Reprint

From Protocol Models to Their Implementation: A Versatile Testing Methodology

M. Varsamou, N. Papandreou, and Th. Antonakopoulos

IEEE Design and Test of Computers

VOL. 21, NO. 5, SEPTEMBER-OCTOBER 2004, pp. 416-428

From Protocol Models to Their Implementation: A Versatile Testing Methodology

Maria Varsamou
University of Patras

Theodore Antonakopoulos
University of Patras

Nikolaos Papandreou
Computer Technology Institute

Editor's note:
In this protocol design and verification scheme, high-level models serve in generating simulation sequences for low-level models, and all simulation is based on directed testing. The methodology is versatile and flexible, but it might be difficult to set up the first time.

—Carl Pixley, Synopsys

Design and test of communication protocols relies extensively on formal description languages. Along with the textual description, the International Telecommunication Union (ITU), the IEEE, and other organizations provide formal language representations in protocol specifications.[1] The telecommunications industry uses the Specification and Description Language (SDL)[2] as an efficient way to develop a precise protocol model and validate its functionality under various conditions. These conditions take the form of message sequence charts (MSCs)[3] and enable validation of formal characteristics, such as deadlock avoidance, to permit error detection early in the design process.

Although a high-level model constitutes a complete and accurate representation of the protocol specifications, its porting in a real embedded system entails additional constraints. The target hardware imposes several limitations, not only on the final implementation but also in the protocol's testing process, because it's necessary to implement an appropriate test bench. Often, creating a custom protocol implementation along with a custom test bench will overcome these difficulties. Although this approach seems to handle the system constraints efficiently, it also minimizes the design's flexibility. Protocol porting to a new system architecture requires the additional effort of rewriting part of the protocol's code and developing new testing modules.

However, various commercial tools support the compilation of SDL models into lower-level programming languages, such as C/C++. Using functions and libraries that reflect the specific hardware architecture permits further translation of the compiled protocol model into executable code for a target microprocessor. In particular, these functions enable integration of the SDL model with an operating system and a microprocessor-based hardware platform. Thus, using system-specific library functions lets us obtain different protocol implementations for different target devices.

Final code optimization depends on the compiler's efficiency and the flexibility of the high-level interface between the protocol's model and its environment. In any case, the final code requires reevaluation. Although this code was created from an already verified SDL model, the various low-level system attributes—such as interrupt handling, task scheduling, and memory or peripheral access—might result in different or unexpected behavior. It can be very hard or impractical to integrate these attributes into the MSC simulations.

0740-7475/04/$20.00 © 2004 IEEE Copublished by the IEEE CS and the IEEE CASS

Therefore, we need an appropriate testing module that exploits the analytical protocol testing scenarios developed in the high-level system model.

In this article, we present such a flexible and versatile design methodology, along with a testing environment for the development of signaling protocols for point-to-point communication links. This approach combines multilevel protocol modeling and validation with a top-down design and test process that enables systematic translation of the high-level abstract model into low-level code. The development and testing environment relies on a reconfigurable setup that does not depend on a specific target processor. A key feature of our approach is that the analytical testing scenarios developed for the high-level model serve to verify the low-level implementation. We accomplish this with a custom testing-sequence translator that uses information from the MSC simulations to configure the low-level testing module. This article also shows how the proposed approach can apply to the development and testing of the signaling protocol used in asymmetric digital subscriber line (ADSL) communication links.

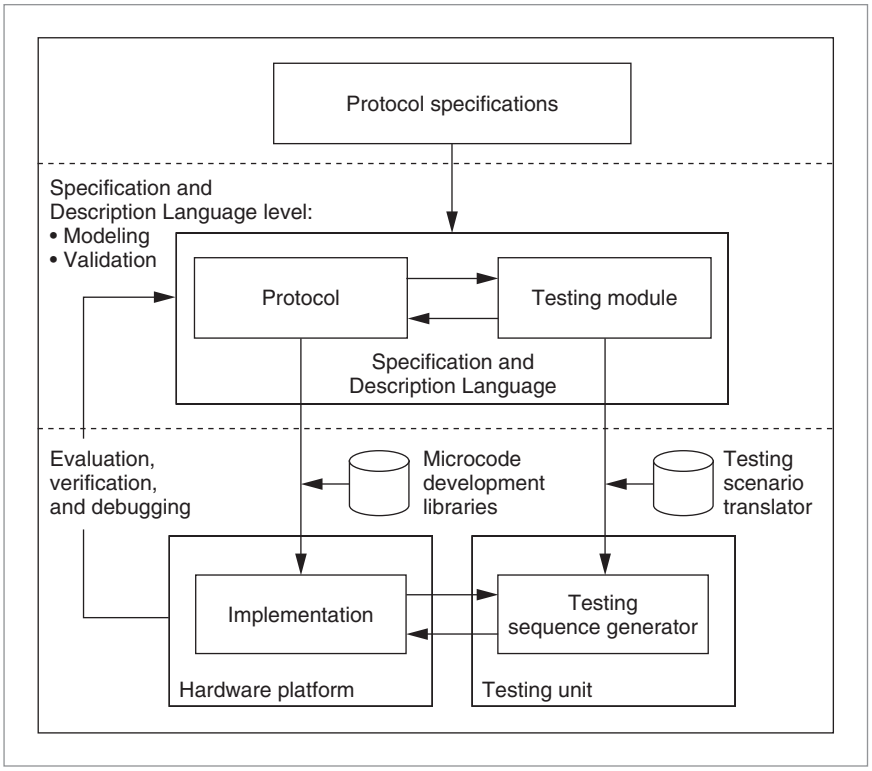

Figure 1. The three stages of protocol design and test methodology.

Protocol design and test methodology

Figure 1 shows protocol design and validation divided into three main stages. The first stage, specifications capture, provides the necessary information for determining the protocol's functional and behavioral requirements, and for estimating system resources (number and type of signals, messages, timers, and so on). At the second stage, we develop a formal model using SDL. With this model we can use structured and object-oriented methods to define the system's behavior in terms of finite state machines (FSMs).[4] We also validate the protocol's functionality using a testing module that emulates the behavior of the protocol's environment. To analyze the protocol's dynamic behavior and to evaluate error-prone cases, we must determine several test sequences that represent events generated in the SDL environment. On the basis of these test sequences, we execute different simulation scenarios and create the corresponding MSCs.

At the last stage, we have a verified protocol model and a set of testing scenarios that provide a complete test bench conforming to the initial specifications. By designing and functionally verifying the protocol in a platform-independent manner, we increase the reusability and extensibility of its implementation.

From the high-level modeling stage we move to the implementation stage. Using commercial tools, we automatically translate the verified protocol model into low-level executable code (C or C++). The final code is a stand-alone module that we can adapt to the target platform using platform-specific interfacing functions. Moreover, we also translate the testing module used at the SDL-level simulations into an equivalent testing sequence generator so that we can reuse the SDL-level test sequences to validate the protocol's implementation. To do this, we record the test sequences that initiated the scenarios in the SDL environment, along with the signals exchanged between the protocol model and its environment during simulation. A custom application then translates these test sequences into the appropriate format required by the final system testing environment, which is based on MatLab tools.

