# FIBRE CHANNEL PROTOCOL: FORMAL SPECIFICATION AND VERIFICATION

Vijay Nagasamy Sreeranga Rajan† Preeti Ranjan Panda‡

Software R&D, LSI Logic Corporation, Milpitas, CA 95035.
†Computer Science Laboratory, SRI International, Menlo Park, CA 94025.
‡Dept. of Information and Computer Science, UC Irvine, CA 92717.

**Abstract**

*The complexity of the Fibre Channel protocol poses a challenge to design its implementation "right-the-first-time". In this work we have investigated the formal specification and mechanical verification of the Fibre Channel Protocol to address issues of correctness. We present abstract models of the FC-2 layer protocols in SpecCharts and the automated verification of a portion of the fibre channel node loop core implementation in a mechanical verification system called PVS. We show that this approach can supplement traditional simulation based approaches to obtain better coverage and confidence during verification.*

## 1. Introduction

In this work we investigate the formal specification and efficient mechanical verification of a Fibre Channel Protocol core. The ANSI standards document for Fibre Channel [1] contains a good description of the protocol, but it is not in a machine readable or executable format. Hence the document can only serve as a guide for manually checking whether an implementation satisfies the standards or not. Manual checking is tedious and error prone. We therefore provide a formal specification of the protocol and the core implementation in a mechanical verification system, and show how this can be used to check whether the core satisfies certain properties stipulated by the protocol standard. We believe that this approach can supplement the traditional simulation based verification paradigm to provide a greater degree of confidence in the correctness of the design specification and implementation.

The rest of the paper is organized as follows. In Section 2, we briefly describe the Fibre Channel standard. We present an abstract specification of the protocol in section 3. In Section 4, we briefly describe the Prototype Verification System (PVS), a mechanical verification system which allows properties related to the correctness of the specification to be semi-automatically verified. Section 5 contains an example illustrating the verification of various properties of a loop port state machine, a part of the Fibre Channel protocol specification. We show that with our approach the designer can obtain a greater degree of confidence towards a "right-the-first-time" concrete design implementation. We summarize the results and suggest directions for future work in section 6.

## 2. The Fibre Channel Standard

Fibre Channel [1] is an integrated set of standards being developed by the American National Standards Institute (ANSI) to address the definition of a practical, inexpensive, yet expandable method for achieving high-speed data transfer among a variety of platforms like workstations, mainframes, supercomputers, desktop computers, storage devices and display devices. It specifically addresses the need for very fast transfers of large volumes of data.

The Fibre Channel standard is considered a common backbone underlying and supporting multiple high-level protocols like SCSI (Small Computer System Interface) and HIPPI (High Performance Parallel Interface.) The standard supports multiple cost/performance levels, from small systems to supercomputers. An important performance requirement is data transmission at rates ranging from 133 megabits per second to 1.0625 gigabits per second.

The Fibre Channel standard is organized into five functional levels, labelled FC-0 though FC-4. They are described in brief below.

- **FC-0** deals with the physical portions of the Fibre Channel, including electrical and optical characteristics of the media.
- **FC-1** defines the transmission protocol and encoding/decoding schemes for combining data with clock.
- **FC-2** defines the signaling and framing protocol, including the layout of the frames, the content and use of frame header.

- **FC-3** defines common services that may be available across multiple ports in a node.
- **FC-4** defines the mapping between the lower levels of the Fibre Channel and the ULPs (Upper Level Protocols) like SCSI and HIPPI.

The specification for FC-0, FC-1 and FC-2 levels of the Protocol is described in the Fibre Channel Physical and Signalling Interface (FC-PH) [1] document. The protocol for FC-2, which forms the bulk of the specification, is modeled in the present work. The Fibre Channel Implementation Guide (FC-IG) [2] along with specific inputs from the design architect was used in developing the FSM structures of the model.

## 3. Modeling the FC-2 Protocol

In this section, we describe the language and tools used to model the FC-2 protocols that helped us gain a better understanding of the Fibre Channel standard.

We chose the SpecCharts [4] language to develop the initial abstract specification. SpecCharts was developed at the University of California, Irvine to aid system-level specification and design. It is based on the *Program-State-Machine* (PSM) model and is implemented as an extension to VHDL (VHSIC Hardware Description Language). A PSM is a heterogeneous model that integrates hierarchical concurrent finite-state machines with a programming language paradigm. The model basically consists of a hierarchy of *program-states*, in which each program-state represents a distinct mode of computation. At any given time only a subset of the program-states will be active. The control flow amongst program-states can be either sequential or concurrent depending on the behavior. The functionality of the leaf-level program-states are specified using sequential statements in VHDL. More information on SpecCharts can be found in [4].

