Specification and Validation of Communications in Client/Server Models

F. Joe Lin

Bellcore, 445 South Street, NJ 07960, U.S.A.
E-Mail: fjlin@bellcore.com

Abstract

Errors such as deadlock and race conditions are very common yet extremely difficult to debug in the communications design of client/server models based on remote procedure calls and multi-threading. This paper presents an effective approach to detect these errors. It shows how to apply the specification and validation techniques in Protocol Engineering to discover those errors in the early stages of a client/server software development. The work is based on the protocol specification and validation tool PROMELA/SPIN. PROMELA is extended to a new language called PROMELA-C/S for additional expressive power of specifying client/server communications. A PROMELA-C/S translator then is built to convert PROMELA-C/S to PROMELA for validation using SPIN. The paper also reports the results of some specification and validation trials using PROMELA-C/S, its translator, and SPIN.

1. Introduction

Client/server models have gained wide acceptance for implementing distributed systems in recent years. The debugging of their implementations, however, is known to be extremely difficult. This paper proposes a strategy to tackle an essential part of this problem. It focuses on the detection and removal of the communications errors in client/server models in the design stage. The thrust is to detect as many errors as possible in the most error-prone area of a client/server system (i.e. its communications), and to do this in the early stages of a development.

Most client/server models use RPCs (Remote Procedure Calls) as their inter-process communications mechanism. All the processes in the model are structured as a group of servers that offer services to a group of clients by means of RPCs. This is very different from *cooperating processes* models where all processes are viewed as equal peers and they communicate with each other either by message passing conforming to a protocol or by data sharing based on global variables.

RPC can be regarded as a simple request/reply protocol mimicking a procedure call. RPC makes inter-process communications in a client/server model conceptually simple. Nevertheless, it pays a price of limiting the degree of parallelism that can be achieved between processes, i.e., between clients and servers. This is because a client (or a server) is normally blocked (or occupied) when invoking (or serving) an RPC.

In order to achieve maximum parallelism while retaining the simplicity of RPCs, servers and clients are often designed as multi-threaded processes. This allows an RPC be invoked (or served) by a client (or server) in a dedicated thread while other threads within the process are still active (or available).

Nevertheless, new concerns that didn't exist in a single-threaded process now arise. One of these is the *intra*-process communication among multiple threads in a client or server, which is done through shared variables. Critical sections thus have to be carefully designed to prevent the threads from trying to update the same data structure at the same time.

These observations reveal that the major complexity of client/server communications is caused by:

1. Multi-threading of the process to do RPCs for maximum parallelism
2. Synchronization among multiple threads thus created.

Consequently, to correctly design a distributed application based on client/server communications is not an easy task. Design errors such as deadlock and race conditions are as prevalent as in protocol design[1]; they just migrate from inter-process communications to intra-process communications.

Furthermore, debugging at the code level could be very difficult because of the following problems:

> "the interference of the debugger with the code which can make it impossible to detect the effects of race conditions, the presence of indeterminisms (which hinder the reproducibility of subsequent debugging sessions), the vast parallel events to be perceived by

0-8186-6685-4/94 $4.00

the user, and the lack of support for notions like 'distributed breakpoint' and 'distributed single-step'."[2]

This paper applies the specification and validation techniques of Protocol Engineering to tackle these problems. The work is based on the protocol specification and verification tool PROMELA/SPIN[3]. PROMELA is extended to a new language called PROMELA-C/S to give it additional expressive power for specifying client/server communications design. A PROMELA-C/S translator is then built to convert PROMELA-C/S to PROMELA for running validation using SPIN.

The rest of the paper is organized as follows. Section 2 explains how to model client/server communications in cooperating processes such that the validation techniques developed for the latter can be readily applicable to the former. Section 3 defines PROMELA-C/S for specifying client/server communications. Section 4 explains the PROMELA-C/S translator. The results of some specification and validation trials using PROMELA-C/S, its translator, and SPIN are briefed in Section 5. This is followed by concluding remarks in Section 6.

