implementation environment. The PASS technique, like most formal definition techniques, requires an execution model to describe the protocol precisely. It is clear that this execution model cannot be an ideal architecture under all product contexts. Generally, the more complex the execution model, the more difficult it is to embed the generated code into a complete environment. We therefore retain only the essential features of the PASS execution model required to specify the implementation-independent aspects of the session protocol. We then build a complementary

Generally, the more complex the execution model, the more difficult it is to embed the generated code into a product environment.

Figure 11 Session layer in the experimental OSI *implementation environment*



component called Session Interface Process (SIP) to shield the SPM and to complete the implementation-dependent functions of the session protocol. The SPM together with SIP then forms the OSI session layer in the experimental OSI implementation. Hierarchically the SPM is a subroutine called by the SIP. Figure 11 shows a conceptual structure of the session entity.

Session interface process. The Session Interface Process (SIP) represents the session layer to the other system components in our experimental OSI implementation. It communicates with Session Service User (SS-user) and Transport Service Provider (TS-provider) using the work-request mechanism. In processing a work request, SIP handles the implementation-dependent functions previously discussed and requests the services from SPM to perform other protocol-related functions. An interface is established for SIP to access the services provided by the SPM. The following objectives are set in defining this interface:

- Define the partitioning of the OSI session functions between SIP and SPM.
- Define the syntactic and semantic nature of the interactions between SIP and SPM.
- Allow the interactions between SIP and SPM to be simple.
- Prevent the need for any other interface (such as a directory server) between SPM and system-dependent components.
- Allow SIP to be minimally affected by changes in the ISO session standard.
- Allow SPM to be transparent to the changes in the implementation environment.
- Minimize the effect on experimental designs in IBM Palo Alto and IBM Heidelberg.

Message exchange between SIP and SPM. The interaction messages between SIP and SPM are called the SPM in-event and SPM out-event. The format and semantics of these messages closely resemble the session service primitives and the transport service primitives defined in the ISO session service specification and the ISO transport service specification. These messages are exchanged using the input buffer mechanism provided by the PASS technique. When the processing of a work request requires SIP to invoke the SPM services, SIP creates an SPM in-event and deposits it in the SPM input buffer. SPM is then invoked to process the in-event in its input buffer. Upon returning from SPM, SIP finds SPM outputs in the SIP input buffer. SIP then maps the SPM out-event into the work request to communicate with the next layer.

In SPM, the input buffer is allocated from permanent storage. Where the OSI session protocol specification requires an SPM to queue an in-event, SPM simply leaves the in-event in the input buffer.

The IBM Palo Alto experimental implementation, however, allocates the input buffer from automatic storage. In this case, SIP is responsible for preserving the appearance of the permanent nature of the SPM input buffer.

Session control block. An SCB, which exists for each active session connection, contains all information related to the session connection. SIP is responsible for creating, initializing, and deleting the session control block. The local options subblock is initialized with implementation-dependent values. All other bytes are initialized to X'00.' Once initialized, SPM is responsible for maintaining all but the local options subblock of the session control block.

SPM needs to know the values of the implementation-dependent session entity parameters. The following local options are stored in the SCB by SIP for SPM:

- Version number
- Availability of transport-expedited service
- Transport-connection reuse option
- Maximum TSDU sizes to be proposed
- Send-segmenting option (i.e., local to remote)
- Receive-segmenting option (i.e., remote to local)
- User data-hiding option
- Functional units supported

Concluding remarks

The specification technique PASS introduces an intermediate step on the way from an informal protocol specification to the program code. A PASS description is precise and allows a semiautomatic derivation of program code. This paper has presented the basic concepts of PASS and has demonstrated the synthesis of code from a given specification. PASS was applied to the ISO session protocol. A generator has been implemented to produce code from the specification automatically. With this method the code development time is considerably reduced. The OSI experimental project has proved the feasibility of creating an implementation-independent (session) protocol machine that is generated directly from the PASS specification of the (OSI session) standard by automatic means. There are some path-length overheads introduced, such as mapping the interface records at the layer boundary to the SPM interface records. However, considering the size of the SPM and the functions it performs, we feel that these overheads are negligible. Besides, the clear separation of the functions may localize execution, thereby reducing the number of page faults. In the testing phase, the clear separation of the communication aspects (PASS graph) and the internal aspects (PASS graph refinement) simplified the testing effort enormously. Because of the automatic code generation for the PASS graph, there can be no difference between the specification and the code. Further research is needed to analyze the performance of the generated code.

