Demand Access Protocol Design and Validation with SPIN

John S. Seguí
Communications Networks Group
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
John.S.Segui@jpl.nasa.gov

Abstract—In order for distributed systems to communicate reliably engineers standardize on communication rules (or protocols). Unforeseen behavior in communication protocols can push faults up to applications resulting in uncontrollable systems and should not be tolerated. However, while most modern protocols undergo extensive testing, rigorous formal methods, such as model checking, are rarely used due to complexity and massive incomputable state spaces. Nevertheless, starting formal validation early in the development cycle carries less-complexity, significantly smaller state spaces and allows for full validation. As a side benefit early error detection and correction costs significantly less than redesigns later in a system's life-cycle.

This paper discusses the preliminary design and validation of the Demand Access Protocol (DAP) using the SPIN model checker. Using English language specifications and flow charts, the author developed a PROMELA (the language used by SPIN) specification and tested basic safety properties. Used correctly, a model checker can thoroughly validate complex systems and guarantee absence of fault conditions.

TABLE OF CONTENTS
1. INTRODUCTION ......................................................1
2. SPECIFICATION ......................................................2
3. VALIDATION ..........................................................4
4. DISCUSSION ...........................................................6
5. ACKNOWLEDGMENTS ............................................6
REFERENCES .............................................................7
BIOGRAPHY ...............................................................7
APPENDIX A - PROMELA SPECIFICATIONS ..........7

1. INTRODUCTION

In general, computer protocols are specified using time charts, state-transition diagrams/tables, or natural (spoken) languages. However, diagrams do not allow for direct validation but require human inspection to identify possible failure conditions, which while often effective is neither formal nor guaranteed. Moreover, the inherent lack of scalability of charts and diagrams make them ill-suited for reasonably complex protocols. As a consequence, most modern computer protocols are specified in quasi-formal subsets of natural languages with diagrams for clarification.

One should note that implementation issues caused by ambiguous protocol specifications can result in corrupted communication, loss of communication or in some cases injury to users. So, with a desire for correctness, formal (non-ambiguous) specification languages have been developed; however, looking at the Internet Engineering Task Force's (IETF) Request For Comments (RFCs) regarding communication protocols, one realizes that formal languages are rarely used. While RFCs are not direct specifications they are used as the basis for protocol design and facilitate interoperability.

This paper discusses the application of a formal language, PROMELA, for specification and validation of the Demand Access Protocol with the SPIN model checker. PROMELA specifications involve non-deterministic execution and send/receive channels. Most importantly, PROMELA allows

![Figure 1 – Demand Access Architecture from [1]](image-url)

1 1-4244-1488-1/08/$25.00 ©2008 IEEE.
2 IEEAC paper #1333 Version Final, Updated January 15, 2008
for specification of complicated safety requirements (internally called “never” claims) which when validated using SPIN can guarantee the absence of undesired conditions. For more complex properties, users can convert safety requirements from Linear Temporal Logic (LTL) to PROMELA “never” claims.

PROMELA models are similar to non-deterministic extended finite state machines (EFSMs) and tools allow for direct translation of Specification and Description Language (SDL) EFSMs to PROMELA [4] processes. Furthermore, other authors show that formal specification can be used for performance analysis as described in [5], but is beyond the scope of this paper.

Section 2 details design and PROMELA specification with examples from Demand Access Protocol. Section 3 introduces the validation tool, SPIN, and describes the validation process.

2. SPECIFICATION

Demand Access Protocol Overview

The Demand Access Protocol architecture defines an application, a resource manager and a communication system. In between these systems the Demand Access Protocol handles message formatting. Figure 1 is from [1] and shows the Demand Access architecture. The specification of translation layers, communication systems, and resource managers is beyond the scope of the Demand Access Protocol specification. Thus, DAP is primarily a messaging system.

The Demand Access Protocol (DAP) was originally described in natural language reports and flow charts [1]. From these descriptions and face-to-face meetings with the original developers, the author developed a PROMELA specification. Figure 2 is an extract [1] of the DAP Provider flowchart.

Since, DAP is a fairly simple protocol this work was completed in short time and had little difficulty modeling the high-level protocol behavior. However, implementers of DAP can leverage the PROMELA specifications to create initial code structure and, by adding implementation-specific information to the specification, continually validate correct operations. Additionally, multiple implementations can create low-level (implementation-specific) detailed PROMELA modes and validated interoperability.

