From Formal Specification to Implementation: Experience with Protocol Tools

T.J. Baumgartner[1], D.M. Kristol[2], J.D. Milleville[3], P.S. Parikh[4]

AT&T Bell Laboratories
[1]6200 East Broad St., Columbus, OH 43213
[2]600 Mountain Ave., Murray Hill, NJ 07974
[3]263 Shuman Blvd., Naperville, IL 60566
[4]1200 East Warrenville Rd., Naperville, IL 60566

Abstract

We describe here our efforts to develop the Q.931 protocol software for the 5ESS™ Central Office Switch. Q.931 is a layer 3 message-based, user/network interface protocol for the Integrated Services Digital Network (ISDN). Our approach uses a single formal specification written in the Augmented Protocol Specification Language to model the Q.931 protocol as a set of communicating extended finite state machines. From this specification, tools validate the protocol and generate the bulk of a C language implementation. We successfully integrated this code, along with other handwritten code, into a working system in the lab. Our results demonstrate that we can use protocol tools effectively to develop high quality protocol software quickly from formal specifications.

1. Introduction

In telecommunication systems, protocol specifications are typically described using high-level specification languages (e.g. Specification and Description Language (SDL), LOTOS, or Estelle[1]) along with natural language text to resolve discrepancies. Despite this approach, misinterpretation, inaccuracies, and incompleteness occur when these high-level specifications are transformed into implementations. Furthermore, the specifications must be changed to correct inaccuracies or to add capabilities. Thus, revisiting protocol implementations to add new capabilities becomes an important design concern.

The work reported in this paper describes an experiment to implement the Q.931 protocol software for the 5ESS™ Central Office Switch[2]. The goal of the experiment was to specify Q.931 in a formal specification language, use protocol development tools to derive substantial parts of the implementation from the specification, and embed the implementation in a UNIX® STREAMS module.[3]

1.1 Background

The Q.931 protocol is an Open System Interconnection (OSI) layer 3 message-based user/network interface protocol for the Integrated Services Digital Network (ISDN)[4]. We developed Bellcore's Q.931 protocol as specified in TR268[5], which was the standard for National ISDN-1.

In our experiment we considered how to package the protocol. Because of the availability of the UNIX STREAMS development environment within the 5ESS Switch development environment, and because of the numerous advantages of UNIX STREAMS in packaging protocols[3], we decided to develop the Q.931 protocol within a UNIX STREAMS module.

With these capabilities defining the base architectural components, we also decided to experiment with high-level specification languages, and with a new process based on protocol engineering tools, to develop the Q.931 protocol.

1.2 Related Work

Formal methods have received considerable attention in software development in recent years. Protocol development, in particular, has applied formal methods to the front-end and back-end of the development process: the specification, verification, and testing of protocols[6]. Recent work frequently focuses on one of these areas at a

® UNIX is a registered trademark of UNIX System Laboratories, Inc.

0-8186-3670-X/93 $03.00

time. The NewCoRe project[7], for example, focuses on the front-end of the development process, to make protocol requirements correct. Using a tool for automated validation, NewCoRe demonstrated the usefulness of discovering specification errors early in the development process.

At the back-end of the development process, researchers report a variety of theoretical results from using different conformance test techniques based on formal methods and practical applications based on these experiences[8]. When a formal specification exists for the system, the testing process can be automated. For example, the *POSTMAN* software package[9] tests an implementation, provided it can be described by a *single* finite state machine (FSM) specification. *POSTMAN* produces a list of inputs to apply to, and outputs to expect from, a correct implementation. This list exercises all parts of the implementation, and it can be used to drive automated testing. A recent paper[10] describes an adaptive testing procedure that can be used when the system specification comprises *multiple* FSMs.

In another recent trend, in software development, application generators produce executable code from specifications written in high-level specification languages. In relational database applications, for example, application generators have been widely used to translate forms, menus, reports, and more, into intermediate languages that a database system understands[11] [12].