The MatLab environment communicates with the final system using a custom interface that provides data exchange and synchronization between the embedded

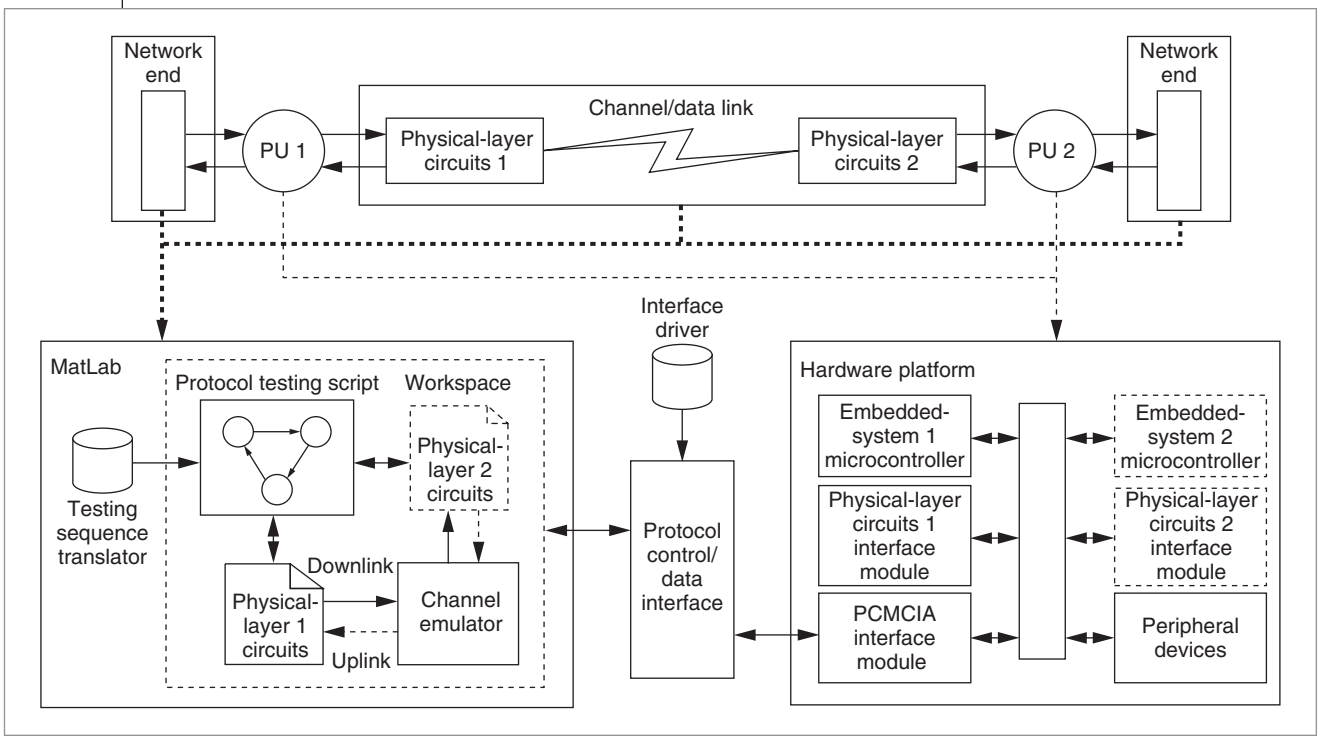

Figure 2. Protocol debugging and testing environment (bottom) and the connection of two protocol units (top). The dashed lines illustrate the modules needed for a testing environment that includes both protocol units. When a single protocol unit is used and the protocol testing script includes the rest of the testing environment functionality, the modules illustrated with dashed lines are not used.

processor and the MatLab workspace. As a result, MatLab scripts interact with the hardware platform and provide the testing sequences to the embedded protocol unit. Moreover, we can collect information on the protocol's execution progress as well as on the system status, and process it in the MatLab environment. This feature also enables debugging of the protocol's implementation to resolve any possible errors and to optimize the final microcode. We can also modify certain attributes in the protocol's model to improve the low-level microcode's behavior and generate a more efficient implementation. The developer's experience determines the time this procedure requires.

Protocol testing environment

The testing environment used in our protocol design and test methodology is based on MatLab tools. Figure 2 presents the debugging and testing setup, which includes a reprogrammable hardware platform with embedded processors for protocol execution, and the MatLab environment, which runs the scripts executing testing scenarios based on the SDL-level protocol modeling. The MatLab functions and the hardware circuits use a PCMCIA-based custom FPGA module to exchange data. This module extends the available memory space that MatLab can access via an appropriate I/O device driver, while the embedded processors access the module as a peripheral device. This architecture integrates the entire MatLab environment as a peripheral device that is transparent to the total hardware system. As a result, we can map the signals interfacing the protocol implementation and its environment into the MatLab workspace, and the protocol's engine can interact with user-developed custom scripts. Control and status signals synchronize the hardware system with MatLab. In our test bed, we have measured a data transfer rate of approximately 2 Mbps between the MatLab workspace and the FPGA interface module.

In this setup, the hardware platform and the MatLab environment constitute a complete functional system that lets us test a protocol's implementation. The top portion of Figure 2 shows a block diagram of an end-to-end connection between two protocol units (PUs). At the SDL level, an interface defined between the protocol model and its environment determines a complete set of data and control signals between the PU and the network end regarding protocol management and maintenance, as