DAP Natural Language Specification

DAP users and providers are similar to clients and servers, respectively, in software architectures. During a session DAP users send requests and acknowledgments, while DAP
providers send replies, acknowledgments, and commands to handle resource allocations per session. Moreover, the DAP providers interact with a Resource Manager through a translation layer. DAP does not specify the workings of the translation-layer.

Additionally, each message can alternate between one-way and two-way transmission modes. Two-way transmissions require acknowledgments and response timers and are useful in environments where the communication system does not fully guarantee delivery. Also, each session uses a priority tag, but DAP does not directly specify a priority system (that will depend on the exact system requirements and so is implementation specific). So, the QoS system validation should be considered when implementing each system, but is not discussed in this paper.

Appendix 1 of [1] provides a flowchart style description of the DAP user and provider and a high-level overview of protocol operations. No other software specification was provided to the author. Starting validation at an early stage allows the quickest use of automated validation by not bringing the burden of a complete specification with added complexity from application environments, such as: memory managers or process schedulers.

On the other hand, if a complete specification and even implementation was provided, additional environment models could have been developed which would increase validation time but provide system specific validation more interesting to application developers. This paper only considered high-level validation. So, while one must still perform environment validation, starting with a simpler model reduces the chance of bugs being added and, more importantly, reduces the work required to abstract away details from a sometimes highly complex implementation.

**PROMELA Specification Overview**

PROMELA uses non-deterministic execution and send/receive channels to specify protocol behavior. Consider each node (ellipse) in figure 3 as a set of states with arrows being possible transitions to another set of states. In essence, each PROMELA operation (e.g., send, receive, or variable manipulation) creates a possible transition. So, figure 3 shows all possible transitions of the Provider model without respect to data. However, depending on local variables transition execution may not always be possible.

The number of states within each set depends directly on the number of process variables (and size). So, to calculate the total possible model states one needs knowledge of the internal data. For instance, assuming the Provider model contained one (and only one) variable, a bit, the number of states directly doubles from the number seen in figure 3 (as there are 2 possible values for the bit). So the total number of possible states is directly a product of the number of transitions and the number of possible values for all variables contained in the model. Since the number of possible states can go rapidly as variables are added, the burden rests on the user to limit variables to reasonable sizes without losing fidelity.

Moreover, the number of system states is the multiple of previously mentioned states for each model. Say we have one instance of model X and one instance of model Y, then the total system states is the product of the number of states for model X and the number of states in model Y. As one can surmise, the number of possible states grows quickly, resulting in state spaces too large to fit into storage or with too many states to check in any reasonable amount of time.

However, the one saving factor is that while the number of possible system states grows rapidly, the number of possible reachable states from a specific start state grows at a lower rate, and most importantly, with good abstraction, can be limited to a number allowing for full validation in a reasonable time period.

Demand Access is functionally simple and so modern computers are capable of performing full validation with little difficulty. Also, since, at the time of this paper to the author, little was known about other interacting systems (i.e., the environment), this work only validated basic safety properties of DAP such as the absence of illegal end states (deadlocks) or infinite runs (live-locks).

DAP was structurally simple enough that full validation was possible. Using only the description given in [1], the author created a PROMELA specification with limited interaction.
between the protocol designers. Additionally, the communication channel was easily modeled because Demand Access requires the communication system to guarantee delivery of uncorrupted messages. Text boxes 1 and 2 show the first PROMELA specification of the DAP user and provider, respectively. For this initial validation only basic liveliness properties were checked. No never claims were used.

**PROMELA Overview**

PROMELA is a straightforward language. Text boxes 1 and 2 show sample specifications. All commands are either executable or block until possible. All lines beginning with double colons (::) are unordered guarded command sequences. Semi-colons separate command strings (but are not required to terminate the command string).

PROMELA “if” control statements are different than the C programming language counterpart and somewhat similar to “switch/case” control statements, particularly because of the ability to have multiple execution paths. However, unlike, “switch/case” statements PROMELA “if” statements can have multiple executable paths and do not specify an order of preference (i.e., non-deterministic) whereas, in C, case checks are sequential. Furthermore, PROMELA contains only one type of loop statement: the “do” loop. Other common loops can be created by combining “do” loops with additional “break” commands. “If” and “do” blocks end with “fi” and “od,” respectively.

Command sequences preceded by “proctype” and surrounded by curly-brackets specify individual processes. Processes can either start active (preceded with the “active” keyword) or wait for invocation by another process.