In protocol development applications, Abstract Syntax Notation 1 (ASN.1) compilers have been developed that transform high-level specifications of application messages into actual message data structures for implementations[13]. Also in protocol development, templates of implementations have been generated from high-level specifications and transplanted into UNIX™ STREAMS modules with other handwritten code[14]. To support these efforts, powerful tools are already available to aid the development of compilers that transform specifications into implementations[15].

1.3 Our Work

We experimented with a development process for protocols in which automated tools support requirements specification and code generation. A single specification of the protocol drives the tools. The results obtained from this experiment show that this type of development process has definite advantages.

We used the Augmented Protocol Specification Language (APSL)[16], a high-level protocol specification language that models protocols as a collection of communicating extended finite state machines. Along with the more typical *validation tool*, we have available a tool that generates C language code from the APSL specification. Used together, these tools comprise a powerful development process. A developer can write the description of a protocol in APSL and validate the description to remove errors introduced in its writing. Once the protocol description is complete and validated, the code generator can be used to generate the code for the target environment.

The development process based on these tools provides a number of advantages. Finite state machines (FSMs) are a familiar, accepted abstraction. By starting with an FSM model, validating the specification, and generating code from the specification, we reduce the probability of errors when we embed the code in its execution environment. Extending the protocol because of enhancements or protocol corrections is simpler, because one high-level specification describes the protocol completely.

1.4 Application Architecture

Our target architecture platform for the Q.931 protocol was the 5ESS Switch, as depicted in Figure 1.

The Q.931 messages coming from originating and terminating Customer Premises Equipments (CPEs) are buffered in a First In First Out (FIFO) queue called **INQUE**, which preserves the order of messages. The read-side service routine of the Q.931 STREAMS driver, **USq931**, processes the incoming messages and converts them to an internal format, and the **USqmsg** module distributes them to the appropriate call processing process, **CP**, one per half-call, via a protocol-independent interface.

CP in Figure 1 implements call processing applications. When **CP** wants to send a message to a **CPE**, it invokes the write-side service routine of **USqmsg.** Here the message is processed and converted to Q.931 format. Then, the **USq931** driver sends it to the **CPE** using the I/O drivers.

The **USprsp** system process builds the STREAMS stack, provides stack integrity capabilities, and does "sanity checks" and reliability checks.

The Q.931 driver (**USq931**) implements the Q.931 protocol based on Bellcore's TR268. It is responsible for the following functions:

1. Layer 3 Finite State Machine
2. Message Translation
3. Error Control
4. Timer Support
5. Support for Multiple Calls
6. Protocol-Independent Interface

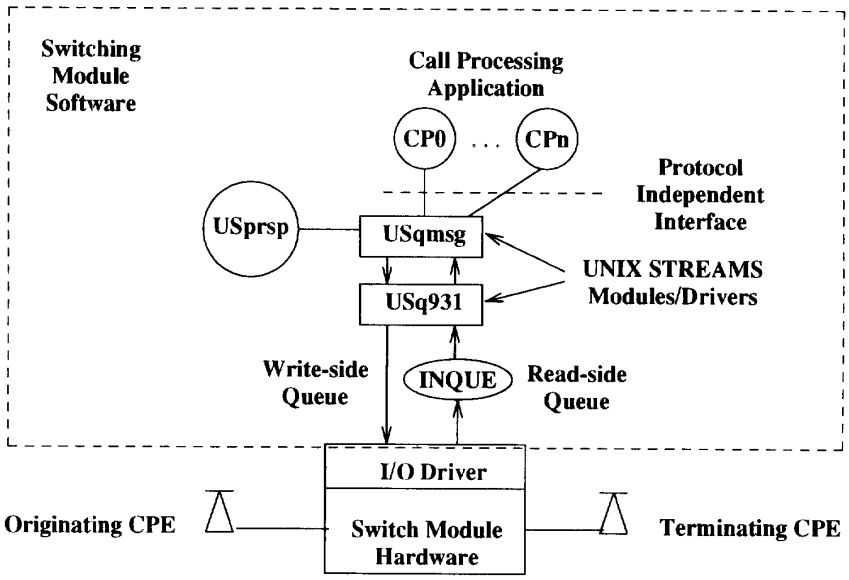


Figure 1: Q.931 Protocol Architecture

The *Layer 3 Finite State Machine* captures the essence of the Q.931 protocol and is the part of the implementation that we generate from the APSL specification. The Q.931 STREAMS module comprises two state machines, one for the originating side of the call and another for the terminating side.

