Implementation and validation methods applied to SSCOP

Julian Harris and Jan Madsen
Laboratoire de TeleInformatique
EPFL
CH-1015 Lausanne Switzerland
email: harris@di.epfl.ch madsen@di.epfl.ch

Keywords: Synchronous C++, Randomwalk, Validation, ATM, Signalling.

Abstract

This paper outlines the methods used to realise the implementation of the Service Specific Connection Oriented Protocol (SSCOP) which forms a main component in the Signalling ATM Adaptation Layer (SAAL). We give a short introduction to Synchronous C++, a concurrent extension to C++, followed by a protocol programming method based on these extensions. We show how this environment is made complete, by introducing the way in which random walk simulations can be performed upon the implementation code, therefore avoiding translation from model to implementation. The specific example of SSCOP and its environment is then described including proposals to simplify and streamline the appropriate protocol specifications. This more specifically addresses the interactions between SSCF and SSCOP and the integration of these two layers. The general programming method applied to this shows its suitability for implementing these ideas . A random walk analysis peformed upon the resulting code then gives indications of the correctness of the implementation.

1 Introduction

There are many languages and systems developed for protocol specification and programming. The existance of a single modular integrated design environment which encompasses all aspects of the design, validation and implementation can greatly ease the problems usually encountered. The environment based upon the Synchronous C++ language gives us a methodology with which design, implementation and random walk validation can be performed, resulting in an object oriented, readable and understandable code. In this paper we show each of these aspects and the suitability of Sync.C++ in these areas, and in particular its use in implementation considerations applied to ATM signalling.

2 Synchronous C++

Synchronous C++ is an extension to C++ developed at the Computer networking laboratory at the EPFL [1]. It merges the concepts of objects and processes. Sync.C++ defines the concept of 'active' objects, that are considered as separate executable entities able to run in pseudo parallelism with other active objects. The code of an active object is stored in a body declared as a private method of the class describing the object. This method has the same name as the class preceded by the symbol '@'. When an active object is instantiated, the execution of the body follows that of the constructor. It is the execution of this body that gives each object it's own 'life' and therefore the term 'active'. Execution of the body can be suspended by means of the 'accept' statement, followed by the name of a public method owned by that object. This suspends execution until the specified method has been called and executed by a calling object. At this point, both the caller and the called object resume execution. This can be seen below in Figure 1. If on the other hand the object being called is not ready to accept the call, the caller is blocked until the called method is accepted and then executed by the calling object. This type of blocking causes the synchronisation of the two objects involved, and is termed 'rendezvous' communication.

Sync.C++ can also handle multiple accepts and method calls in parallel by means of the select statement. This is in fact one of the main advantages Sync.C++ has over other concurrent languages such as Concurrent C++ [9] and ADA which do not allow calls and accepts in the same select statement. When the select statement is entered, the first executable event reached in the order in which the list is written, is taken for execution. In Figure 2 below, this would mean either method_a is called from outside or method_b is accepted in object_x. Once completed, the actions described after the event are executed and

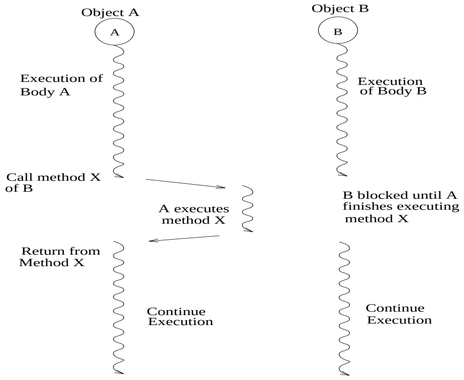

Figure 1: Inter-object communication

the select exited. If no event in the list is executable, a default action can be added. Without a default, the select would simply be exited and no actions taken.

```
select{
   accept method_a ;   /*Local method accepted
   actions;
||
   object_x.method_b;   /*External method call
   actions;
||
   default;
}
```

Figure 2: Select statement

2.1 General architecture

By considering the concept of a system of communicating finite state machines in our understanding of layered protocol architectures [8], we can make important analogies on which to base Sync.C++ implementations. Firstly we can make the analogy between the concept of a finite state machine being a seperate entity, and the concept of an active object. We see that each state machine and therefore each protocol layer in a system can be realised with a seperate active object. Secondly we observe that communication between these entities must provide both a synchronisation point in their execution and also some kind of procedure specifying the actions corresponding to such a communication. We can make the analogy between this observation and the concept of the synchronous method call. The method call provides the synchronisation point required and instructions following the 'accept' in a receiving active object provide a corresponding procedure. We can now describe a general programming method for the communication between protocol layers and also the corresponding behaviour.