2. Modeling Client/Server Communications in Cooperating Processes

This section shows how client/server communications using RPCs and multi-threading can be modeled by cooperating processes communications using both message passing and shared variables. The goal is to map a client/server model to a cooperating processes model while preserving the client/server's communication semantics. This would make it possible to convert the validation problem of client/server communications into that of cooperating processes communications; any design errors found in the latter can be tracked back to those made in the former, and the correctness of the former can be inferred from that of the latter. As there are many well-developed validation tools for the cooperating processes model in Protocol Engineering[4], the validation for the client/server model will be readily applicable.

The communications paradigm of client/server models using RPCs is depicted in Fig. 1. Each process in this system is either a client or a server, or both. RPC is the communication mechanism between processes. The processes use dedicated threads for invoking or serving RPCs to achieve maximum parallelism. The threads within a process communicate and synchronize with one another via shared variables. There are no shared variables between processes; a process can only talk to another process using RPC.

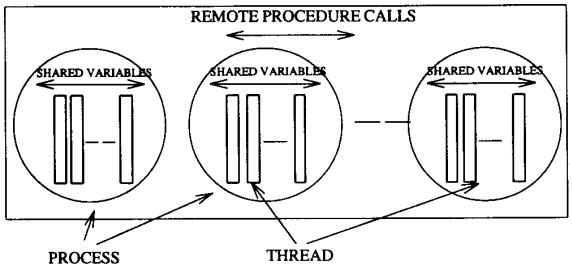

Figure 1. Client/Server Communications Using RPCs and Multi-Threading

The communications paradigm of cooperating processes models is depicted in Fig. 2. Each process in this system only has a single thread of control. All processes are viewed as equal peers. They communicate through message passing and/or shared variables. Message passing can be synchronous or asynchronous. This paper assumes the use of asynchronous message passing.

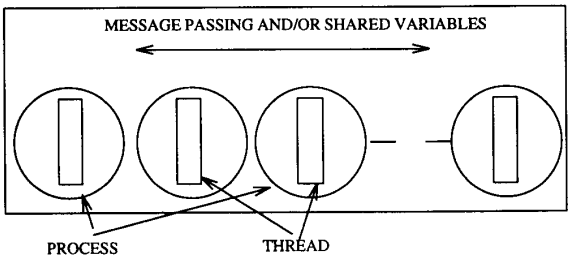

Figure 2. Cooperating Processes Communications Using Message Passing and/or Shared Variables

A client/server model can be mapped to a cooperating processes model based on the following ideas (depicted in Fig. 3).

1. RPC can be modeled by asynchronous message passing that conforms to a simple request/reply protocol. It works as follows: the client's procedure call is modeled by sending a request message to the server, then waiting to receive a reply message from the server. The server's execution of the procedure can be modeled by continuously listening for a request message to come in from a client; when such a message arrives, it executes the procedure, sends the reply message back to the client, then goes back to listening. Finally, the client receives the reply

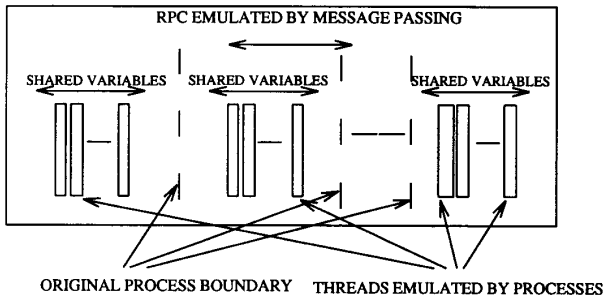

Figure 3. Mapping from Client/Server to Cooperating Processes

message that it is waiting for and resumes its operations.

2. Communicating threads can be modeled by communicating processes that abide by the following two rules (see Fig. 3):
   - Rule 1. The only communications among threads, now emulated by processes, are shared variables; no message passing is allowed.
   - Rule 2. There are no shared variables across the original process boundary; only message passing that emulates an RPC is allowed.

Such a mapping is based on the assumption that no thread scheduling is done within a process. When there are more than one threads ready in a process, one of them will be nondeterministically selected to run. This assumption is harmless to error detection since the validation will simply try all the possible scheduling policies and priorities during its exhaustive state exploration.

3. PROMELA-C/S Language

This section describes PROMELA-C/S using examples. For a complete definition of the language, readers are referred to the PROMELA-C/S language report[5].