In *Message translation*, the raw Q.931 message is translated from the **CPE** form into C data structures. Message translation supports both the encoding of messages being passed down, and the decoding of messages being passed up, the stream.

Error control manages two forms: (1) errors resulting from mis-sequenced messages; and (2) errors resulting from corrupt messages. *Mis-sequenced messages* arrive at inappropriate times. The state machine within the module recognizes them and performs appropriate error processing procedures in conjunction with **CP**. In a similar fashion, the module processes *Corrupt messages* that arrive from **INQUE**.

Timers are needed in the protocol to handle messages that must receive responses within some specified time. When timers expire on certain messages, either these messages will be retransmitted or error handling will be invoked. If the message arrives before the timer expires, the timer is disabled.

To support *Multiple calls* in the STREAMS module, the private data area of the module stores current state and associated call information of each half-call. A single call has private data associated with both the originating and terminating sides of the call.

The interface between **CP** and STREAMS modules is *Protocol-Independent*. Therefore the **CP** application should not change as the Q.931 protocol evolves.

2. The Development Process

In this project we used protocol validation and code generation tools to implement the protocol embedded in the Q.931 STREAMS driver of Figure 1. Figure 2 shows the development process we used to implement this protocol. This process is separated into three phases: (1) protocol specification, (2) protocol coding, and (3) integration and testing.

In the first phase the protocol is transformed into an APSL description based on the requirements given in Bellcore's TR268. The TR268 description was written in the System Description Language (SDL), along with a textual description. Figure 2 shows that we had to take into account requirements of the 5ESS Switch base architecture.

We transliterated the SDL version of TR268 to APSL by hand. The protocol validation tool (described in the next section) assisted by detecting a variety of errors, both syntactic and semantic. We modified the APSL specification as needed until the validator detected no errors. Then we began phase two, code generation.

In the second phase, we assume we have validated the APSL specification, and we can use it as input to the APSL-to-C compiler. The compiler also accepts a second input (described later), a description of how to