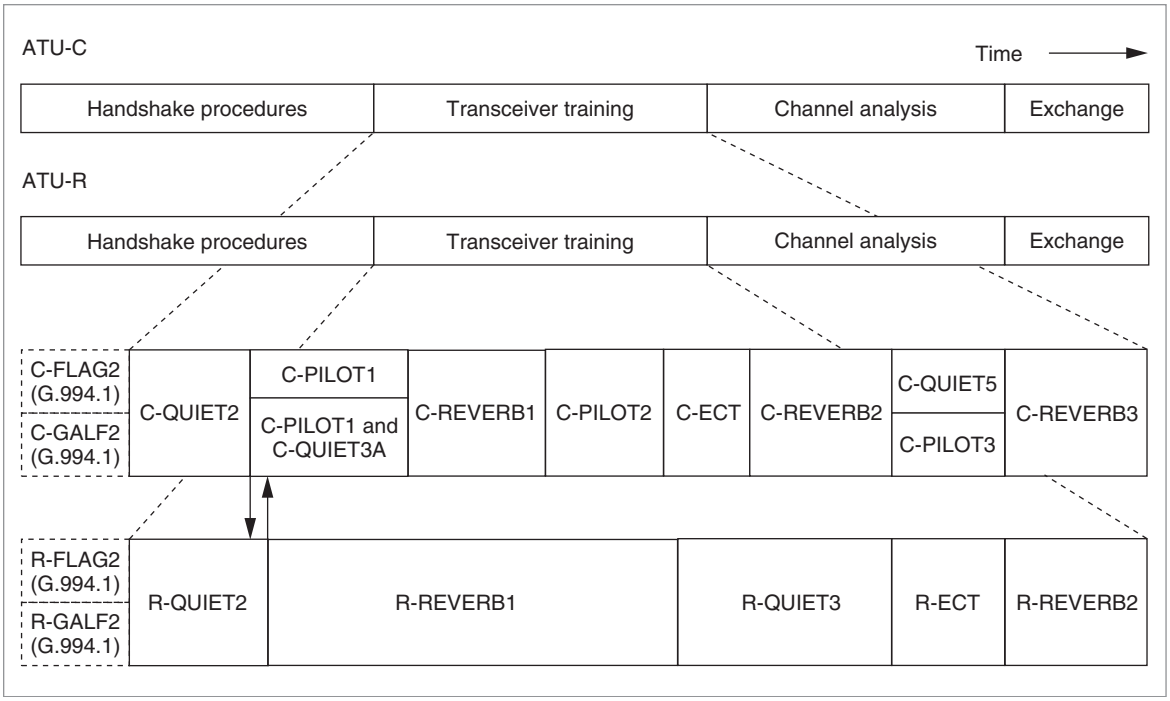

Figure 3. Asymmetric digital subscriber line signaling phases and detailed transition states of the transceiver training phase. (R = remote, C = central)

well as signals between the PU and the physical-layer circuits. These signals implement the handshake procedures over the real channel. We map the SDL protocol models into the system processors using various interface functions that reflect the processors' architecture, while the MatLab portion of the testing environment emulates the protocol's operating environment.

This testing environment supports the evaluation of either a single transceiver's PU or a complete configuration comprising both transceivers. In the first case, the MatLab environment simulates the physical-layer circuits, the transmission channel, and the far-end transceiver. In the second case, MatLab includes only the physical-layer circuits and the transmission channel. Moreover, we translate the testing scenarios executed in the SDL environment into MatLab scripts used for extensive validation of the platform implementation, letting the designer observe how implementation aspects such as timing and resource allocation affect the protocol's functional behavior, which we previously verified at the SDL level.

Protocol developers can use the methodology described in Figure 1, along with the testing setup of Figure 2, to develop and test any signaling protocol. The following sections describe our methodology for designing and testing the signaling protocol for ADSL modems.[5]

ADSL signaling protocol case

Recommendation ITU G.992.1[6] defines the procedures for initializing the ADSL transceiver at the user end, denoted ATU_R, and at the network operator end, denoted ATU_C. These procedures include the transmission of specific signals and messages in both directions that let each side determine certain communication channel attributes. On the basis of these attributes, both modems train their signal processing circuits (SPCs)—for example, equalizers, timing recovery, and automatic gain control units—and also establish certain transmission settings that maximize the achievable throughput. There are four discrete initialization phases:

- handshake procedures,
- transceiver training,
- channel analysis, and
- exchange.

Note that the handshake procedures phase (according to Recommendation ITU G.994.1)[7] is the common activation phase for every DSL technology (not considered in this article), while the other phases are ADSL specific.

Figure 3 shows the timeline of the ADSL signaling phases and includes the specific transition states of

the transceiver training phase. Recommendation G.992.1 defines the terminology used for state and signal description. The transition points, indicated by two vertical arrows, correspond to the first synchronization between ATU_C and ATU_R on the state sequence of the transceiver training phase. Throughout the initialization phases, timing recovery and synchronization between the far-end transceivers occur during certain signal transmission states. An interactive procedure for frame synchronization maintenance—based on message reception, message check, and message response—is also provided. Upon the report of a time-out error or a message checksum error, the initialization process resets to the handshake procedures phase. After the exchange phase, both modems enter into data transmission mode, usually referred to as "showtime." Thereafter, the transmission is based on the settings established and negotiated during initialization.

Protocol functional requirements

As Figure 3 indicates, the signaling protocol defines a specific set of transmission (Tx) and reception (Rx) states at each transceiver, as well as the transitions from one state to another. Each state is associated with specific signals and messages sent to or received from the far-end transceiver. Developers can implement the protocol on an FSM model that interacts with the logical circuits and SPCs to initiate transmission of the output-line signals or to receive any incoming-line signals. The protocol FSM is independent of the implementation details of the modem's circuits. The protocol FSM and the modem's circuits interact through a custom interface that enables the protocol state machine (PSM) to take control of the circuits during the modem's initialization procedures. At an abstract level, the protocol's functionality decomposes into the PSM module, the SPC modules, and the appropriate interface between these modules (defined in the form of signals, variables, and status/control registers).

Protocol model

The Tau SDL Suite by Telelogic is a commercial tool that supports development and validation of SDL models using simulation.[8] We used this tool to design the ADSL signaling protocol as a PSM module. We developed two distinct systems, one for the ATU_C side and the other for the ATU_R side. To validate the protocol using simulation, we also developed an intermediate auxiliary testing module. This module emulates the response of the far-end modem regarding message exchange through the transmission channel, and the response of the SPC circuits regarding signal generation and signal processing.

We implemented the model's behavioral description using communicating state machines represented by logical processes. Signals and messages between these processes and the system model's environment determine communication. These signals and messages are based on the interface determined between the PSM and the SPC modules.

We determined three types of interface signals. *Timing and synchronization signals* ensure that the procedures and state transitions occur precisely as defined in ITU Recommendation G.992.1. The *circuit control signals* correspond to the initialization and control of the modem's SPCs, while the *ADSL signals* and messages carry the various types of signals and parameters that the modems exchange.

Figure 4 shows the top-level abstract realization of the SDL model associated with ATU_C, and it also shows the channel emulator. We can distinguish example signals related to user data information (atu_pars), to timing and synchronization procedures (atu_timer and s_cnt), to circuit control (agc_on and ec_on), and to signal exchange (tx_signal and rx_msg).

Two processes implement the transition states defined at each phase: One handles transmission, and the other handles reception of the signals sent by the far-end side. These processes interact using internal signals for synchronization and error recognition. Figure 5 shows the ATU_C PSM module's internal structure. Both processes include error monitoring for time-out conditions, message checksum errors, and invalid parameter reception. In case of error detection, the PSM modules provide diagnostic messages as they enter the corresponding error state.

Simulations with a complete set of message exchange scenarios—both erroneous and error free—verify the SDL models at both sides. The simulation environment is useful for testing not only the SDL model's functionality but also its interaction with the environment and another SDL model during runtime. We executed numerous test sequences and studied PSM module behavior for both sides in detail. In the absence of the real SPCs, the model-level channel simulator made the SPCs' interaction feasible. Figure 6 is an MSC diagram depicting signal exchange and timing synchronization between ATU_C and ATU_R.