Acknowledgments

We would like to thank our colleagues at the IBM European Networking Center and the IBM Palo Alto OSI research group for their helpful and constructive comments. We thank particularly J. B. Staton for his thorough review of the manuscript.

Cited references

1. ISO 7498. *Information Processing Systems—Open Systems Interconnection—Basic Reference Model*, International Organization for Standardization, Geneva, Switzerland (1983); may be obtained from Omnicom, Inc., 501 Church Street, Vienna, Virginia 22180.
2. A. Fleischmann, *Description of the Specification Technique PASS*, IBM ENC Technical Report (1987); may be obtained from the IBM European Networking Center, Tiergartenstrasse 15, 6900 Heidelberg, Germany.
3. ISO 9074. *Information Processing Systems—Estelle—A Formal Description Technique Based on an Extended State Transition Model*, International Organization for Standardization, Geneva, Switzerland; may be obtained from Omnicom, Inc., 501 Church Street, Vienna, Virginia 22180.
4. ISO 8807. *Information Processing Systems—Open Systems Interconnection—LOTOS—A Formal Description Technique Based on the Temporal Ordering of Observation Behavior*, International Organization for Standardization, Geneva, Switzerland; may be obtained from Omnicom, Inc., 501 Church Street, Vienna, Virginia 22180.
5. C. A. Vissers and G. Scollo, "Formal specification in OSI," *Proceedings, Networking and Open Systems, Lecture Notes in Computer Science No. 248*, Springer-Verlag, Heidelberg (1987).
6. E. Brinksma, "A tutorial on LOTOS," *Proceedings of the IFIP Conference on Protocol Specification, Testing, and Verification V*, North-Holland Publishing Co., Amsterdam (1986).
7. R. Milner, "A Calculus of Communicating Systems," *Lecture Notes in Computer Science No. 92*, Springer-Verlag, Berlin (1980).
8. H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification I*, Springer-Verlag, Berlin (1985).
9. ISO 8326. *Information Processing Systems—Open Systems Interconnection—Basic Connection Oriented Session Service Description*, International Organization for Standardization, Geneva, Switzerland (1983); may be obtained from Omnicom, Inc., 501 Church Street, Vienna, Virginia 22180.
10. ISO 8327. *Information Processing Systems—Open Systems Interconnection—Basic Connection Oriented Session Protocol Specification*, International Organization for Standardization, Geneva, Switzerland (1983); may be obtained from Omnicom, Inc., 501 Church Street, Vienna, Virginia 22180.

Albert Fleischmann *IBM European Networking Center, Tiergartenstrasse 15, 6900 Heidelberg, Germany.* Dr. Fleischmann joined IBM in 1985 as a guest scientist at the IBM Science Center in Heidelberg, where he worked on software technology for specifying and implementing communication protocols. He joined IBM Germany in 1986 as a research staff member of the European Networking Center. Prior to joining IBM, he worked two years for the DFN project (German Research Network), a government-sponsored project for establishing an open computer network for the German research community. Albert Fleischmann received a Diploma in computer science in 1981 and a doctoral degree in computer science in 1984, both from the Friedrich Alexander University of Erlangen, Germany.

Seung-tae Chin *IBM Communication Products Division, P.O. Box 10500, Palo Alto, California 94304.* Mr. Chin joined IBM in 1981 in Research Triangle Park, North Carolina, where he worked on meta-implementation and conformance testing of LU 6.2. He is now a staff programmer at the CPD Advanced Programming Center at Palo Alto, where he has worked on LU 0 and recently on the development of OSI-related programs. Mr. Chin received an M.S. in computer science in 1981 from the University of Tennessee.

Wolfgang Effelsberg *IBM European Networking Center, Tiergartenstrasse 15, 6900 Heidelberg, Germany.* Dr. Effelsberg joined IBM in 1983 as a Postdoctoral Fellow at the IBM Research Center in San Jose, California, where he did research on highly available database systems. He joined IBM Germany in 1984 as a research staff member of the Heidelberg Scientific Center. Since 1985, he has managed the Distributed Applications Research project at the IBM European Networking Center in Heidelberg. Prior to joining IBM, he spent two years as an assistant professor at the University of Arizona in Tucson, where he worked on the optimization of buffer management in database systems. Dr. Effelsberg received a Diploma in computer science in 1976 and a doctoral degree in computer science in 1981, both from the Technical University of Darmstadt, Germany.

Reprint Order No. G321-5297.