These primitive calls form the input events to finite state machines which describe the behaviour of the protocol layers. In each state of the protocol state machine, different groups of input events are waited for in parallel. We can perform this using the Sync.C++ 'select' and the corresponding list of method accepts. To manage the different selects corresponding to each state, we put the selects into a control structure (C 'switch') inside the body of the active object in the way described below.

The C 'switch' statement is contained in an infinite loop (for) and is dependant upon the current protocol state. The code of each state is defined in each 'case' of the switch and takes the form of a select. The actions appearing after the synchronisation events in the selects are responses to the input events and consist of appropriate output events i.e. external method calls, variable updates, and a modification of the state variable. Once the select has been exited, the switch statement is 'broken' ('C' break instruction) and re-executed with the new value of the state variable. Timeouts can also be accommodated by using the 'waituntil' statement as a synchronisation choice and appropriate actions following it. The code for this structure is shown below in Figure 3.

```
SSCOP::@SSCOP(){
enum state{state1,state2.....staten}
for(;;){
  switch state{
    case(state1):                    /*state1 code
      select{
          accept messagetype1;      /*receive message
          out.send(PDU1);
          variable1++;
          state = state2;
      ||
          accept messagetype2;      /*receive message
          out.send(PDU2);
          state = state1;
      ||
          waituntil (timeout_time);      /*timeout
          timeout_actions;
      }
    break;
    case(state2):                    /*state2 code
      select{.....}
    break;
  }                                  /*end of for loop
}                                    /*end of switch
}
```

Figure 3: General architecture

3 Random Walk Validation

To provide some indication as to the correctness of a program's behaviour, it is necessary to perform some type of validation. This means that some or all the system states of a program are explored and checked for error. A system state is the combination of the states of all variables, queues and finite state machine states at some point in time. A full state space analysis requires that every one of these states is explored and analyzed for errors such as deadlock and unspecified reception. A deadlock occurs when there is no possible evolution of the execution from the present state and an unspecified reception is the lack of ability to handle an input event when in a state. In order to perform this type of validation, the system states visited need to be stored in a database in order to avoid exploring a state more than once and also to indicate that all states have been visited and exploration should be terminated. For a complex system or protocol, the large number of system states present can be impossible to store which is a common problem termed 'state space explosion'. Methods to avoid this by enhancing storage mechanisms are available, although with large programs even these can fail resulting in reduced validation performance.

Random walk analysis [7] provides us with an attractive solution. With this technique, execution sequences in a program are made as a function of a random number. This causes the evolution of the execution to follow a random path with the system states visited being tested for errors. With such a system the states already visited need not be stored therefore avoiding the state space explosion problem. However this means also that there is no natural termination of the analysis as it is impossible to determine when all system states have been visited. Similarly, complete absence of errors cannot be proved if guarantees of state space coverage cannot be given. These disadvantages are outweighed by the ability of random walk to perform efficient analysis upon very complex systems and also the simplicity of the system compared to that of a full state space exploration.

3.1 Synchronous C++ random walker

During normal Sync.C++ execution, each time a synchronisation choice is presented to the kernel within a select statement, the kernel immediately selects the first executable choice, according to the priority mentioned in Section 2. With modifications to this kernel, all possible executable synchronisation choices in the application at a particular time can be presented and a choice made as a function of a random number. This is performed each time the application has no more instructions to execute which are not synchronisations. This causes the program to execute in a random fashion. It is because this can be achieved by only modifying the kernel that a random walk validation can be performed on any Sync.C++ application. It is worth mentioning that the random walk kernel does not execute the application in real time, but much faster. This is because any synchronisation choice, a method call, method accept or waituntil is executed immediately. Time is therefore not respected. If, at some point, no synchronisation choices are enabled, the analysis is stopped and a deadlock reported. A record of the synchronisations taken to arrive at the deadlock can be made for diagnostic purposes.