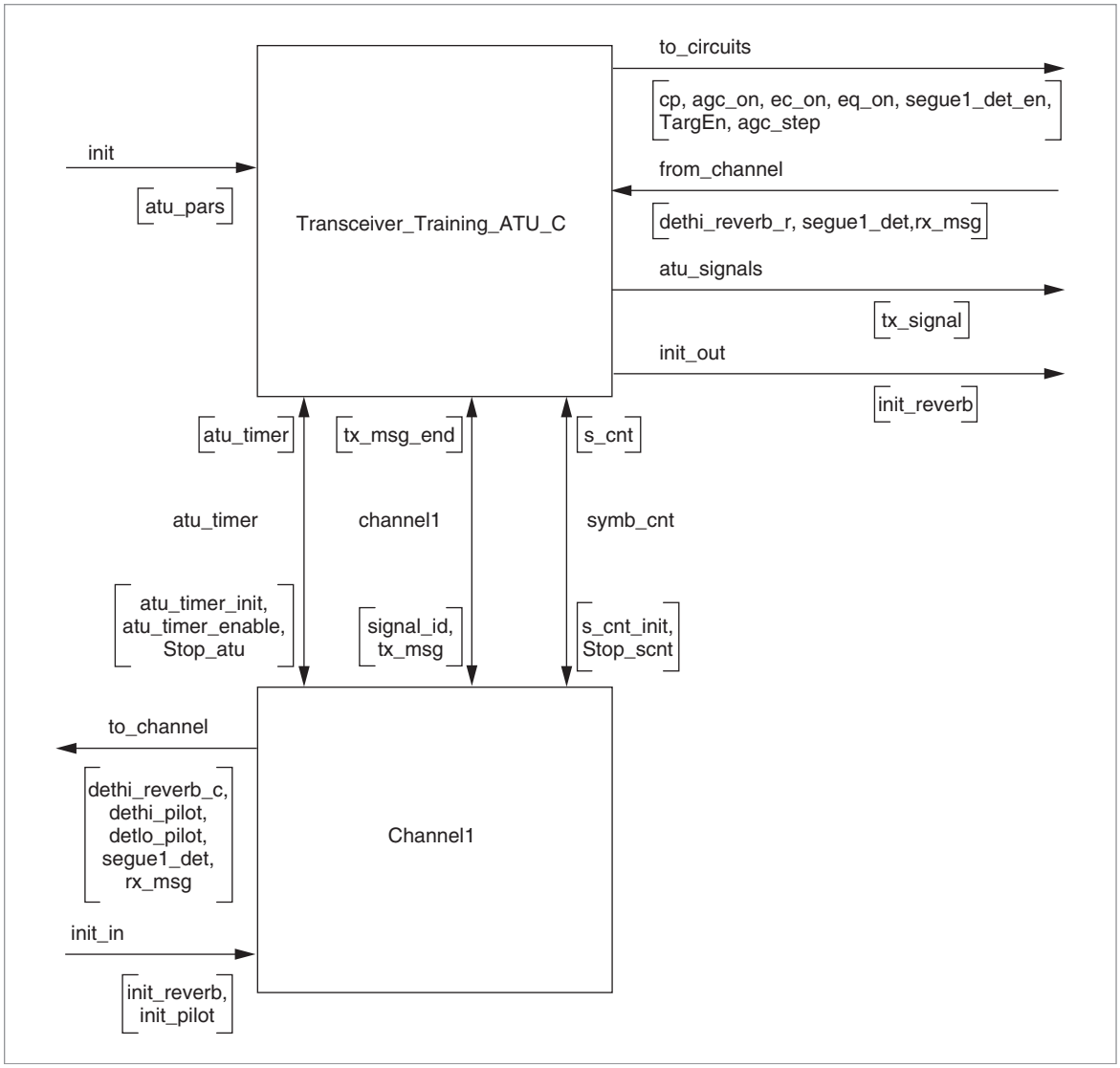

Figure 4. Protocol state machine module of ATU_C, with the channel emulator.

Embedded-microcode development

The SDL protocol models generated executable C code using the parser and certain translation libraries supplied by Telelogic, along with a standard C compiler from Microsoft. The resulting code reflects the SDL's structured features and is not highly optimized, but its major advantage is that with the development of the proper interfacing routines, it is integratable into virtually any platform. Telelogic also provides basic function libraries that facilitate the integration of an SDL model either with an operating system or in a hardware platform with a microprocessor and its peripherals. There are two distinct integration methods: *tight integration* interfaces the generated code directly to the runtime operating system so that the operating system can handle SDL process scheduling, memory management, timer handling, and so on. *Light integration* runs the generated code on a primitive SDL runtime kernel and has minimal or no interaction with an operating system.

The prototype environment uses the light integration model, shown in Figure 7. SDL system execution relies on a primitive scheduler function, which permits each SDL process in turn to execute one transition. Signals sent between processes go into the receiving process' input queue, and custom environment functions handle signals into and out of the SDL system. The basic environment functions are xInitEnv, xInEnv, and xOutEnv. Function xInitEnv relates to the system initialization procedures. Function xInEnv handles signals sent into the SDL system *from* the environment, while