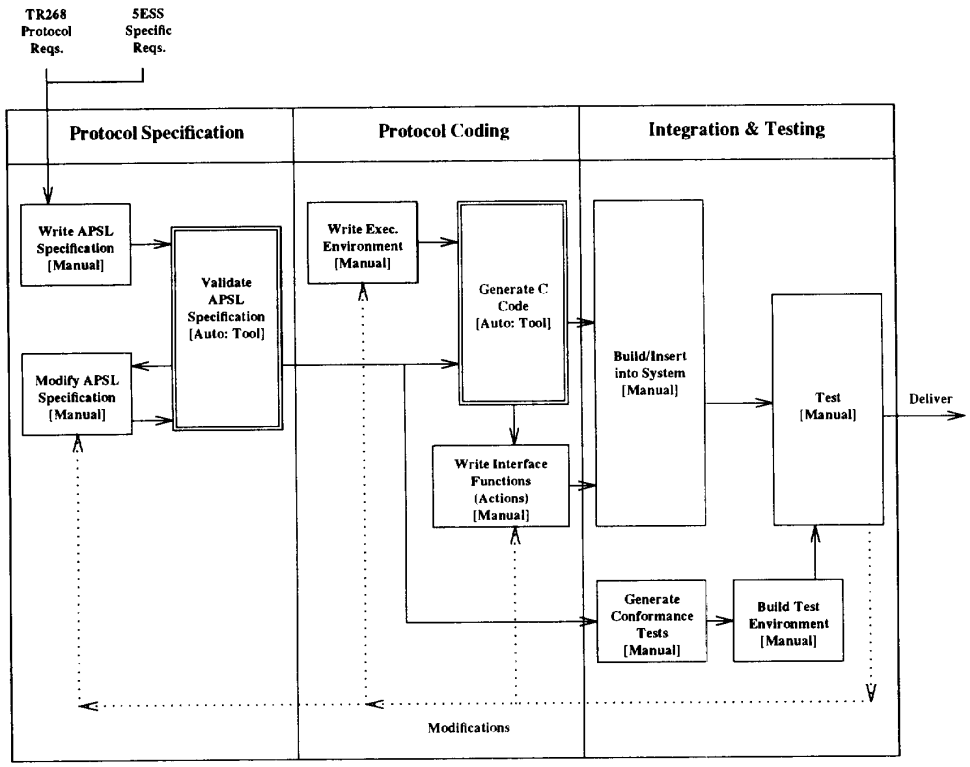


Figure 2: Protocol Validation and Code Generation Process

generate C code from the specification. Given these two inputs, we generate the C code.

In parallel, we code by hand a set of interface functions. These functions serve as the interface between the FSMs of the specification and the implementation environment. In our application, these interface functions typically are small. The APSL-to-C compiler produces templates for them. In the third phase of the process, we compile and integrate the generated and handwritten code with the system for testing.

We manually derived tests of the implementation from different threads of control supported by the protocol. If testing is successful, the software is delivered to the field. Otherwise, we must repair the APSL specification, the execution environment description, or the interface functions, as depicted in Figure 2. We found it was easy to determine which piece of code needed repair when an error occurred. We attribute this ease to our use of a formal specification and the supporting tools.

3. Specification and Validation

3.1 APSL and the State Machine Model

APSL[16] is an extended finite state machine (EFSM) specification language. A protocol is specified as a collection of communicating FSMs. Each FSM is described in terms of its states and the transitions between them. Transitions may include communication between machines, and tests on, or assignment to, *context variables*.

Communication between finite state machines is *synchronous* in the style (and syntax) of the Communicating Sequential Processes (CSP) language.[17] That is, communicating FSMs participate in *rendezvous*: an FSM that sends (receives) a message must wait until the partner FSM is ready to receive (send) the same message. At that point both may proceed. Messages may incorporate values that are passed from the sender to the receiver.

Figure 3 gives the state machine model used for our Q.931 specification. In this model we specify three state machines that work together: CPE, USQ, and CPK. CPE represents the Customer Premises Equipment (CPE) state machine on the originating and terminating sides of the

call. USQ represents the UNIX STREAMS Q.931 protocol state machine for the originating and terminating sides of the call. And CPK represents the call processing interface state machine for the originating and terminating sides, which resides in the CPs shown in Figure 1. The user/network interface that Q.931 defines exists between the CPE and USQ state machines.

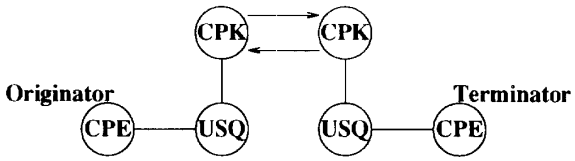


Figure 3: Validation State Machine Model

The validator program requires a complete specification of the interacting state machines. The originating and terminating sides of USQ, CPE, and CPK have identical specifications (although, of course, the parts that handle the originating and terminating behavior are different). In the implementation, not all of these state machines are needed: CPE models an ISDN phone, and CPK models part of the existing 5ESS call processing functionality. Only USQ actually resides in the Q.931 STREAMS driver of Figure 1.

The following example illustrates a single APSL transition from our application:

```
Null_0: Call_Init_1 WHEN
                CPE?setup(USQsutype)
              * CPK!setupind(USQsutype),
```