We started the first phase of this project by creating models for the FC-2 layer protocol in SpecCharts. The modeling was done at the frame, sequence and exchange levels of abstraction. Figure 1 shows a portion of the SpecCharts description. The top level entity called "FibreChannel" is described along with input and output ports in a syntax similar to VHDL. This is followed by a description of the architecture of the entity with all the global constants and signals in the architecture declarative segment. The architecture itself consists of six concurrent program-states. These states include N_Port_Login, Exchange Management, and Sequence Management (with class 1, 2 and 3 services) as shown in Figure 1.

The N_Port_Login program-state handles the transmission and reception of Login frames. (These frames are necessary to exchange information relating to buffer size of nodes, classes of service supported by nodes, etc.) The Exchange Management program-state handles the transmission and reception of complete sequences. It triggers the Sequence Management program-states if the host intends to transmit a sequence, and is indicated to by the Sequence Management states upon reassembly of a complete sequence. The Sequence Management program-states are responsible for performing the segmentation and reassembly of frames. Acknowledgment of frames received is also handled in these states.

Each of these concurrent program-states is in turn modeled as a set of sequential sub-behaviors. For example the N_Port_login program-state shown in Figure 1 is composed of seven sub-behaviors; viz. Logout, Initiate, ReInitiate, WaitForAccept, IndicateStatusToHost, N_Logged_In and Accept. These sub-behaviors are sequential because at any given instant of time, only one of the behaviors can be active. The behavior of each sequential state is described using sequential statements in a VHDL like syntax similar to the one for "IndicateStatusToHost" shown at the bottom of Figure 1.

Transitions amongst these sub-behaviors are specified explicitly in the SpecCharts descriptions using a concise notation. For example, in case of the "Logout" state, the following description

```
(TOC, NportRequest = '1' and
        NportRequestType = N_PORT_LOGIN_REQUEST, Initiate)
```

means that on completion of task (indicated by TOC), the machine enters state "Initiate" iff signal "NportRequest" is set and "NportRequestType" equals N_PORT_LOGIN_REQUEST.

In our experience, we found the SpecCharts models resulted in concise, intuitive and readable descriptions of the protocols. The SpecCharts descriptions were automatically translated to VHDL, since the syntax is quite similar. The resulting VHDL descriptions were used to simulate the models to verify correctness. These descriptions can also be used as system models to generate stimuli for verifying synthesizable implementation code written in Verilog or VHDL.