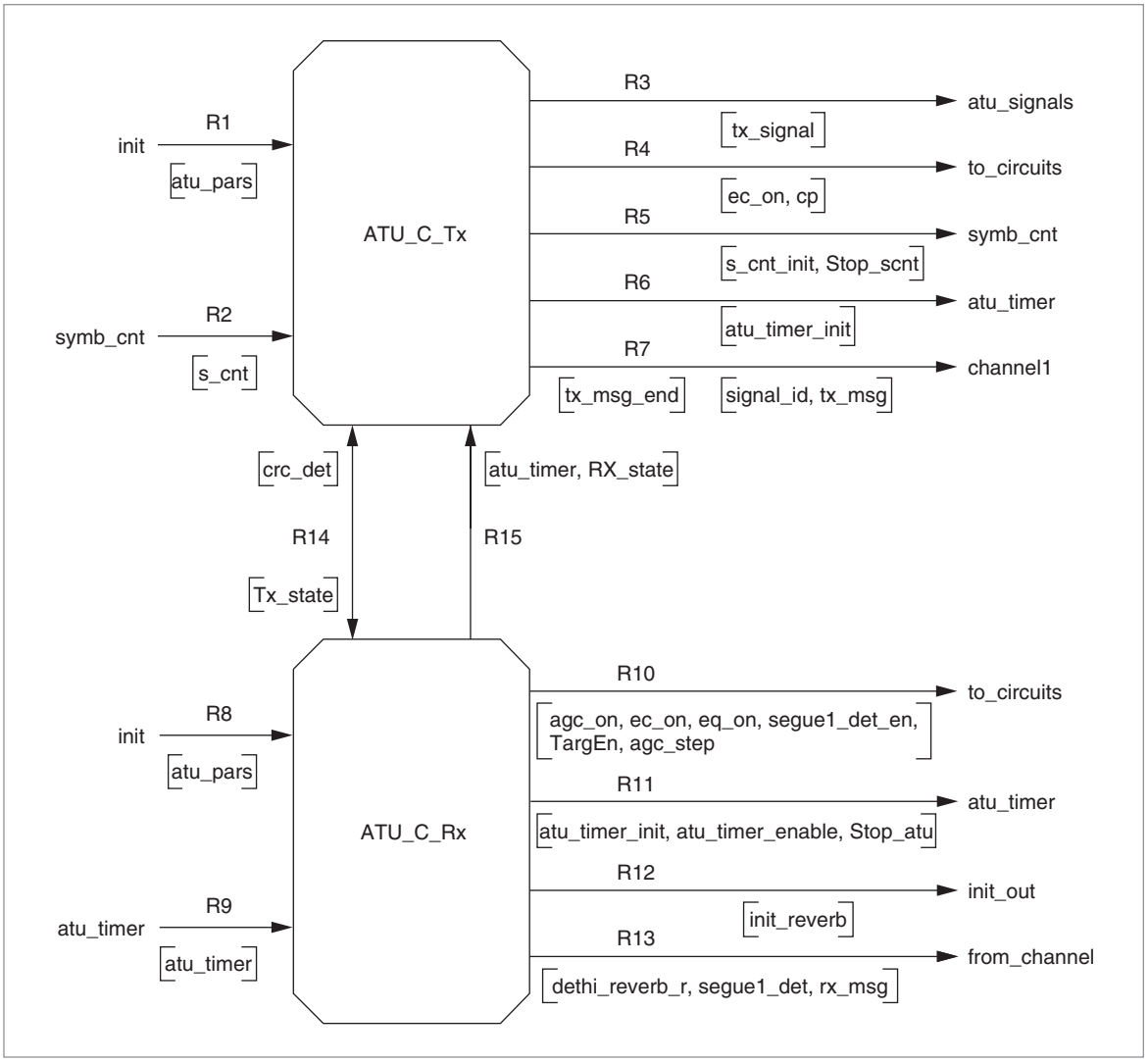

Figure 5. Top-level structure of the ATU_C protocol state machine module.

the xOutEnv function handles signals sent by the SDL system *to* the environment. The scheduler function calls xInEnv and xOutEnv iteratively but can serve only one incoming and one outgoing signal in each iteration. A queue dedicated exclusively to the SDL system environment stores additional signals.

As Figure 2 shows, the protocol's developer simulates the PU's environment in MatLab using custom scripts generated from the testing processes developed in SDL. The MatLab workspace is memory mapped as a peripheral device in the hardware architecture via a custom FPGA module that enables the implementation to exchange control and data with its environment. Functions xInEnv and xOutEnv enable this information exchange.

Figure 8 shows an example from the ATU_C SDL FSM, which invokes the output of a tx_msg signal carrying message parameters; the figure includes the message's translation into low-level C code. The Telelogic tools and the Microsoft C compiler automatically generate the C structure definition for the signal, along with the respective code for the signal's output and the memory allocation for its parameters. Function SDL_OUTP_PAR_ENV adds the signal to the queue dedicated to the environment.

The user-defined xOutEnv routine provides the signal's actual output. In the case of tx_msg, this routine accepts the signal's parameters and maps them in the peripheral module's dedicated memory location so that the testing environment has information about the FSM's status. It is therefore obvious that, apart from the task that represents the SDL module (which has been extensively tested in the SDL application environment),

the environment functions are also important because proper communication between the several modules in the prototype setup depends on their functionality. Moreover, custom routines consistent with the target microprocessor's characteristics handle dynamic memory allocation or deallocation. As soon as all code components are available, the tools provided for the specific processor compile and link them. To produce the respective stand-alone executable code, users must create an appropriate *makefile* that defines the rules for this process.

System architecture

Our application uses the soft-IP MicroBlaze core embedded processor.[9] We implemented the hardware platform on a 1-million-gate Virtex-II FPGA device. The hardware architecture is based on the CoreConnect bus and includes additional peripheral devices, along with the SPC interface module. The final architecture includes two separate embedded systems, one for the ATU_C side and the other for the ATU_R side. Each system includes a MicroBlaze soft processor core, an on-chip block of RAM, CoreConnect bus interconnections, and on-chip peripheral bus (OPB) based peripheral devices.[10] The SPC interfaces are physically assigned to a dual-port RAM (DPRAM) peripheral module. This module includes the interface logic for

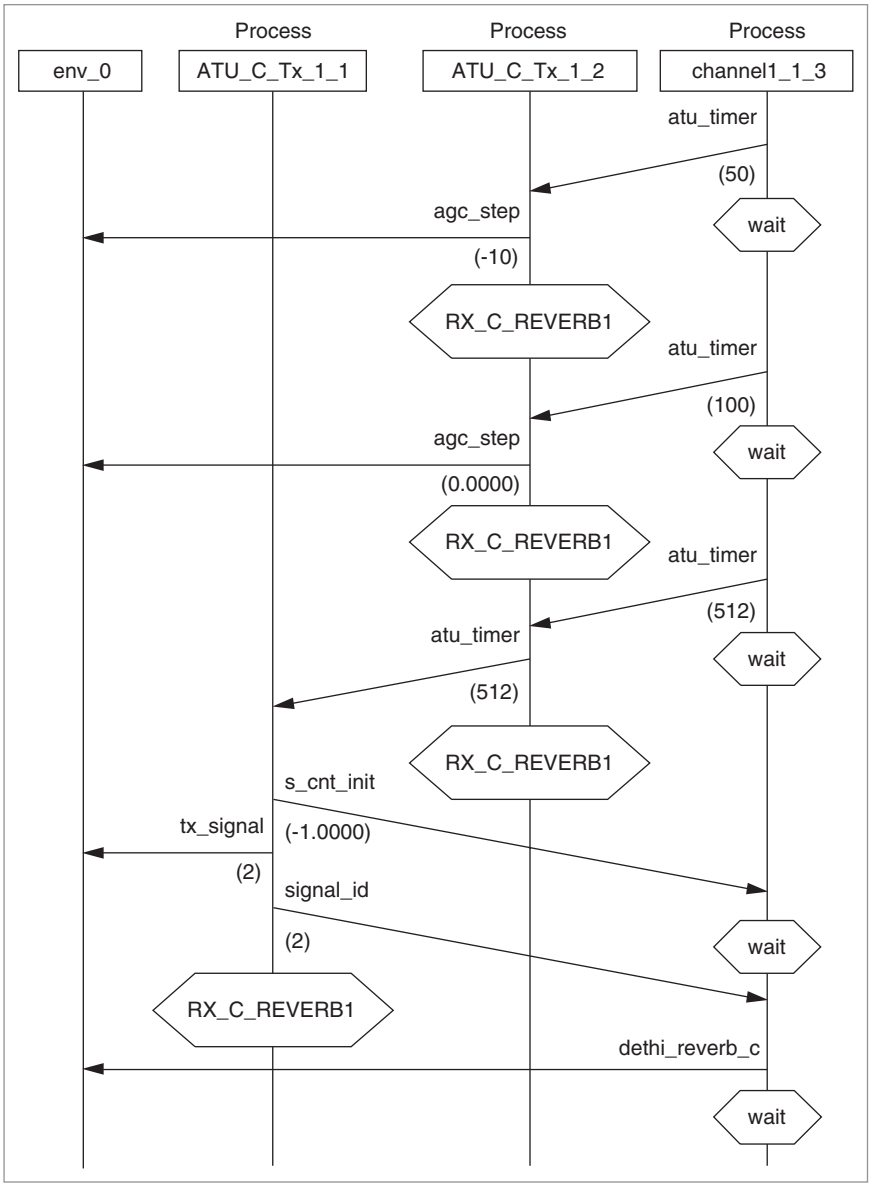

Figure 6. Example of message sequence chart state transitions.