3.1 Communications Characteristics of Client/Server Models

Several characteristics of client-server communications were not explicitly discussed in Sec. 2. This section will address these issues to prepare readers for the introduction of PROMELA-C/S. The design of PROMELA-C/S is largely influenced by the client/server models underlying the OSF™ (Open Software Foundation) DCE (Distributed Computing Environment)[6] [7].

DCE is an emerging industry standard for developing distributed applications in heterogeneous environments. It is a collection of integrated software components that sit on top of networked computers' operating systems. DCE's RPC and thread service are the foundation of all DCE software components and DCE applications.

3.1.1 RPC

Two important characteristics of RPCs that are not yet addressed are:

1. How a client identifies a server which provides the procedure requested by the client.
2. How a server identifies the client (more precisely, the thread within the client) which is waiting to receive the reply.

In DCE, the first problem is solved by the DCE *directory service*, through which a server advertises itself and a client locates a server it needs. The second problem is solved by DCE automatically when a client issues an RPC to request a server's service.

PROMELA-C/S has to deal with these problems as well, though by different mechanisms. PROMELA-C/S solves the first problem by its language translator, i.e., by how PROMELA-C/S gets mapped to PROMELA. PROMELA-C/S solves the second problem by requiring a specifier-defined thread ID in every RPC call (further details in Sec. 3.3).

Furthermore, advanced RPCs like the ones in DCE support the following capabilities:

1. The client has the option to maintain its context between RPC calls. This means that the client can request the server to maintain its state information after an RPC call. Then through subsequent RPC calls, the client can come back to the same server for more services.
2. The server has the option to allocate concurrent threads to serve several calls to the same procedure for maximum parallelism. This means that each time a call to this procedure arrives, a fresh thread would be spawned by the server to handle it.

DCE supports both capabilities with little programming effort from a DCE application programmer. For the first capability, the programmer can request a *context-handle* be returned from a server to the client in an RPC call. In the subsequent RPC calls, this context handle then can be used by the client to continue its operations with the same server. For the second capability, the DCE programmer only has to declare this property when programming the server, DCE will automatically do the creation of a concurrent thread for each incoming call.

PROMELA-C/S supports the specifications of both capabilities at the same level of abstraction as that of DCE (details in Sec. 3.3). Its translator automatically implements both capabilities in PROMELA; little effort is required from the specifier.

3.1.2 Thread Service

The thread service allows multi-threading of servers and clients and is an indispensable part of client/server communications. DCE's thread service is based on the pthreads interface specified by POSIX in their 1003.4a standard[8]. It provides three facilities for inter-thread communications within a process.

1. *Mutex.* A mutex is used to synchronize multiple threads' accesses to a given resource, such as a shared variable. A mutex ensures that only one thread can access the shared resource at a time. There are two primitives on mutexes: lock and unlock. They are used in pairs to form a critical section as shown below.

```
lock(mutex);
...critical section ...
unlock(mutex);
```

2. *Condition Variable.* A condition variable provides further flexibility in manipulating a critical section. It allows a thread to "take a break" (i.e. be blocked) within a critical section. There are two primitives on condition variables: cond_wait and cond_signal. The 'cond_wait' allows a thread to release the lock and wait for a 'condition' within a critical section. Upon the signaling of the condition, the thread then regains the lock and resumes its operations in the critical section (see the example below).

```
lock(mutex);
...
cond_wait(cond_var, mutex);
...
unlock(mutex);
```

   The 'cond_sig', on the other hand, is for a thread to signal another thread upon a condition.

```
lock(mutex);
...
cond_sig(cond_var);
...
unlock(mutex);
```

3. *Join Facility.* 'Join' is a primitive that allows a thread to wait until the completion of another thread before continuing its execution.

PROMELA-C/S is defined to support the declarations of both condition variables and mutexes and all the primitives described above. Its translator automatically implements these mechanisms in PROMELA.

DCE's thread service supports dynamic creation and deletion of mutexes and condition variables during the run time, which makes little sense in a validation. It also extends the semantics of cond_sig to include the broadcast of a signal to all waiting threads and supports other variations of mutexes and mutex operations such as recursive mutexes and trylock operations. These extensions are not supported by PROMELA-C/S for the moment.