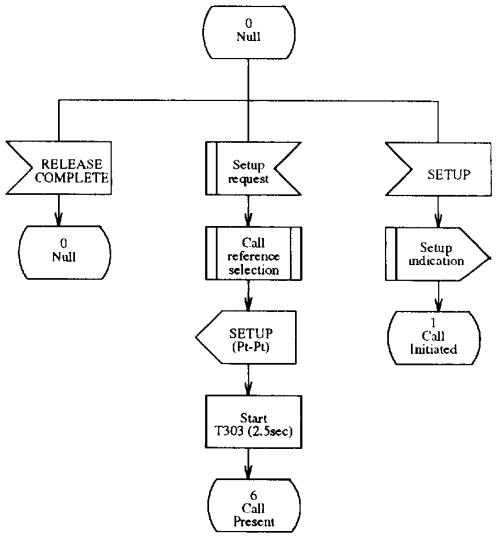
The example is from FSM `USQ`, which processes call setup. The transition may be read as follows: Make a transition from state `Null_0` (the idle state) to state `Call_Init_1` when we receive the message `setup` from process `CPE` ("customer premises equipment" — someone's telephone) and send a `setupind` message to the call processing terminal process, `CPK`. Setup information will be stored into context variable `USQsutype` when the message is received, and the same information will be passed along to the call processing terminal process.

Arbitrary code may be embedded in an APSL protocol specification within `{{ }}` brackets. The code (C language code, in our implementation) is ignored by most APSL tools. The APSL-to-C compiler, however, inserts this code into the implementation at the designated place. Although APSL context variables and embedded C variables are distinct, the embedded C code can refer to and set the values of context variables.

Example 1 shows a fragment of our APSL code (with line numbers added in [ ] brackets) that corresponds to the SDL in Example 2. The declarations of the state names and messages have been omitted; these provide an essential cross-check with the information in the `TRANSITIONS` section. Lines 9-11 are identical to the earlier example. Other transitions from state `Null_0` on lines 5 and 6 correspond to the other messages in Example 2. By specifying all of these message transitions for all state machines in the model, we completely specify the Q.931 protocol. With this specification, we can begin validation and code generation.

```
[1] TRANSITIONS /* for FSM USQ */
[2]
[3] /***** State 0: Null *****/
[4] /* Page B-9 of TR268 */
[5] Null_0:Null_0 WHEN CPE?relcom,
[6] Null_0:Call_Pres_6 WHEN
[7]           CPK?setupreq * CPE!setup
[8]         * TIMER!start(303),
[9] Null_0:Call_Init_1 WHEN
[10]          CPE?setup(USQsutype)
[11]        * CPK!setupind(USQsutype),
```

Example 1: APSL Specification of the Q.931 Protocol



Example 2: SDL Fragment for Q.931 Protocol

3.2 Validation Algorithms

We first applied an APSL protocol validator to our protocol specification. The validator identifies two classes of problems: static and dynamic. Static errors are those that can be found simply by examining the

specification, such as:

- Syntactic errors.
- States that have no successors or no predecessors.
- Messages sent but not received, or received but not sent.
- Mismatches in the number of parameters sent and received in messages.
- Context variables that are set but not used, or used but not set.
- FSMs that are never used.

Dynamic errors arise when the protocol is exercised. We use a technique like West's random exploration technique[18] to identify *deadlocks* and *livelocks*.

In random exploration, the validator starts with the protocol in the initial *global state* (comprising the states of the component FSMs and the values of the context variables). Then it repeats the following steps to simulate the protocol's behavior:

1. Identify all transitions in all the component FSMs that are possible in the current global state.
2. Choose one such transition at random.
3. Simulate the transition, which leads to a new global state.

The goal is to simulate as much as possible of the protocol's behavior and reach any invalid or deadlock states that may exist. A *deadlock* arises when there are no transitions possible from a global state. Thus the validator can detect deadlocks directly. In the implementation, a deadlock would cause the protocol to "lock up" and fail.

A *livelock* occurs when the simulation explores the same set of states repetitively and the protocol makes no progress; the implementation would fail to do what it was meant to. Livelock is inferred from a metric that the validator provides. Specifically, the validator provides a measure of how thoroughly the protocol has been explored using the exploration algorithm above. The more thoroughly the protocol has been explored, the more likely it is to be correct. When a livelock occurs, the simulation leaves much of the protocol unexplored, according to the metric.

The validator also provides a single stepping facility. Using this mode, a developer can step through a call one state transition at a time to debug the APSL code. This mode could also help a new developer learn the behavior of the model.