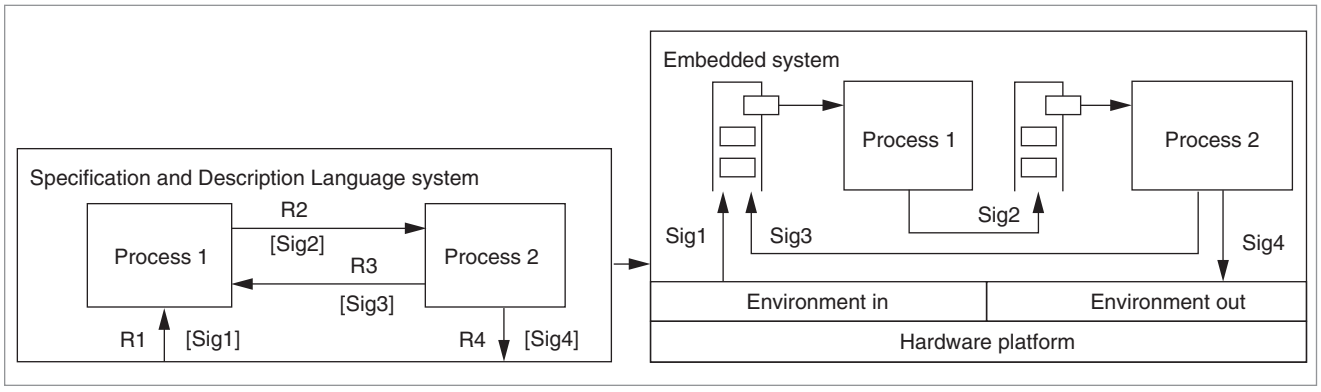

Figure 7. Specification and Description Language model translation to low-level microcode using the light-integration model.

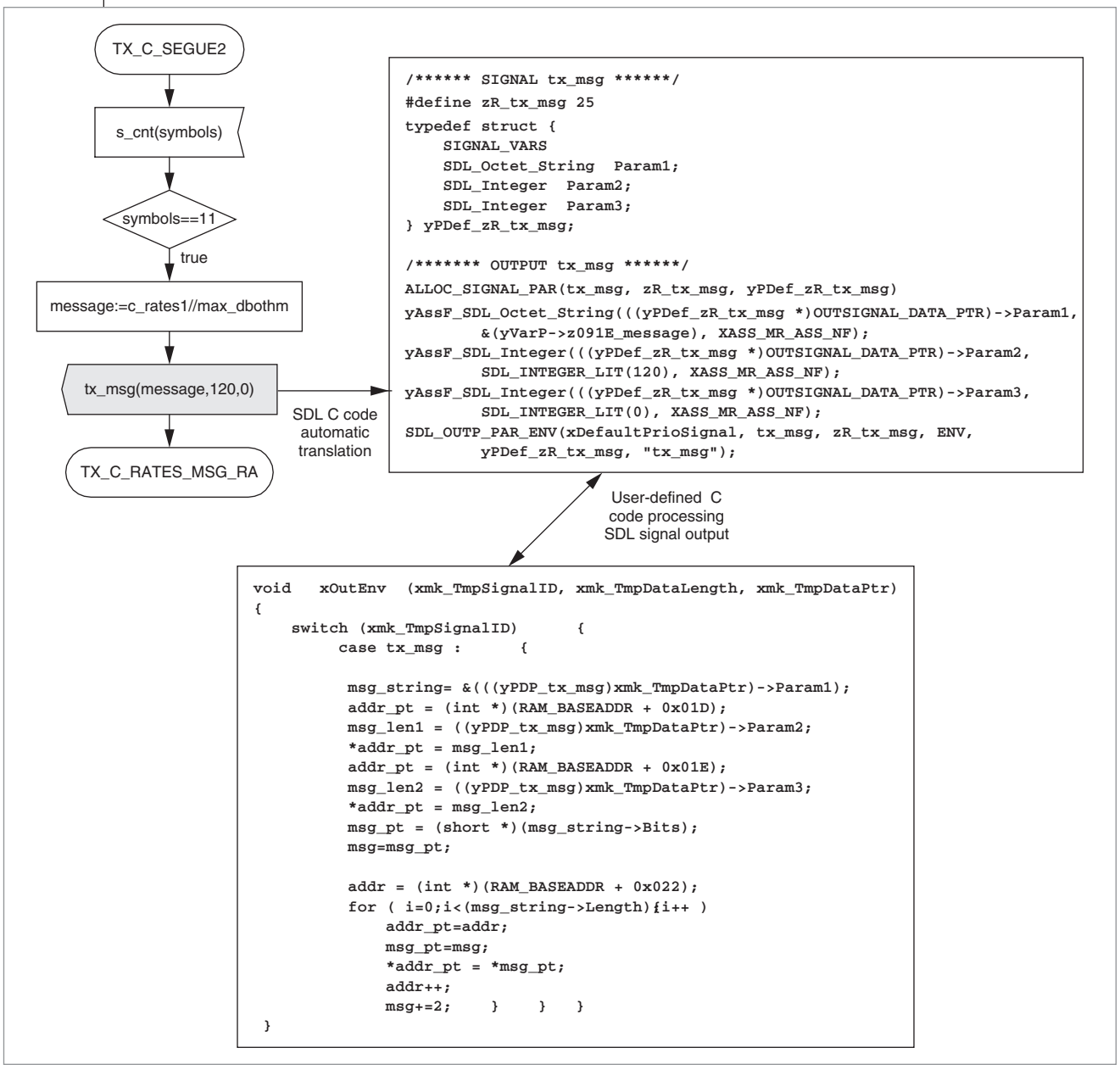

Figure 8. Low-level implementation of a Specification and Description Language signal.

its connection to the two separate OPBs as a standard peripheral device. Although the DPRAM behaves as a single memory module at the PCMCIA interface, we implemented it as two separate memories with distinct buses, one for each processor. Each processor is assigned to a different memory space organized into three distinct functional regions:

- The control memory region contains control information and data for configuring the hardware modules and for initializing communication between the MicroBlaze processor and the testing applications.
- The SPC memory region contains the data variables used for the interactions among the PSM and the system's SPCs.
- The user data memory region contains the data variables that determine certain signaling-protocol parameters, defined during system initialization.