Processes interact through channels. Processes can read from (using “?”) or write to (using “!”) channels. For example, the command “achan?var” reads from channel “achan” and stores the value in the local variable “var.” If the channel is empty the command is not executable and the process blocks until capable of reading a value from the channel. Additionally, processes can check channel status by surrounding variable names with square-brackets (e.g., “achan?[var]”). Writing to (“!”) a channel functions identically to reading and checking channel status before writing is also possible.

Advanced users can use “syntactic sugar” to expedite validation. For example, surrounding command sequences with the “atomic” keyword and curly-brackets notifies the SPIN compiler that the commands should be treated as a single state and executed in one step. For atomic sequences the user must guarantee that all commands are executable if the first (the guard) is executable. Most models do not require atomic sequences, but “syntactic sugar” shortens validation time and can prove crucial for large complex systems.

**Automata Theory**

For those so inclined, in terms of automata theory, the PROMELA specification and defined safety properties are similar to finite state automata (as shown in Figure 3). These automata are specified by four sets: automaton states, transitions, valid start states, and valid end states. During validation, SPIN computes the asynchronous product, A, of all protocol automata. Afterwards SPIN computes the synchronous product of the negation of the safety automaton and A. Consequently, in terms of automata theory, searching for violations equates to finding an ω-run [2]. The core verification algorithms are discussed in [6] and [2].

### 3. Validation

The SPIN model checker works by compiling a PROMELA specification into a binary validation program, which makes for compact, quick and efficient runs. Additionally, SPIN has various options allowing for tradeoffs in memory or computational requirements depending on the specification's complexity and validation hardware resources.

Once a protocol is specified in PROMELA and the safety

```plaintext
do ::userToChan!daRequest
::if
  ::atomic { chanToUser?[daAck] -> chanToUser?daAck }
::else -> if
  ::atomic { chanToUser?[daReply] -> chanToUser?daReply } -> userToChan!daAck
  ::else -> if
    :: atomic { chanToUser?[daCommand] -> chanToUser?daCommand } -> userToChan!daAck
    ::else
      fi
    fi
fi
od
```

Text 1: Demand Access - User (v1)
requirements are defined, one can use the Simple PROMELA INterpreter, SPIN, to compile the model into a self-validating binary. The SPIN tool allows for various simulation and validation methods. For instance, if enough memory is available users can run an exhaustive validation and guarantee absence of the defined error conditions. While full validation is desired, for systems too complex to fit in memory one can run simulation which can guarantee full state-space traversal at the expense of requiring an undetermined finite amount of time. Users can run simulations until enough time has elapsed that a high percentage of states were traversed.

After creating a PROMELA model, one uses the “spin” command to generate a binary validation program. The description of this process and list of command line options can be found in [2]. For this test the author used “spin -a demandAccess.pml” where demandAccess.pml is the file containing the PROMELA specifications.

Text box 3 shows the output from validation. The first line warns that validating this specification through SPIN resulted in an “invalid end state” (deadlock). However, upon further inspection one realizes that this deadlock was not directly due to any faults of the Demand Access model, but was actually a result of the buffer model. Particularly the “userToChan!daRequest” write operation was blocking since the “userToChan” message channel filled up and did not allow overflow. In this case, the buffer model did not discard messages, and so, the provider's send buffer would fill when sending acknowledgments while the user's buffer would fill sending requests and neither would be able to process the other's message until the local channel allows message transmission. Since user requests can be generated at any time an excessive amount can overload the user to provider channel while simultaneously the provider overloads the provider to user channel with acknowledgments and the system deadlocks. This failure case is commonly known as circular blocking.

However, given the nature of demand access (and other user-driven protocols) this case is unavoidable. So, correct implementations should either use application layer flow control to limit the number of outstanding requests or simply allow the communication system to discard messages.

The second version of the DAP model implemented flow control by allowing providers to ignore application requests when the outbound channel is full. Adding this allowed full validation without error. On the other hand, spin can compile with the “-m” option to specify for channels to allow overflow and discard messages. Both were tested and allowed error free operation.

The rest of the lines in Text box 3 are standard validation reporting. While full validation was not completed, 53,032
states were traversed and 47 megabytes of memory was used. Note that 40 megabytes was pre-allocated for the Depth-First-Search stack (maximum depth of 1,000,000). With the corrections made full validation required similar time and memory.

An imperfect communication channel was also tested by adding a separate thief process with a continuous loop that checks for messages and removes them at will. Given the nature of distributed systems this “theft” can occur at any (and all) possible time. So a full validation will inject this into all possible orders.