3.2 Brief Description of PROMELA

This section briefly describes the main language features of PROMELA to prepare readers for Sec. 3.3. PROMELA is a protocol specification language that is designed to model and specify the cooperating processes communications as depicted in Fig. 2. All processes in PROMELA are specified in a flat structure. Processes are defined using process-type (*proctype*) declarations that specify the process name, parameters, and process body. Processes can communicate with one another by either data sharing or message passing.

Data sharing among the processes is achieved by declaring global variables outside the scope of the processes. *Channels* are a special class of variables that define the communication channels between processes. Message passing between processes can be specified in terms of channels using two primitives: Send (denoted by '!') and Receive (denoted by '?'). For example, 'a!123' denotes sending a message '123' to channel 'a'; 'a?var' denotes receiving a message from channel 'a' and storing it in variable 'var'. Channels are declared with the name, buffer size, and message type. For example,

chan a = [5] of {byte}

specifies a channel 'a' with buffer size '5' and message type 'byte'.

Processes in PROMELA do not execute themselves; a *run* statement is required to explicitly execute a process. The exception is the *init* process which will be automatically run and must be included in every PROMELA specification. Once a process is run, it can execute the 'run' statement to execute more processes (even itself recursively). Since a process specification can be run (or instantiated) many times, each 'run' will return a unique process ID to distinguish one process instance from another.

The process body consists of local variable declarations and PROMELA statements. Each PROMELA statement can be regarded as an expression: if it evaluates to true (i.e. nonzero), it is executable; otherwise, the process will be blocked until the statement becomes true. Some statements are always executable; the assignment statement is an example among others.

'Send' and 'Receive' statements are evaluated as follows: 'Send' is true when the channel is not full; 'Receive' is true when the channel is not empty. For example, the following is a legitimate fragment of PROMELA specification.

```
chan?i;
i < 0;
i = i + 5;
i + 5;
i = i + 10;
```

Note that how the above specification is executed depends on what value, if ever, 'i' is to receive at statement 'chan?i'. For instance, if i receives -10, i will be -5 when the process is blocked at statement 'i + 5'. PROMELA provides a semicolon-equivalent notation '->' to make the specification easier to read. Using '->' the above specification can be rewritten with more clarity.

```
chan?i   -> (i < 0 -> i = i + 5);
i + 5 > 0 -> i = i + 10;
```

The statements above are *guarded statements*. They always start with a condition (a guard) followed by one or more statements.

The statements available in PROMELA are expression, assignment, *if*, *do*, *goto*, and some others. The 'if' statement allows selection from several guarded statements. If multiple guarded statements are executable, one of them will be nondeterministically selected and executed; if none is true, the process will be blocked until one becomes true. The 'do' statement is similar to the 'if' statement except that it loops until a *break* statement is encountered. For example, the following 'do' statement computes n factorial (n!).

```
i = n; result = 1;
do
:: i == 0 -> break
:: i > 0  -> result = result * i; i = i - 1
od;
```

Readers who are interested in more details of PROMELA are referred to [3].

3.3 PROMELA-C/S Description

This section presents the usage of PROMELA-C/S in specifying client/server communications.

3.3.1 Specification of RPC in PROMELA-C/S

An example of PROMELA-C/S specification using RPC is shown in Fig. 4. This is a simple client/server model without multi-threading; there is only one client, one server, and one RPC defined. The client calls the procedure 'factorial' to compute the factorial of a sequence of numbers. Note that all the PROMELA-C/S statements are printed in italics with their keywords in boldface (the rest are in PROMELA).

```
/* A simple client/server specification */
#include "std.vh"
#define NSERVERS_PLUS_1 2
#define NTHREADS 1
#define NUM 5
/* Declare characteristics of a remote procedure */
rpc factorial(NSERVERS_PLUS_1,NTHREADS,para(byte),
                        para(byte));
proctype client(byte caller_id; byte rand)
{
  byte facto; byte i;
  i = rand;
  do
  :: i <= 0 -> break
  :: i >  0 -> /* remote procedure call */
    call factorial(NO_CONTEXT, _, caller_id, para(i),
                        para(facto));
    printf("%d factorial is equal to %d.\n",i,facto);
    i = i - 1
  od
}
proctype server()
{
  byte caller_id;
  byte ori_rand;
  /* begin model remote procedure */
  begin_listen
  :: proc factorial(NO_CONCUR,NO_CONTEXT,caller_id,
                                ori_rand)->
    byte rand; byte facto;
    rand = ori_rand;
    facto = 1;
    do
    :: rand == 1 -> break
    :: rand != 1 -> facto = facto * rand;
          rand = rand - 1
    od;
    return(caller_id,_,facto)
  end_listen
  /* end model remote procedure */
}
init {     run client(0,NUM);
           run server()
}
```