The embedded processor accesses the DPRAM during protocol execution; the host computer accesses it for interactions with the protocol FSMs. Adding several

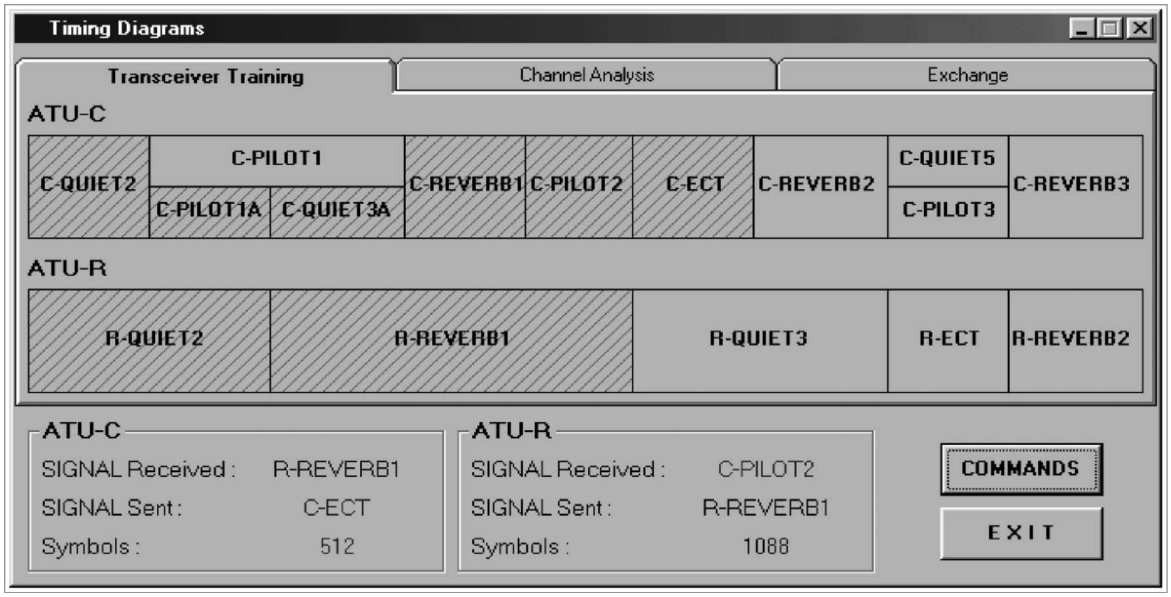

Figure 9. Graphical environment for testing the protocol's execution progress.

hardware modules from the ADSL data pump lets us gradually extend the hardware architecture of this development environment to the general ADSL system. We can design these modules as standard OPB devices, so that they communicate directly with the MicroBlaze processor. Or, they can be stand-alone devices.

Testing the implementation

Testing the protocol's implementation requires a channel simulator that interacts with the embedded PSM modules. So we developed a MatLab-based testing module that simulates the data path from the PSM modules' output to the SPCs and the transmission channel. We used the SDL-level simulations to generate the testing sequences between the protocol units and the physical-layer circuits and implemented these sequences in custom MatLab scripts. These scripts interact with the SPC interface module in the hardware platform to appropriately process and respond to the signals generated by each PSM module.

We modeled the transmission channel in MatLab as a separate custom module that interacts with the scripts generated by the SDL level; this provides a complete functional testing module.

We simulated the channel as a symmetric channel that modifies the exchanged data with a user-defined bit-error-rate probability. This lets us test the PSM response in case of message cycle-redundancy-check (CRC) errors or other data pattern violations. Moreover, a suitable configuration of the testing scenarios also tests time-out conditions and illegal or unaccepted parameter negotiations.

We developed an additional graphical application for visualizing the protocol execution and providing diagnostic messages. Plain state diagrams with system information graphically represent the state transitions and protocol execution, according to the interactions between the PSM module and the testing environment. This application is also responsible for acquiring and displaying system parameters exchanged between the two protocol units, along with timing information regarding state transitions. Moreover, it lets users select among available test scenarios by choosing the channel conditions to emulate in MatLab. Whenever the PSM module indicates system failure, the application produces an appropriate diagnostic message.

Graphics manipulation is a resource-consuming process, and MatLab scripts must maintain real-time communication with the hardware platform. Therefore, because we use the graphical application only for observation, we did not implement it in the MatLab environment with a GUI but rather as a stand-alone application designed using Visual Basic. Figure 9, an example of the graphical environment, shows the progress of the protocol's transceiver training phase for the ATU_C and the ATU_R transceivers. The areas with diagonal lines correspond to the already accomplished protocol stages of each PSM. The complete signaling procedures execute entirely in the hardware system, and the channel emulator enables the interactions

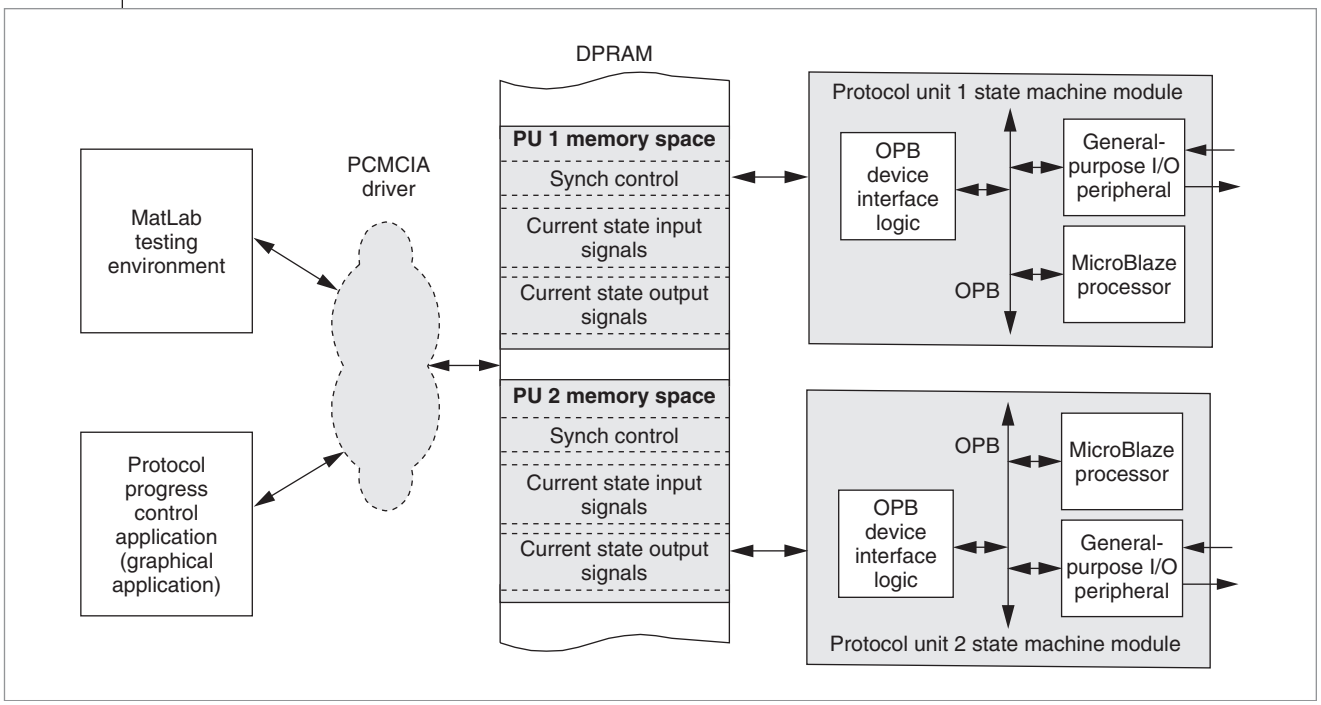

Figure 10. System architecture and data exchange between the applications environment and the protocol modules.

between the two separate modems. This application can serve as a template for easily reproducing and designing graphical applications suitable for testing other signaling protocols.