4. Code Generation and Implementation

4.1 Generating C Code from the APSL Specification

The second software tool we used produces C code for the core of our implementation. Figure 4 shows how information flows through this compiler. Its unusual structure separates the "policy" of code generation ("Execution Environment Program in APSLANG") from the "mechanism" ("Interpreter"). APSLANG is a meta-language that describes what to do, given a protocol specification. By changing the APSLANG description, developers can alter the compiler's generated code to suit different execution environments.

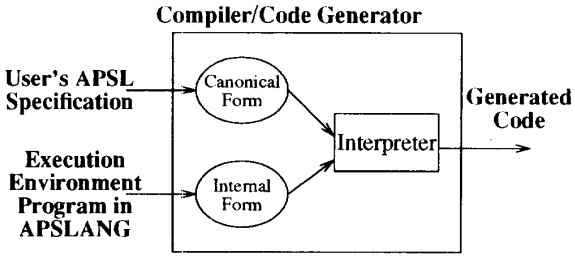


Figure 4: Information Flow

We run the compiler after we have validated the Q.931 specification. The compiler generates C code that completely describes the state machines defined by the APSL specification and interface functions that provide a clear interface to the inputs and outputs of the state machines. By our definition, "stimulus functions" provide the inputs to the state machines, and "response functions" provide the outputs or actions of the state machines, as shown in Figure 5.

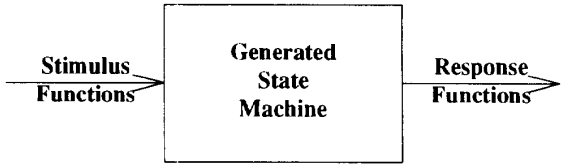


Figure 5: State Machine Interface Functions

The compiler generates all the C code for stimulus functions. Ordinarily it generates no C code for response functions, which, although they have a predictable interface, must be hand-coded. However, we wrote additional APSLANG code so the compiler would emit the skeleton of each response function. For both stimulus and response functions, the compiler easily derives the names of the functions from the names of the communicating FSMs and the name of the message. The compiler would

generate functions `STIM_CPE_setup_USQ()` and `RESP_USQ_setupind_CPK()`, for example, based on messages on lines 9-10 of Example 1.

4.2 Implementation

The Q.931 STREAMS module consists of three types of code:

1. Code generated from the APSL specification, comprising the USQ state machine and its stimulus and response functions. This code is integrated into the STREAMS module on the 5ESS Switch (see Figure 1). The Q.931 STREAMS module calls all generated code through STREAMS read and write service procedures.
2. Existing 5ESS code called from the STREAMS module. This code is reused from a previous implementation and comprises routines that translate the incoming (outgoing) binary form of the Q.931 messages to (from) the C data structures that the call processing kernel understands. It also finds the correct application process destination to which to send incoming Q.931 messages.
3. New code written specifically for the Q.931 STREAMS application. This code maintains ISDN call information and generated code context information in a hash table on a per-call basis. This allows the Q.931 STREAMS module to support multiple calls. When the CPE or switch sends a new call setup message, the code initializes a state machine context for that call and stores it in the hash table. Also, when the code receives a message that is the last message for a given connection (as specified by the Q.931 protocol), it removes that connection's context information from the hash table.

All Q.931 messages from the CPE to the switch, and from the switch to the CPE, pass through the STREAMS module. Each time, the ISDN call context information is retrieved from the hash table, based on ISDN call identification information received in the message. A read service routine accepts incoming Q.931 messages and does message translation (decoding). It calls the appropriate generated stimulus function, based on the Q.931 message type, and passes along the call's context. Calling this stimulus function initiates processing in the generated state machine. State machine processing proceeds based on the current state of this connection and the event (Q.931 message), and the appropriate response function is called.

The response functions determine to which application process to send the message, address the message for that process, and then send the message upstream to its destination. The response functions for messages coming upstream may also send messages back downstream, depending on the control flow (e.g., error processing). The response functions for outgoing messages perform message translation (encoding) of the Q.931 message and send the message downstream to the appropriate destination, based on addressing information contained in the message and on the call's context. Response functions that handle the final message in an ISDN call scenario (i.e., which terminate a connection) also remove the ISDN call information from the hash table.