Figure 4. A Simple RPC Example

3.3.1.1 Declare RPCs Before specifying the client/server communications in PROMELA-C/S, the specifier must first determine the size of the validation model. This includes the following:

1. Identify how many remote procedures are required in the application.
2. Define each procedure's calling convention.
3. Find out how many servers are required in the model.

4. Find out how many threads in the model will be making RPC calls.

The first two items specify information similar to that required by a DCE RPC's *interface definition*. The last two items are required to create a closed, finite validation model. This information is declared by the PROMELA-C/S remote-procedure-call (*rpc*) declaration. For each remote procedure in the model one such declaration is required. For example, the RPC in Fig. 4 is declared as follows.

[Example 1] **rpc** *factorial(NSERVERS_PLUS_1, NTHREADS,***para***(byte),***para***(byte));*

In general, the 'rpc' declaration specifies the procedure name of an RPC followed by a list of four RPC properties in parentheses.

— The first property specifies the number of servers in the model plus 1.
— The second property specifies the number of threads in the model that make RPC calls
— The third and fourth properties specify the order and the data types of RPC's *in* and *out* parameters, respectively.

Note that the first and the second properties are common among all RPCs in a model. Consequently, they always appear as constant identifiers in the 'rpc' declarations.

Example 1 above declares an RPC with procedure name 'factorial'. This RPC operates in a model where there is only one (i.e. NTHREADS) thread and one (i.e. NSERVERS_PLUS_1 - 1) server. It has one 'in' parameter and one 'out' parameter, both of type 'byte'.

3.3.1.2 Specify RPC calls in the client A client can make an RPC call using the *call* statement without any prior knowledge of which server will execute the procedure . For example, the call to procedure 'factorial' in Fig. 4 is shown below.

[Example 2] **call** *factorial(NO_CONTEXT, _, caller_id,* **para***(i),* **para***(facto));*

In general, the 'call' statement specifies a procedure name, followed by a list of five parameters in parentheses.

— The first parameter is an *in-context*. The 'in-context' is used by a client to reach the server which maintains the client's context between RPC calls. Before it can be used, the client must first identify a server through an RPC call. This is accomplished through the second parameter.
— The second parameter is an *out-context*. The 'out-context' is used to get a server's ID such that a client can use it to reach the same server in subsequent RPCs.

The 'in-context' and 'out-context' parameters together emulate the context-handle in DCE (discussed in Sec.3.1.1).

— The third parameter is the ID of the thread that makes the RPC call. This ID is chosen and supplied by the specifier.
— The fourth and fifth parameters are the 'in' parameters and 'out' parameters of the RPC procedure, respectively. The data types of these parameters should match with those specified in the 'rpc' declaration.

Example 2 above shows a 'NO_CONTEXT' as the 'in-context' and an underscore ('_') as the 'out-context'. The 'NO_CONTEXT' is a PROMELA-C/S language-defined constant. It declares that the client doesn't discriminate one server from another; any server that provides the wanted RPC procedure is acceptable. If this is not the case, a server's ID will be specified here, instead. The underscore ('_') is a PROMELA-C/S language-defined variable, used as an indication of "don't-care". In this case, it means that the client has no interest in knowing the identity of the server.

To maintain a client's call context between RPC calls, a typical sequence of RPC calls would look like the following.

call *procedure_name1(NO_CONTEXT, handle, thread_id, ...);*
.....
call *procedure_name2(handle, _, thread_id, ...);*

Note that the out-context ('handle') returned from the first RPC is used as the in-context of the second RPC.