One of main problems with simulation based verification is that it is not practical to cover all possible cases especially in designs consisting of a hierarchy of concurrent complex communicating FSMs. Formal verification is well suited to addressing such problems. In the next phase of the project, which is currently in progress, we are using PVS (described in section 4) to formally verify the protocol models.

```
entity FibreChannel is
port            (
                -- INPUT PORTS
                clock: IN BIT;
                NPortRequest: IN BIT; NportRequestType: IN INTEGER; NewFrameReceived: IN BIT;
                FromHost_SID: IN INTEGER;
                . . . . . .

                -- OUTPUT PORTS
                ToHost_P_RJT_REASON: OUT INTEGER; ToHost_TransmitStatus: OUT BIT;
                TransmitNewFrame: OUT BIT
                . . . . . .
                );
end;

Architecture SpecChart of FibreChannel is
begin
behavior StateMachine type concurrent subbehaviors is

constant        IDLE_FRAME: INTEGER := 0;
constant        DATA_FRAME: INTEGER := 1;
constant        P_RJT_FRAME: INTEGER := 2;
. . . . . .
-- Following are Concurrent states at the top level
begin
                N_Port_Login:;
                Exchange_Mgmt:;
                Class12_Sequence_Intiator:;
                Class12_Sequence_Recipient:;
                Class3_Sequence_Intiator:;
                Class3_Sequence_Recipient:;

-- Description of N_Port_Login FSM. It consists of 7 sub-behaviors
behavior N_Port_Login type sequential subbehaviors is
begin
                Logout:                 (TOC, NportRequest = ‘1’ and NportRequestType = N_PORT_LOGIN_REQUEST, Initiate),
                                        (TOC, NewFrameReceived = ‘1’ and rframe_Type = P_LOGI_FRAME, Accept);
                Initiate:               (TOC, true, WaitForAccept);
                ReInitiate:             (TOC, true, WaitForAccept);
                WaitForAccept:          (TOC, (NportRequest’event and NportRequest = ‘1’ and
                                        NportRequestType = N_PORT_LOGIN_REQUEST)
                                        or (NewFrameReceived = ‘1’ and rframe_Type = LSCR_FRAME), Logout),
                                        (TOC, NewFrameReceived = ‘1’ and rframe_Type = P_RJT_FRAME
                                        and rframe_P_RJT_REASON = CLASS_NOT_SUPPORTED_ERROR, ReInitiate),
                                        (TOC, NewFrameReceived = ‘1’ and rframe_Type = P_RJT_FRAME
                                        and rframe_P_RJT_REASON /= CLASS_NOT_SUPPORTED_ERROR, IndicateStatusToHost),
                                        (TOC, NewFrameReceived = ‘1’ and rframe_Type = P_LOGI_FRAME
                                        and rframe_SID > FromHost_SID, Logout),
                                        (TOC, NewFrameReceived = ‘1’ and rframe_Type = ACC_FRAME, N_Logged_In);
                IndicateStatusToHost:   (TOC, true, Logout);
                N_Logged_In:            (TOC, NewFrameReceived = ‘1’ and rframe_Type = P_RJT_FRAME
                                        and rframe_P_RJT_REASON = LOGIN_REQUIRED_ERROR, IndicateStatusToHost),
                                        (TOC, (NportRequest = ‘1’ and NportRequestType = LSCR_REQUEST)
                                        or (NewFrameReceived = ‘1’ and rframe_Type = LSCR_FRAME)
                                        or (NportRequest = ‘1’ and NportRequestType = N_PORT_LOGOUT_REQUEST), Logout),
                                        (TOC, NewFrameReceived = ‘1’ and rframe_Type = P_LOGO_FRAME, Logout);
                Accept:                 (TOC, True, N_Logged_In);

                . . . . . .

                -- Description of the behavior for IndicateStatusToHost
                behavior IndicateStatusToHost type code is
                begin
                                        ToHost_TransmitStatus <= ‘0’;
                                        ToHost_P_RJT_REASON <= rframe_P_RJT_REASON;
                end IndicateStatusToHost;

. . . . . .

end StateMachine;
end SpecChart;
```

Figure 1. Portion of a SpecCharts description for the FC-2 layer protocol

## 4. Prototype Verification System (PVS)

The Prototype Verification System (PVS) [5] is an environment for specifying entities such as hardware/software models and algorithms, and verifying properties associated with the entities. An entity is usually specified by asserting a small number of general properties that are known to be true. These known properties are then used to derive other desired properties. The process of verification involves checking relationships that are supposed to hold among entities. The checking is done by comparing the specified properties of the entities.

PVS has been used for reasoning in many domains, such as in hardware verification [6,7,8], protocol verification, and algorithm verification [9]. We briefly give the features of the PVS specification language in Section 4.1 and the PVS verification features in Section 4.2.

### *4.1 PVS Specification Language*

The specification language features common programming language constructs such as arrays, functions, and records. It has built-in types for reals, integers, naturals, and lists. A type is interpreted as a set of values. One can introduce new types by explicitly defining the set of values, or indicating the set of values, by providing properties that have to be satisfied by the values. The language also allows hierarchical structuring of specifications. Besides other features, it permits overloading of operators, as in some programming languages. The PVS specification language allows embedding of special logics. In particular, property specifications can be expressed in temporal logics specialized for specific hardware/software/real-time applications.

### *4.2 PVS Verification Features*

The PVS verifier is used to determine if the desired properties hold in the specification of the model. The user interacts with the verifier by a small set of commands. The verifier contains procedures for boolean reasoning, arithmetic and (conditional) rewriting. In particular, binary decision diagram (BDD) based state-space exploration may be invoked for model-checking and Boolean reasoning. It also features a variety of general induction schemes to tackle large-scale verification. Moreover, different verification schemes can be combined into general-purpose strategies for similar classes of problems, such as verification of microprocessors [6,7].

A PVS specification is first parsed and type-checked. At this stage, the type of every term in the specification is unambiguously known. The verification is done in the following style: we start with the property to be checked and repeatedly apply rules on the property. Every such rule application is meant to obtain another property that is simpler to check. The property holds if such a series of applications of rules eventually leads to a property that is already known to hold.