A set of variables stored in the DPRAM enables the transaction of information between the testing applications and the embedded processors. This set includes synchronization and control information for the data exchange process, and protocol data that constitutes the input and output signals of each protocol phase. Figure 10 depicts data exchange between the MatLab testing module, the graphical application, and the PSM modules. A control field synchronizes the PSM modules' execution timing reference with the execution delay of the emulator-based operations. We use this control field to identify any protocol state transition, read new state output data from memory, process data according to the current state, and write new state input data to memory.

Experimental results

Using the approach described, we exhaustively tested the implemented FSMs of the ADSL signaling protocol and collected statistics on various error conditions. Test cases resulting in error detection include CRC errors, time-out occurrences, and parameter failures. In these test cases, we deliberately inserted noise and errors in the exchanged signals and messages, using either a dialog menu in the graphical application or an automatic test generator during system testing. We measured response times during each test case so that we could collect statistical data on the actual time needed for each state transition, and we measured each scenario's total execution time. To verify the efficiency of the protocol's implementation, we compared these results with the time specifications of ITU Recommendation G.992.1. Table 1 summarizes the error and time statistics collected at both transceivers, along with the parameters used during testing. Total execution time includes all possible scenarios, both successful and erroneous. (We omitted the handshake phase because it's not ADSL specific.)

During final system testing, the application software requires additional time to retrieve the collected data, reinitialize the state machines, and start the next testing scenario. This reconfiguration procedure takes a few seconds (depending on the host processor's speed), so the total time required for exhaustive system testing is 16 to 20 minutes for 140 test cases. After certifying that the implemented protocol is in total accordance with the timing specifications of Recommendation ITU G.992.1, users can execute additional tests by adding noise in the channel and collecting statistics about erroneous or successful executions.

Table 1. Asymmetric digital subscriber line signaling: parameters and testing results, by initialization phase.

Parameters	Transceiver training		Channel analysis		Exchange	
No. of states, ATU_C	11		6		16	
ATU_R	5		9		17	
No. of CRC errors, ATU_C	0		2		5	
ATU_R	0		2		5	
No. of time-outs, ATU_C	4		0		3	
ATU_R	2		1		3	
No. of parameter failures, ATU_C	0		24		3	
ATU_R	0		40		46	
	Time statistics (seconds)					
	Min.	Max.	Min.	Max.	Min.	Max.
ATU_C successful execution	1.81	3.16	4.30	4.30	0.42	3.51
ATU_R successful execution	1.81	3.64	4.45	5.17	0.27	3.10
ATU_C total execution	4.45	7.14	14.00	32.62	5.15	18.21
ATU_R total execution	5.48	9.14	16.00	66.70	14.64	91.20

VALIDATING AND TESTING communication protocol implementations is not trivial. Developing the testing environment itself required a lot of effort, but it is easily reusable for developing other protocols and for extensively testing their implementations. Overall, this means decreased design and test time compared with direct implementation of the protocols on a hardware platform.

As a future extension of our protocol testing methodology, we are looking at ways to substitute other high-level tools for modeling and implementing the protocol state machines in place of the SDL environment. It would be beneficial to have a unified environment that allows multilevel system development and testing, an environment based on a complete system model that includes protocol execution units and physical-layer circuits. We previously described an initial approach for developing and testing physical-layer circuits.[11] ■

Acknowledgments

This work was partially supported by the University of Patras' Karatheodoris R&D program and the Greek Ministry of Industry's Project 00BE33, "Digital Subscriber Lines Technology."

References

1. *IEEE P802.11, Draft Standard for Wireless LAN Medium Access Control and Physical Layer Specification*, IEEE, 1997.
2. *Recommendation ITU-T Z.100, Specification and Description Language*, Int'l Telecommunication Union, 1999.
3. *Recommendation ITU-T Z.120, Message Sequence Chart*, Int'l Telecommunication Union, 1999.
4. S.A. Edwards, *Languages for Digital Embedded Systems*, Kluwer Academic Publishers, 2000.
5. T. Starr, J.M. Cioffi, and P.J. Silverman, *Understanding Digital Subscriber Line Technology*, Prentice Hall, 1999.
6. *Recommendation ITU G.992.1, Asymmetrical Digital Subscriber Line Transceivers*, Int'l Telecommunication Union, 1999.
7. *Recommendation ITU G.994.1, Handshake Procedures for Digital Subscriber Line Transceivers*, Int'l Telecommunication Union, 1999.
8. *Telelogic Tau SDL Suite User's Manual*, Telelogic AB, 2001.
9. *MicroBlaze Hardware Reference Guide*, Xilinx Inc., 2002.
10. *On-Chip Peripheral Bus: Architecture Specifications*, Version 2.1, IBM, 2001.
11. M. Varsamou et al., "From Matlab/Simulink Models to Prototype Implementation: A Communication Systems Development Environment," *Proc. Nordic MatLab Conf.*, ComSol (http://www.comsol.dk), 2003, pp. 160-163.

Maria Varsamou is a graduate student in the Department of Electrical Engineering and Computer Technology at the University of Patras, Greece. She participates in various R&D projects with European industries. Her research interests include digital communications with an emphasis on error control coding. Varsamou earned a BS in electrical engineering from the University of Patras. She is a member of the Technical Chamber of Greece.

Nikolaos Papandreou is a graduate student in the Department of Electrical Engineering and Computer Technology at the University of Patras, Greece, and a research engineer with the Communications and Embedded Systems Group of the Computer Technology Institute in Greece. His research interests include digital communications with an emphasis on performance analysis and rapid prototyping. Papandreou earned a BS in electrical engineering from the University of Patras. He is a member of the IEEE and the Technical Chamber of Greece.

Theodore Antonakopoulos is an associate professor in the Department of Electrical Engineering and Computer Technology, University of Patras. He participates in several R&D projects with European industries. His research interests include digital communications with an emphasis on performance analysis, efficient hardware implementation, and rapid prototyping. Antonakopoulos has a PhD from the University of Patras. He is a senior member of the IEEE and a member of the Technical Chamber of Greece.

■ Direct questions and comments about this article to Theodore A. Antonakopoulos, University of Patras, Department of Electrical Engineering and Computer Technology, 26500 Rio, Patras, Greece.