3.3.1.3 Specify RPC procedures in the server All the RPC procedures of a server are defined in the server's 'listening loop' which is enclosed between the keywords 'begin_listen' and 'end_listen'. For example, the 'factorial' procedure in Fig. 4 is specified as follows.

[Example 3]

```
begin_listen
::proc factorial(NO_CONCUR,NO_CONTEXT,
                 caller_id,ori_rand) ->
        byte rand;
        byte facto;
        rand = ori_rand;
        facto = 1;
        do
        :: rand == 1 -> break
        :: rand != 1 -> facto = facto * rand;
                        rand = rand - 1
        od;
        return(caller_id,_,facto)
end_listen
```

Note that a procedure is defined in three components: an entry ('proc') declaration, a procedure body, and a

'return' statement. The 'proc' statement defines the entry of a remote procedure. In general, it is a keyword 'proc' followed by the name of the procedure, then by a list of parameters in parentheses.

— The first parameter specifies whether a concurrent thread will be spawned for each procedure call. The specifier has two choices: CONCUR or NO_CONCUR (these are PROMELA-C/S language-defined keywords.) If a procedure is specified as a CONCURrent procedure, a fresh thread will be spawned each time a new call arrives. Otherwise, the procedure will be executed in the server's main thread.
— The second parameter matches with the 'in-context' parameter of the client's procedure call. If 'NO_CONTEXT' is specified here, it indicates that the server doesn't discriminate between clients; it will accept the procedure call from any client. On the other hand, if the server's own ID is specified here, the server will only accept a procedure call whose 'in-context' parameter matches with its ID.
— The third parameter matches with the third parameter of the procedure call, which is a thread's ID. This parameter informs the server where to return the results.
— The remaining parameters are the dummy 'in' parameters that match with the actual 'in' parameters of the procedure call in the client.

Example 3 above defines a procedure 'factorial' with no concurrency and with no client discrimination. The procedure has only one dummy 'in' parameter, 'ori_rand'. The third parameter 'caller_id' is used to receive the calling thread's ID.

The 'return' statement specifies what parameters should be returned from a procedure to the client (this is different from the 'return' of a typed function in a programming language.) In general, it is a keyword 'return' followed by a list of parameters in parentheses.

— The first parameter is the calling thread's ID received at the entry of the procedure.
— The second parameter is the server's own ID. This parameter will match with the 'out-context' parameter of the procedure call.
— The remaining parameters are the dummy 'out' parameters that match with the actual 'out' parameters of the procedure call.

Example 3 above shows that the procedure 'factorial' will return the value of 'facto' to a thread in the client whose ID is 'caller_id'. Moreover, this will be done without identifying the server itself.

Note that both variables 'rand' and 'facto' are the local variables of procedure 'factorial'. They are not accessible from elsewhere in the server.

3.3.1.4 Include standard header file "std.vh" Every PROMELA-C/S specification has to include a standard validation header file "std.vh" (Fig. 4). This file provides the data and constant declarations required by the PROMELA specification to be translated from PROMELA-C/S. Pre-defined constants and variables such as NO_CONTEXT and '_' ("don't-care") are all declared in this header file.

3.3.2 Specification of Thread Service in PROMELA-C/S

An example of PROMELA-C/S specification using the thread service is shown in Fig. 5. There is only one process in this specification, but two threads are spawned from the main thread of the process. The communications and synchronization among the two spawned threads implement a solution to the well-known producer-consumer problem. One thread is a producer, whereas the other is a consumer. The producer keeps producing numbers and storing them in a buffer of size 'N'. The consumer continues taking numbers from the buffer and consuming them. Because the producer and the consumer are run concurrently, synchronization between them is required to maintain correct communications through the buffer.

3.3.2.1 Execution of Threads Specifying the execution of a thread in PROMELA-C/S is different from that of a process.

1. A thread requires a specifier-defined thread ID as its mandatory first parameter.
2. The execution of a thread doesn't return any value.