Statistical weighting can be applied to the execution choices within an application by applying constraints on the likelyhood of choosing a particular synchronisation within the kernel. For example, we could cause an active object representing a data channel to loose messages at some controlled rate by applying weighting to statements which either transmitted or discarded a message. This can be used to 'steer' a random walk analysis in an attempt to explore sequences thought to relate to more realistic situations. As the random walk has been performed on the actual Sync.C++ code, the analysed application can be executed directly with the normal execution kernel. This avoids having to make an implementation of a validated model, which itself is susceptible to the introduction of errors. As well as this advantage, the environment can be quite easily used without specialist knowledge of validation techniques.

4 ATM Signalling

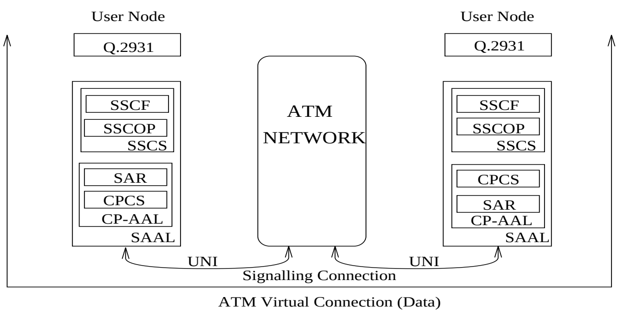

Figure 4: ATM Signalling

Figure 4 shows the ATM signalling architecture. We are interested in the signalling performed at the User Network Interface (UNI) by the protocols shown at the user nodes. Here we can see protocols classified in different groups. The uppermost layer is the Q.2931[3] and is responsible for exchanging signalling data intended for setting up ATM data connections. The Q.2931 also performs the task of initiating setup and release of signalling connections provided by the protocols residing below it. This group is termed the Signalling ATM Adaptation Layer (SAAL) which is itself split into two groups. These are the Service Specific Convergeance Sublayer (SSCS) and the Common Part AAL protocols (CP-AAL). The common part protocols perform unassured transfer of signalling protocol data units. The part of SAAL in which we are interested in implementing, the SSCS, is again divided into two parts, the Service Specific Coordination Function (SSCF) and the Service Specific Connection Oriented Protocol (SSCOP)[4].

SSCOP contains mechanisms for establishing and releasing signalling connections, and also provides reliable message transfer by means of a sliding window protocol. We restrict our study to the setting up and release of signalling connections, and do not consider data transfer over this channel. SSCOP consists of ten main states. Two of these are the 'Idle' and 'Data Transfer Ready' states signifying presence and absence of a signalling connection respectively. The other eight states are pending states and wait for responses from either the peer SSCOP entity, or the local SSCF layer above. The peer SSCOP entities communicate via SSCOP PDUs. Exchange of these messages form the handshaking protocol that establishes, releases and resynchronises the connection.

SSCF is defined as being a coordination function mapping the general functions provided by SSCOP to the possible different layers that may reside above it. This is done by specifying different versions of SSCF depending upon this above layer. It holds information regarding the state of SSCOP and translates the messages to and from the layer above accordingly. In the case of the UNI it translates messages to and from Q.2931 therefore SSCF at UNI [5] is used.

5 Implementation considerations

A study into the messages exchanged between SSCOP, SSCF and Q.2931 resulted in a proposal to simplify the state machines involved and therefore the implementation. An example of the existing signalling connection establishment is shown in Figure 5 . When the BGN resulting from the initial Establish request, arrives at the peer SSCOP, an AA-Establish.indication is sent to the SSCF above, shown at point (1) in the diagram. Here, the SSCOP moves into its 'Incoming Connection Pending' state and awaits an AA-Establish.response, point (2), from SSCF. This response is independant of external actions so is always given. This causes SSCOP to leave it's incoming pending state and move to Data Transfer Ready. At this point, SSCOP sends a BGAK PDU to acknowledge the initial establish request. At the same time as sending the response to SSCOP, SSCF sends an AAL-Establish.indication to Q.2931 to signal the connection request.

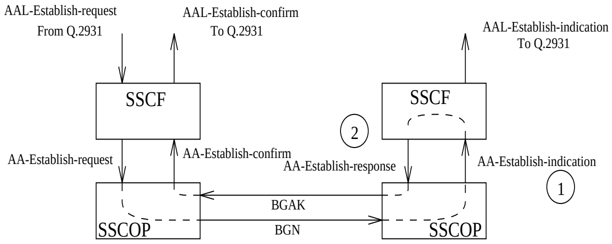

Figure 5: Connection Establishment

The proposal is to move this response mechanism into SSCOP and return the BGAK in the example as soon as the BGN is received. The AA-Establish.indication that was originally sent to SSCF can be renamed AAL-Establish.indication, and sent directly from SSCOP to Q.2931 bypassing SSCF completely. This new sequence can be seen in Figure 6. We see also that the left hand side SSCOP directly outputs an AAL-Establish.confirm to Q.2931, again bypassing SSCF. In fact the functionality of SSCF has been integrated into SSCOP and not really been removed as such.