5. Testing

Because we used a formal specification language, we could actually begin testing at the specification step. As we said earlier, we used a validation program to look for a variety of errors. However, when we integrated the code into the 5ESS system lab, we used standard testing procedures. Specifically, we used three techniques in the following order to test the Q.931 STREAMS module: (1) a STREAMS simulator, (2) a 5ESS system simulator, and (3) the 5ESS system lab.

The earliest technique we used, in the coding phase, was a *STREAMS simulator*. The simulator comprises a UNIX process that includes the STREAMS environment, the STREAMS module(s) to be tested, and a simple user command interface. The process executes on any general-purpose machine running the UNIX operating system. During simulation the user manually builds the desired stream and creates and sends messages on the stream using the STREAMS simulator user interface. The STREAMS module can then be debugged with standard UNIX debugging tools. This tool makes it quick and easy to bring up a module under development and to test it during coding. Its limitation is that it can be hard to create the data and test sequences needed to exercise the module completely.

The second technique we used was a *5ESS simulator* test environment. The simulator runs on a mainframe computer and allows a developer to debug 5ESS code from a computer terminal. The absence of 5ESS switch hardware makes this environment a simulator. It provided the first step toward integrating the coded Q.931 module with 5ESS switch software. This environment comprises a standard source level debugger (at the C level) for developers. Example capabilities include setting breakpoints, function tracing, and inserting data. With this simulator, we debugged critical call flows (test sequences) along with the 5ESS call processing application.

In our final technique we actually moved the Q.931 STREAMS module into the *5ESS system lab*. It is here

that we first tested the interaction of the Q.931 STREAMS module with real 5ESS switch hardware (and software).

As described earlier, the Q.931 STREAMS module consists of three types of code: (1) generated code, (2) existing 5ESS switch code, and (3) new handwritten support code. We found different kinds of errors in each type. For example, when considering the *generated code*, most of the errors were found not in the generated C code (which includes the state machine and the interface functions), but in a mismatch between the APSL specification and the 5ESS environment. In other words, the APSL specification we wrote did not match 5ESS requirements. We had to modify the APSL code so we could integrate our Q.931 module successfully into the 5ESS environment. Once we identified those errors, however, they were easy to fix, since all that was required was a change in the APSL specification and a re-compilation into the C state machine.

We found few errors in the *existing 5ESS switch code* because we reused that code from an existing implementation. These errors were mostly due to software integration inconsistencies.

And finally, the few errors we found in the *new handwritten support code* were easily identified, because the clear interface to the state machines made the message flow easy to trace.

Overall, we estimate we found about 30-40 errors of varying seriousness. Of those, about ten related to the protocol specification itself, and we found them early, using our tools. As a result, we experienced fewer errors during the other testing procedures, where they are harder to find and more expensive to fix.

6. Conclusions

In this paper we described how we used protocol validation and code generation tools to implement the Q.931 protocol on the 5ESS Switch. Here we summarize our experience. Our basic method is to write a formal specification for a protocol, validate it, and generate code from the same specification. The implementation model follows a simple abstraction (the finite state machine model). We found this method has many advantages for software quality, easier maintenance, and reduced development cost.

1. We can detect errors in the protocol and implementation and resolve them early in the process.
2. The interface to the generated code via interface functions is clearly defined, which simplifies integration and testing.
3. There are fewer lines of code to develop and maintain. In our experiment, we wrote 100 lines of APSL and supporting C code to implement Q.931. We estimate that the existing implementation has about 2500 lines of C code to do the same thing. Extrapolating to a complete implementation of Q.931, we estimate we would need about 900 lines of APSL and C code to correspond to about 25,000 lines in the existing implementation.
4. Since we generate code directly from the APSL specification, we only need to maintain it, not the generated code.
5. The clear protocol specification minimizes "discovery time" for engineers that join the project and must understand how the protocol works. Furthermore, a formal specification makes possible more powerful browsing tools that can reduce discovery time even further. For example, the previously mentioned single stepping facility in the validator allows the user to observe the protocol's behavior step by step.
6. Once we make the initial investment to set up the protocol generation environment, the process remains the same throughout the implementation's lifecycle.