More information on PVS and its use in mechanical verification can be obtained through World Wide Web: <http://www.csl.sri.com/sri-csl-fm.html>.

## 5. Automatic Verification of a Core Module in PVS

This phase of the project starts with translating the SpecCharts descriptions or the Verilog/VHDL implementation descriptions to the PVS language. Various properties related to the behavior of the protocols are then specified in PVS and mechanically verified using the capabilities outlined in Section 4.2. Using this method we were able to verify modules in the implementation of the Fiber Channel node loop Core.

Figure 2 shows the PVS description of the loop port state machine (LPSM) translated from a Verilog RTL implementation description. The specification for the LPSM can be found in the FC-AL standards document [3]. A closer look reveals that the PVS description is very similar to the original Verilog description in so far as the state type, variables and the description of the state transition relations are concerned. The PVS description starts with the definition of the state type which is an enumerated type consisting of all the states in the LPSM. This is followed by a declaration of all the input (output) variables and the state variable for the machine. The function "*next*" is a mapping from input variables and current state to the next state. In effect, this function describes all the state transition relations for the LPSM. The mapping is expressed using an IF-THEN-ELSIF-ELSE statement similar to a Verilog or VHDL description.

The remaining portion of the PVS description is concerned with the verification of specific properties of the LPSM. First of all, the next state relation defined by function "*next*" is now expressed as a predicate "*N*" over the current and next state records "*s1*" and "*s2*" respectively. The state record "*staterec*" is a record type similar to the struct type in the C programming language. It is used to package objects of different types within a single type. We can then treat an object of such a type as one single object externally, but with an internal structure corresponding to the various fields in the record. The properties that the specification should satisfy are then expressed as *LEMMAS* or *THEOREMS* in the PVS specification. For example the following prop-

```
al_state : THEORY

BEGIN
IMPORTING ctlops                                    % CTL Theory
state: TYPE =    { LPST_MON,                        % State type for LPSM
                   LPST_ARB,
                   LPST_OPEN,
                   LPST_OPENED,
                   LPST_XMIT_CLS,
                   LPST_RCVD_CLS,
                   LPST_XFER,
                   LPST_INIT,
                   LPST_OPNI_RCVD,
                   LPST_OPNI_IDLE,
                   LPST_OPNI_TSM,
                   LPST_OPNI_ALPA,
                   LPST_OLD_PORT}
i_idle, i_arbx, . . . . . . , i_clrl   : VAR bool   % Input variables
s : VAR state                                       % State variable
% State transitions for Loop Port State Machine
next(s, i_idle, i_arbx, i_arbx_f0, i_opnx, i_cls, . . . . . . , i_match, i_clrl ) : state =
(IF (i_clrl) THEN LPST_MON
ELSIF s = LPST_MON THEN
       (IF   (i_req_init = TRUE) THEN LPST_INIT
        ELSIF (i_lip_3 = TRUE) THEN LPST_OPNI_RCVD
        ELSIF ((i_opnx = TRUE) AND (i_match = TRUE) AND
             (i_allow_opn = TRUE)) THEN LPST_OPENED
        ELSIF ((i_req_arbx = TRUE) AND (i_winacc = TRUE))
            THEN LPST_ARB
       ELSE  LPST_MON
        ENDIF)

. . . . . . .

ELSIF s = LPST_OLD_PORT THEN
       (IF   (i_req_init = TRUE) THEN LPST_INIT
        ELSIF (i_lip_3 = TRUE) THEN LPST_OPNI_RCVD
        ELSE  LPST_OLD_PORT
        ENDIF)
ELSE LPST_MON
ENDIF)

staterec: TYPE = [# sf:state, i_idle, i_arbx, i_arbx_f0, i_opnx, i_cls, . . . . . . , i_match, i_clrl: bool #]

s1,s2: VAR staterec
N(s1,s2):bool =                                     % Next state predicate
   ( sf(s2) = next(sf(s1),i_idle(s1), i_arbx(s1), i_arbx_f0(s1), i_opnx(s1), i_cls(s1), . . . . . , i_match(s1), i_clrl(s1)))

init(s1):bool = LPST_MON?(sf(s1))                   % Initially in MONITORING state

% Properties
prop1: LEMMA                                        %From initial state, the "OPEN" state is reachable
init(s1) IMPLIES EF(N, LAMBDA s1: LPST_OPEN?(sf(s1)))(s1)

prop2: LEMMA                                        % Given initial state, "XMITED_CLOSE" state is reachable
init(s1) IMPLIES EF(N, LAMBDA s1: LPST_XMIT_CLS?(sf(s1)))(s1)

init1(s1):bool = init(s1) AND NOT i_req_init(s1) AND NOT i_lip_3(s1)
              AND i_req_arbx(s1) AND i_winacc(s1)

prop3: LEMMA                                        % Given intial state, and conditions on input fibre "ARB" state is reachable
init1(s1) IMPLIES EF(N, LAMBDA s1: LPST_ARB?(sf(s1)))(s1)

. . . . . .

END al_state
```