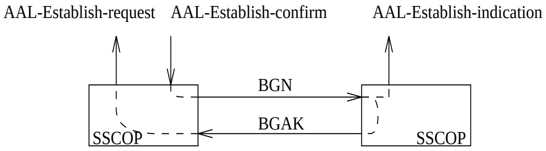

Figure 6: Bypass of SSCF

The integration of the SSCF into the SSCOP has further implications on the SSCOP internal states. As SSCOP now no longer has to remain in a state waiting for the AA-Establish.response from the SSCF, that pending state (Incoming connection pending) can be eliminated from SSCOP. SSCOP can now move directly from 'Idle' to 'Data Transfer Ready', whilst outputing the AAL-Establish. indication and also the BGAK. The original and new state diagrams for this example can be seen below in Figure 7.

This same simplification can be performed on all the incoming pending states in SSCOP. The same mechanism takes place in all incoming pending states, with SSCF always replying to SSCOP with an unconditional response message. The remaining name translation functions of the SSCF can be integrated into

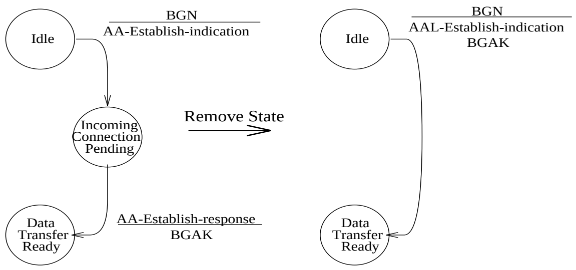

Figure 7: State Modification

SSCOP by directly translating AA signals to appropriate AAL primitives. This leaves us with an SSCOP layer now with only six states. We now only have to deal with programming SSCOP and the appropriate part of Q.2931. This is done by refining the general method outlined in Section 2.1 to our example.

5.1 Creation of model

By taking the general system architecture outlined in Section 2.1, the system can be constructed in Sync.C++ with ease. As shown below in Figure 8, there will be six active objects, each endsystem having two active objects, and the communications medium being described by two active objects, one for each direction of message flow. The uppermost layers shown (Q.2931), will act as stimuli for the system by having the ability to spontaneously call primitives of the layer below (SSCOP). This is needed to perform the random walk analysis.

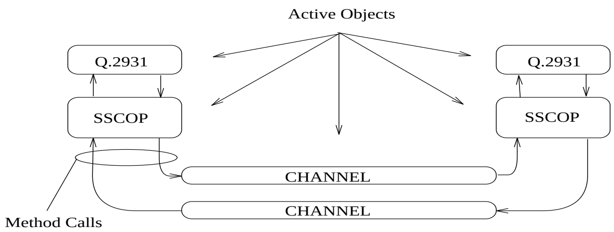

Figure 8: Active Objects

At this point, the general architecture is refined to generate code describing the specific protocol layers. The functioning of the state machines to be described in each body is derived from the ITU specifications of the appropriate layers. These are written in various ways including finite state machine diagrams, finite state machine tables and SDL [6]. This translation is straightforward with the particular events, output actions and variable updates being 'filled in' in the skeleton code architecture shown in Section 2.1. The following figure shows a small fragment of the code describing the modified SSCOP.

```
1   for(;;){
2    switch(state){
3      case idle:
4         select{
5            accept aal_est_request;
6            VTCC = 1;
7            VTSQ = (VTSQ+1)%MAX_VTSQ;
8            out.Send(BGN_PDU_MESSAGE);
9            timer_cc = now()+100;
10           state= out_connection_pend;
11        ||
12           accept  Receive;
13           switch(msg->GetMessageType()) ;
14           case BGREJ_PDU:
                .
                .
15        }
16        break
17   rest of code
```

Figure 9: SSCOP Code fragment

We see an almost identical construct to the general code shown earlier. Line 3 shows the start of the code for the state 'Idle' with line 4 showing the start of the list of possible events in that state (select statement). Lines 5 and 12 show two of these events with the lines beneath them describing actions to be taken. Line 8 shows a resulting PDU being sent with line 11 updating the state variable. Lines 13 to 15 show identification of the PDU accepted at line 12.