For example, in the specification of Fig. 5, the producer and the consumer are executed with specifier-supplied thread IDs '0' and '1', respectively, as follows.

exec *producer(0);*
exec *consumer(1)*

3.3.2.2 Declarations of mutexes and condition variables PROMELA-C/S supports two new data types: *cvar* and *mutex*. The former is used to declare condition variables; the latter to declare mutexes. Both declarations follow the conventions of PROMELA's variable declarations. For example, the specification in Fig. 5 declares the following condition variables and mutexes.

mutex *crit_mutex;*
cvar *empty_cond, full_cond;*

As in PROMELA, both condition variables and mutexes can be arrays.

3.3.2.3 Synchronization Primitives The formats and semantics of the primitives for mutexes and conditional variables simply follow what was discussed in Sec. 3.1.2. For example, the specification in Fig. 5 has the following primitives.

lock(*crit_mutex);*
cond_wait(*empty_cond,crit_mutex)*
cond_sig(*full_cond)*
unlock(*crit_mutex);*

The primitive 'join' requires the specifier to specify both the name and the ID of the thread. For example, if the consumer thread in Fig. 5 needs to wait until the completion of the producer thread, the specification would be as follows.

join(*producer, 0)*

3.3.2.4 Inter-Thread Shared Variables PROMELA-C/S requires the specifier to provide the keyword *shared* to the declarations of the variables that are to be shared among threads. For example, the specification in Fig. 5 declares the following inter-thread shared variables.

shared *byte buffer[N];*
shared *byte counter = 0;*

4. PROMELA-C/S Translator

A PROMELA-C/S translator has been implemented to convert a PROMELA-C/S specification to a PROMELA specification. This automates the mapping from client/server models to cooperating processes models and makes the validation techniques in Protocol Engineering readily applicable to a new area.

The translator consists of two phases. Phase one converts a PROMELA-C/S specification to a PROMELA specification with M4 macros. Phase two expands those macros and generates the final PROMELA specification. The conversion from the client/server model to the cooperating processes model is done in phase one. This includes the flattening of the two-level process-thread hierarchy into a one-level process structure, the creation of support declarations, variables, and labels, and the creation of the macros for phase two. Phase one also checks the syntactic correctness of those language extensions defined for PROMELA-C/S and produces an error report if necessary. It is implemented in 800 lines of AWK and K-Shell. Phase two utilizes the M4 macro processor to do macro expansion. Twenty macros have been defined for this purpose.

5. Specification and Validation Trials

This section presents the results of some specification and validation trials using PROMELA-C/S, its translator,

```
/* The Producer-Consumer Problem */
#include "std.vh"
#define N 10
proctype pcp()
{
  mutex crit_mutex;
  cvar empty_cond, full_cond;
  shared byte buffer[N];
  shared byte counter = 0;
  thretype producer(byte id)
  {
        byte i = 0, pointer = 0;
        do
        :: TRUE ->
          /* produce an item "i" */
          lock(crit_mutex);
          do
          :: counter < N  -> break
          :: counter == N -> cond_wait(empty_cond,crit_mutex)
          od;
          buffer[pointer] = i;
          pointer = (pointer + 1) % N;
          counter = counter + 1;
          if
          :: counter == 1 -> cond_sig(full_cond)
          :: counter != 1
          fi;
          unlock(crit_mutex);
          i = ( i + 1) % (2*N)
        od
        }
  thretype consumer(byte id)
  {
        byte i, pointer = 0;
        do
        :: TRUE ->
          lock(crit_mutex);
          do
          :: counter > 0  -> break
          :: counter == 0 -> cond_wait(full_cond,crit_mutex)
          od;
          i = buffer[pointer]; /* consume an item */
          pointer =(pointer + 1) % N;
          counter = counter - 1;
          if
          :: counter == N-1 -> cond_sig(empty_cond)
          :: counter != N-1
          fi;
          unlock(crit_mutex)
        od
  }
  exec producer(0);
  exec consumer(1)
}
init { run pcp() }
```

Figure 5. A Simple Multi-Threading Example

and SPIN. The purposes of the trials are (1) to get hands-on experience in using PROMELA-C/S in modeling client/server communications, (2) to test the PROMELA-C/S translator, (3) to get a feel of how effective this approach is, and (4) to understand the limitation of the approach.