Figure 2. PVS Specification for the Loop Port State Machine

erty:

```
prop1: LEMMA
init(s1) IMPLIES EF(N, LAMBDA s1: LPST_OPEN?(sf(s1)))(s1)
```

states that from initial state, LPST_MON, the state LPST_OPEN is reachable. This is specified using the CTL operator "EF". For more information on CTL refer to [10,11].

These properties or *theorems* are automatically verified in PVS using a set of decision procedures. The proof of property prop1 listed in Figure 1 is quite straightforward and automatic as show in Figure 3.

```
prop1:
{1}  (FORALL (s1 :   staterec) :
       init(s1) IMPLIES EF(N, LAMBDA s1: LPST_OPEN?(sf(s1)))(s1))

By rewriting and mu-simplifying
This completes the proof of prop1.
Q.E.D.
```

This proof took less than 40 seconds on a Sparc30 with 32 MB main memory. Proofs for other properties that we verified were similar to the one shown above.

The results of formal verification are *stronger* than ones obtained through inexhaustive simulation/testing because they apply to *all* combinations of input and output values for the particular case. This is in contrast to a few combinations of inputs and outputs chosen by the validator during inexhaustive testing.

## 6. Conclusions and Future Work

In this paper we have presented a method for creating system specifications for simulation and formal verification of complex designs. We have applied these techniques to the design of a Fibre Channel Core and have shown that formal verification can be an effective supplement to traditional simulation based verification techniques. We believe that this approach can increase the overall coverage and confidence in the correctness of the specification and implementation in a "right-first-time" design methodology. It is particularly suitable for cases where test vectors or stimuli are not available or are difficult to create.

As part of the future work we plan to model and *formally verify* a complete implementation of our fibre channel protocol core. We also plan to extend and apply these techniques to other domains including multimedia and digital video applications.

**Acknowledgments**

We thank Dr. Srinivasa Malladi for providing us the implementation models and helping us understand the Fibre Channel protocol, its properties and applications.

## 7. References

[1] "Fibre Channel - Physical and Signalling Interface (FC-PH)". ANSI X3T9.3 Working Document, Revision 4.1, August 12, 1993.

[2] "Fibre Channel - Implementation Guide (FC-IG)". ANSI X3T9.3 Working Document, Revision 4.2.1, April 16, 1994.

[3] "Fibre Channel - Arbitrated Loop (FC-AL)". ANSI X3T9.3 Working Document, Revision 4.2, March 11, 1994.

[4] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[5] *The PVS Proof Checker: A Reference Manual (Beta Release)*. N. Shankar, S. Owre, and J. M. Rushby. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[6] D. Cyrluk. "Microprocessor Verification in PVS: A methodology and simple example". Technical Report, SRI International, December 1993. Report CSL-93-12.

[7] Ramayya Kumar and Thomas Kropf, editors. *Preliminary Proceedings of the Second Conference on Theorem Provers in Circuit Design*, Bad Herrenalb (Blackforest), Germany, September 1994. Forschungszentrum Informatik an der Universität Karlsruhe, FZI Publication 4/94.

[8] P. Sreeranga Rajan. "Transformations in High Level Synthesis: Specification and Verification". Technical Report NL-TN 118/94, Philips Research Laboratories, Eindhoven, The Netherlands, April 1994.

[9] Patrick Lincoln, Sam Owre, John Rushby, N. Shankar, and Friedrich von Henke. "Eight Papers on Formal Verification". Technical Report SRI-CSL-93-4, Computer Science Laboratory, SRI International, Menlo Park, CA, May 1993.

[10] J. Burch, E. M. Clarke, K. McMillan, D. Dill, and J. Hwang. "Symbolic model checking: $10^{20}$ states and beyond". In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, June 1990.

[11] S. Rajan, N. Shankar, and M. Srivas. "An integration of model-checking with automated proof-checking". *Submitted for publication*, January 1995.