This formalized method of implementation simplifies the task of developing a protocol initially and revisiting it later to add enhancements. We only need to change the APSL specification, rather than tweak the software directly. This process promotes spiral (or iterative) development. We can develop and augment versions of the protocol specification in an evolutionary manner until we reach a final, complete, protocol specification. We can validate each version, generate the code for it, and try it out in the lab.

7. Acknowledgements

Many people helped bring together the results reported in this paper. Most of these persons are recognized through the documents referenced in the text. We would particularly like to acknowledge the following people: C.S. Curtin, J. Purcell, J.S. Krevitt, W-H.F. Leung, A.M. Mishra, A-M. Mohamed, P.R. Roberts, Z. Ruan, and S.E. Thornton.

8. References

1. S.C. Murphy, P. Gunningberg, and J.P.J. Kelly, "Experiences with Estelle, LOTOS and SDL: a protocol implementation experiment," *Computer Networks and ISDN Systems*, 22 (1991), pp. 51-59.
2. "The 5ESS™ Switch," *AT&T Technical Journal*, Jul./Aug., 1985, Vol. 64, No. 6, Part 2.
3. *UNIX™ System V Release 4 Programmer's Guide: STREAMS*, UNIX System Laboratories, Inc., Prentice-Hall, 1990.
4. CCITT Red Book, Vol. III, Fascicle III.5, Integrated Digital Services Network (ISDN) Recommendation Q.931 (Study Group XVIII), 1984.
5. "ISDN Access Call Control Switching Signaling Requirements," Bellcore TR-TSY-000268, Issue 3, May, 1989.
6. *AT&T Technical Journal*, "Special Issue on Protocol Testing and Verification," Jan./Feb., 1990, Vol. 69, No. 1.
7. J.A. Chaves, "Formal Methods at AT&T - An Industrial Usage Report," Proceedings of the FORTE 91 International Conference on Formal Description Techniques, Sydney, Australia, pp. 83-90, Nov. 19-22, 1991.
8. B.S. Bosik and M.Ü. Uyar, "Finite state machine based formal methods in protocol conformance testing: from theory to implementation.", *Computer Networks and ISDN Systems*, 22 (1991) pp.7-33, North-Holland.
9. A.T. Dahbura, K.K. Sabnani, and M.Ü. Uyar, "Algorithmic Generation of Protocol Conformance Tests," *AT&T Technical Journal*, Vol. 69, No. 1, Jan./Feb. 1990.
10. D. Lee, K.K. Sabnani, D.M. Kristol, M.Ü. Uyar, and S. Paul, "Conformance Testing of Protocols Specified as Communicating FSMs," *Proceedings of IEEE INFOCOM '93*, pp.115-127, Mar., 1993.
11. K.S. Brathwaite, *Relational Databases: Concepts, Design and Administration*, McGraw-Hill, Inc., 1991.
12. A. Zornes, "Relational Databases: Making Peace with the Past," *Computerworld*, pp.65-69, Feb., 1991.
13. G. Neufeld and S. Vuong, "An Overview of ASN.1", *Computer Networks and ISDN Systems*, 23 (1992), pp.393-415, North-Holland.
14. W. Hong, "A Protocol Software Application Generator," IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, June 1-2, 1989.
15. J.C. Cleaveland, "Building Applications Generators," *IEEE Software*, pp.25-33, July, 1988.
16. M.H. Sherif and A.S. Krishnakumar, "Evaluation of Protocols from Formal Specifications: A Case Study with LAPD," *Proceedings of IEEE Telecommunications Conference & Exhibition* (Globecom), pp. 879-886, Dec., 1990.
17. C.A.R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 8, pp. 666-677, Aug., 1978.
18. C.H. West, "Protocol Validation in Complex Systems," *Proceedings ACM SIGCOMM '89 Symposium on Communications Architectures and Protocols*, pp. 303-312, Sep., 1989.