Ten examples across a variety of problems thus were specified using PROMELA-C/S and validated using both the PROMELA-C/S translator and SPIN. Numerous errors were found by validation throughout the development of these specifications. All of them, if not caused by the bugs in the PROMELA-C/S translator, always mapped back to some design errors in the original client/server specifications. Most errors were caught by SPIN as "invalid endstates". They reflect the deadlock and race conditions among processes and threads in the original model, often caused by either incomplete or incorrect design. Readers who are interested in the detailed data of these trials can request a longer version of this paper from the author.

6. Concluding Remarks

This paper shows how a protocol specification language PROMELA can be extended to a new language PROMELA-C/S for specifying and validating client/server models. It also demonstrates how such a specification of the client/server model can be mapped to that of the cooperating processes model to take advantage of the existing, well-developed protocol validation tools such as SPIN.

The proposed approach can be applied to other protocol specification and validation tools as well. Different tools, however, will have different strategies for mapping from client/server communications to cooperating processes communications.

Further applications and extensions of the approach are also possible. Two of them are worth mentioning here. First, it is very likely that both client/server and cooperating processes communications will co-exist in some distributed applications as quoted from [9]:

> "The client/server model is simple and powerful, but cooperative processing is more suited for expressing parallelism. A combination of the two may allow a more flexible way of designing and programming distributed applications..."

Such a mixed style of communications makes the design of a distributed application even more complex. PROMELA-C/S, as an extension of PROMELA, however, can easily accommodate the specification of this mixed style, and support its validation through SPIN.

Second, thanks to SPIN's model checking capability[10], verification techniques based on temporal logic assertions are readily applicable to the PROMELA-C/S specifications as well. These assertions can be written at the level of PROMELA-C/S using temporal operators such as *always*, *implies eventually*, and *implies ... until*. A tool can be built to automatically map those assertions to the "never-claims"[3] at the level of PROMELA. This will link the model-checking capability of SPIN to PROMELA-C/S.

Acknowledgement

The author thanks Gerard Holzmann at AT&T Bell Labs for many opportunities of discussions regarding PROMELA and SPIN. He also thanks Peter Bates and Rong Chang for sharing their insights about DCE's thread service, R. C. Sekar for pointing out an inadequacy in PROMELA-C/S's initial modeling of DCE's RPC, and Bijan Arbab for providing information on one of the test examples. Thanks also to Ritu Chadha, Glen Diener, Gary Herman, and Yow-Jian Lin for providing comments to improve this paper.

References

1. G. Holzmann. Protocol Design: Redefining the State of Art. *IEEE Software Magazine*, pp.17-22, January 1992.
2. M. Bever, K. Keihs, L. Heuser, M. Mühlhäuser, and A. Schill. Distributed Systems, OSF DCE, and Beyond. In *DCE - The OSF Distributed Computing Environment: Client/Server Model and Beyond*, pp.1-20, Spring-Verlag, October 1993.
3. G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, N.J., 1991.
4. F. J. Lin and M. T. Liu. Protocol Validation for Large-Scale Applications. *IEEE Software Magazine*, pp.23-26, January 1992.
5. F. J. Lin. PROMELA-C/S language report. In *Specification and Validation of Communications in Client/Server Models*, Appendix A, Bellcore Technical Memorandum, TM-23977, March 1994.
6. *Introduction to OSF™ DCE*. Open Software Foundation, 1992.
7. *OSF DCE Application Development Guide: Revision 1.0*. Open Software Foundation, 1993.
8. IEEE Portable Operating System Interface for Computer Environments Committee. *Threads Extension for Portable Operating Systems (Draft 6)*. IEEE, February 1992. P1003.4a/D6.
9. Y.-H. Wei and C.-L. Wu. Integrating RPC and Message Passing for Distributed Programming. In *DCE - The OSF Distributed Computing Environment: Client/Server Model and Beyond*, pp.192-206, Spring-Verlag, October, 1993.
10. P. Godefroid and G. Holzmann. On the Verification of Temporal Properties. *Protocol Specification, Testing, and Verification, XIII*, pp.109-124, North-Holland, 1993.