The channels represent all physical and non physical layers which reside below SSCOP and between the two end users. These use Fifo queues which are themselves based upon linked list structures. Sending messages to the channel takes the form of a parameterised method pertaining to the specific PDUs (BGN, BGAK etc) and sending messages from the channel to the above layer is a method call to SSCOP, sending and then removing the message from the front of the queue. Each channel transmits data in a different direction, with each SSCOP reading from one channel and sending on the other. These particular calls can also be put into parallel using the 'select' statement with a mechanism modelling the loss of messages to approximate a more real environment.

6 Random walk tests

Our random walker runs underneath a simple to use interface, utilising buttons and text windows. The random walker is first launched, after which the actual application is launched from the same XTerm. If required, a choice can be made on the random execution pattern by selecting the seed variable for the random number generator, after which a start command is issued. If a deadlock is found in the application, i.e. there is an invalid end to the execution, the random walker stops and reports the deadlock situation. The list of all the executed synchronisations mentioned in Section 3.1 can be saved to a file, which enables the user to perform the diagnostic analysis upon the execution sequence which caused the deadlock. From this list, the last synchronisation successfully performed can be found, and therefore the state the system was in when the deadlock occurred. Corrective modifications can then be made to the application and retests made.

6.1 Results of random walking

Weighting of synchronisations was performed to include a controlled rate of message loss in the channels implemented, to model a more real environment. The code outlined above with SSCF integrated into SSCOP was passed to the random walker environment and executed. Various executions were performed each with a different random number generator seed variable. The applications continued to execute without deadlock. The number of synchronisation events was counted at approximately 40,000. The experiments each lasted approximately half an hour to reach this number of transitions, although many more transitions can be made by letting the tests run indefinitely. This provides us with many more transitions as we could observe in a real time system giving us a good indication as to the validity of the implementation and enhanced confidence of it's error free operation.

7 Conclusions

We have shown here considerations in the implementation of the service specific parts of the SAAL, describing methods to simplify the interaction between the SAAL protocols. We have shown a specific example of the simplification of a sequence of messages passed between SSCOP and SSCF, which has allowed us to integrate the two. This simplification results in a much reduced final state machine, giving a very clear view of the operation and interactions of the SSCOP with it's environment.

We have shown also the implementation methodology of the resulting SSCOP based on a general method using Syncronous C++. This allows code to be easily generated which gives a very readable executable version of the protocol behaviour which can be analysed directly with the random walk validation tool developed. The fact that the validation techniques can be applied to this same code is felt to be on of the main advantages of the environment. Although existing protocols were used in this study, it is intended that Synchronous C++ be used for the whole of the life cycle of protocol software, including the initial design, bringing Synchronous C++ into the realm of specification languages also. An environment such as this can greatly reduce the labours of protocol programming and as outlined in the paper needs no expert knowledge of validation systems.

References

[1] Abel Divin. Un environment de developpement d'applications paralleles base sur le concept d'objet. Phd Thesis November 1995. Laboratoire de TeleInformatique, EPFL, Switzerland.

[2] C.Petitpierre. Realisation de programmes concurrents. 1994. Laboratoire de Teleinformatique, EPFL, Switzerland.

[3] ITU Document Q.2931 BISDN Connection Control Protocol.

[4] ITU Document Q.2110 BISDN ATM Adaptation Layer - Service Specific Connection Oriented Protocol (SSCOP).

[5] ITU document Q.2130 BISDN Signalling ATM Adaptation layer, Service Specific Coordination Function (SSCF).

[6] Ferenc Belina and Dieter Hogrefe and Amardeo Sarma, SDL with Applications from Protocol Specification. Prentice-Hall, 1991.

[7] West C.H., Protocol Verification by Random State Exploration. Proceedings of the IFIP WG 6.1 Sixth International Workshop on Protocol Specification, Testing, and Verification.1986.

[8] Gerard J. Holzmann, Protocol Design Specification and Verification. Prentice Hall, 1991.

[9] N.H.Gehani and W.D.Roome. Concurrent C++, Software practice and Experience, vol 18, 1988 p1157-1178.