Additionally, for quick testing, one can use the built-in SPIN simulation function. Sample output is shown in Figure 4. Running a random simulation allows developers to get a visual “feel” of the model, but does not assure full coverage and so is not directly used for validation.

However, the simulation function is useful when errors are discovered through exhaustive validation. Upon reaching an error case, SPIN outputs a trace file which when run through the simulator shows the exact steps taken to reach the error state. Note that the trace is only one possible error trail (out of an unknown number) and not necessarily the shortest. For instance, in the previous example the error trail was 34,256 steps long, whereas the error case can be quickly reached by having the user loop on “userToClientRequest” until the channels are full.

4. DISCUSSION

Usually, Protocol validation is an afterthought and requires abstracting implemented systems; however, in this case the author worked with the creator of the DAP and wrote the protocol specification while validating it along the way. However, certain details can still be abstracted away to narrow the state space. For instance, while the DAP message format has several fields useful for implementation, the model is not concerned with routing and multiple-access; this model assumes messages will only arrive at the appropriate destination, if at all (the model does allow for loss of messages).

Validating communication protocols during the design phase limits the amount of ambiguities in specifications. For this project, the author started by specifying the Demand Access Protocol (DAP) in PROMELA. However, this method caused difficulty later since no tools existed to generate diagrams useful for collaboration or for visual inspection directly from PROMELA (with the exception of SPIN simulation output and PROMELA-specific transition diagrams).

On the other hand, starting with a protocol specified in the Specification and Description Language (SDL) one can semi-automatically generate a PROMELA specification [4]. As the Demand Access Protocol PROMELA specification gains environment models, the author will check more extensive properties.

5. ACKNOWLEDGMENTS

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

The author would like to thank Dr. Gerard Holzmann, Dr. Jay Gao and Dee Leang for their time and exceedingly helpful discussions.
REFERENCES


BIOGRAPHY

John S. Seguí works as a systems engineer in the Communications Networks Group of the Jet Propulsion Laboratory, managed for NASA by the California Institute of Technology, Pasadena, California. He currently works on network simulation and protocol analysis for the JPL's Interplanetary Network Directorate (IND) and supports NASA's SCAN Integration Project (SCIP).

APPENDIX A - PROMELA SPECIFICATIONS

```c
#define bufferSize 1
#define CHANNEL_THIEF

chan chanToProvider = [bufferSize] of {mtype};
chan providerToManager = [bufferSize] of {bit};
chan managerToProvider = [bufferSize] of {bit};

active proctype channelUserToProv()
{
  mtype m;
  end: do
  :: atomic { nfull(chanToProvider)
              && nempty(userToChan) ->
              userToChan?m; chanToProvider!m }
  od
}

active proctype channelProvToUser()
{
  mtype m;
  end: do
  :: atomic { nfull(chanToUser)
              && nempty(providerToChan) ->
              providerToChan?m; chanToUser!m }
  od
}

#ifdef CHANNEL_THIEF
active proctype userToProvThief()
{
  end: do
  :: atomic { nempty(userToChan) -> userToChan?_ }
  od
}
active proctype provToUserThief()
{
  end: do
  :: atomic { nempty(providerToChan) ->
              providerToChan?_ }
  od
#endif

active proctype user()
{
  end: do
  :: atomic { nfull(userToChan) ->
              userToChan!daRequest }
  ::(1) -> if
  :: atomic { chanToUser?[daAck] ->
              chanToUser?daAck }
  ::else -> if
  :: atomic { chanToUser?[daReply]
              && nfull(userToChan) ->
              chanToUser?daReply; }
  ::else -> if
  :: atomic { chanToUser?[daCommand]
              && nfull(userToChan) ->chanToUser?daCommand; }
  ::else
  fi
  fi
```
active proctype provider()
{
    end: do
        :: atomic {nfull(providerToChan) ->
            providerToChan!daCommand }

        :: (1) -> if
            :: atomic { chanToProvider?[daAck] ->
                chanToProvider?daAck }
            :: else -> if
                :: atomic { chanToProvider?[daRequest] && nfull(providerToChan) && nfull(providerToManager) ->
                    chanToProvider?daRequest;
                    providerToChan!
                    daAck; providerToManager!1 }
                :: else -> if
                    :: atomic { managerToProvider?[1] && nfull(providerToChan) ->
                        managerToProvider!?1;
                        providerToChan!daReply}
                    :: else fi
                fi
            fi
        od

    active proctype manager()
    {
        end: do
            :: atomic {providerToManager?[1] && nfull(managerToProvider) ->
                providerToManager!?1;
                managerToProvider!1 }
            od
    }