1

PROTOCOL DESIGN IN SMURPH

Wlodek Dobosiewicz and Pawel Gburzynski

Department of Computing Science,
University of Alberta, Edmonton, Alberta, Canada T6G 2H1

ABSTRACT

We present SMURPH—a software package for modeling communication protocols at the medium access control (MAC) level. SMURPH can be viewed as a combination of a protocol specification language and an event-driven, discrete-time simulator that provides a virtual (but realistic) and controlled environment for protocol execution. This combination yields a protocol prototyping system which can be used for designing low-level communication protocols and investigating their quantitative (performance) and qualitative (correctness) properties. The essential features of SMURPH are illustrated with an example of the well-known alternating-bit protocol.

1 INTRODUCTION

SMURPH[1] is an object-oriented programming environment based on C++ for specifying communication protocols and modeling communication networks. By a communication network we understand a configuration of *stations* interconnected via *channels*, running a collection of concurrent communicating processes. The distributed algorithm realized by those processes is called the communication protocol.

Unlike other protocol specification systems, e.g., ESTELLE [1, 2], LOTOS [3, 4], or PROMELA [5], SMURPH has a built-in notion of time. Thus, it can naturally and easily express protocol operations and physical phenomena occurring at the medium access control (MAC) level.

[1] SMURPH is an acronym for a *System for Modeling Unslotted Real-time PHenomena.*

Besides the specification language, the system provides a virtual environment for executing protocols. Thus, SMURPH specifications are directly executable. This virtual environment is based on an event-driven, discrete-time simulator which is hidden from the user. The user has an impression of running the protocol in a realistic environment which carefully reflects all relevant physical phenomena occurring in a real network, e.g., limited accuracy of independent clocks, race conditions, faulty channels. From this end, SMURPH can be viewed as a network simulator. Its most typical application is to investigate the performance of a new MAC-level protocol, comparing it to a number of other protocols in the same class. However, compared to other network simulators and evaluators (e.g, COMNET [6], GILDA [7], NETSIM [8]), SMURPH has a number of distinct features which can be stressed in the following points:

- Protocols in SMURPH are fully specified. In principle, a SMURPH specification can be "compiled into silicon," i.e., made completely functional in a mechanical way.
- In SMURPH, networks and protocols are *emulated* rather than simulated. Thus it makes sense to use SMURPH for protocol verification, e.g., conformance testing.
- SMURPH doesn't purport to be a "no-programming" system and we do not perceive it as a disadvantage. Intentionally, SMURPH is a protocol prototyping environment and with the current state of the art in program synthesis it is extremely difficult to produce novel protocols by moving icons on the screen. On the other hand, owing to the object-oriented nature of SMURPH specifications, it is natural and easy to build libraries of protocols and their components.

SMURPH and its predecessor LANSF [9] have been used to investigate the performance and correctness of a number of protocols for local and metropolitan area networks (e.g. [10, 11, 12]). In some cases, modeling in SMURPH revealed hidden implementation problems with apparently simple protocols (e.g. [13]).

The package (together with extensive documentation) is freely available to the research community via *anonymous ftp* from `menaik.cs.ualberta.ca` (129.128.4.241). The present version of SMURPH runs under UNIX[2] on a variety of equipment. There also exists a version of SMURPH for Apple Macintosh.[3]

[2] UNIX is a trademark of AT&T Bell Labs.
[3] Macintosh is a trademark of Apple Computer, Inc.

2 THE STRUCTURE OF SMURPH

The structure of the package is presented in Figure 1. A SMURPH program consisting of the protocol source code, network description, and traffic specification is translated by SMPP into a program in C++. The code in C++ is then compiled and linked with the SMURPH library. This way a stand-alone, executable module is built which can be viewed as a simulator for the system described by the user program.

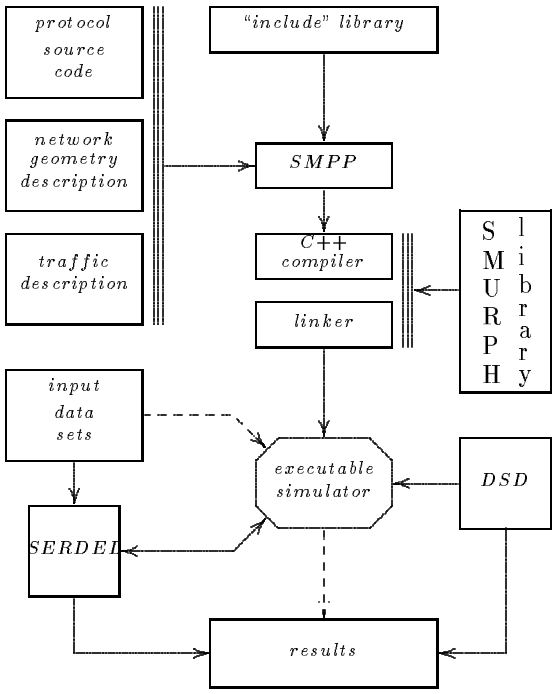

Figure 1 The structure of SMURPH.

The standard linkable SMURPH library is augmented by a source library of types (the "include" library) which is extensible by the user. This library typically contains predefined network topologies (i.e., parametrizable network configurations) and traffic patterns.

When run, the simulator produces results which may be related to the observed performance of the modeled system or to some logical aspects of its behavior. It is also possible to monitor the behavior of the modeled system on-line. This is done by DSD—a separate display program which communicates with the

simulator via UNIX IPC tools and displays selected information in a collection of windows. DSD and the simulator don't have to run on the same machine.

Instead of running the simulator directly, the user may choose to execute it under control of SERDEL[4] which provides support for coordinating multiple simulation experiments executed in the environment of a local network of computers (e.g., fast workstations). SERDEL takes care of migrating the experiments from busy machines to idle ones, starting new experiments when more machines become available, periodic checkpointing, detecting machine/system failures and restarting the affected experiments from checkpoint files, removing experiments from interactive machines when their owners show up, etc.

3 AN OVERVIEW OF THE SMURPH SPECIFICATION LANGUAGE

Protocols are described in SMURPH as collections of communicating processes. The user-supplied type and data definitions, together with the code of the protocol processes, are contained in a file or a number of files. The SMURPH specification language is an extension of C++ (the new features are briefly discussed below). A special support program (called `mks`—for *make smurph*) is provided whose purpose is to organize the process of converting the the user program into an executable simulator instance.

1 Time

The invisible SMURPH kernel is an event-driven simulator responsible for scheduling events and advancing the modeled time. SMURPH models discrete time with practically arbitrary accuracy. Time intervals are expressed in the so-called *ITU*s (*indivisible time units*) and represented as objects of type `TIME`. The precision of `TIME` is selected by the user; there is no explicit limit on this precision. For example, in a homogeneous LAN, the *ITU* can be set to the time required to insert a single bit into the network, whereas in a heterogeneous environment, the *ITU* may correspond to the *greatest common divisor* of all "natural" time units occurring in the system. Standard arithmetic operations on objects of type `TIME` and combinations of these objects with other numeric entities have been implemented by overloading the usual arithmetic operators.

[4] SERDEL is an acronym for a *Supervisor for Executing Remote Distributed Experiments on a LAN.*

Besides the *ITU*, SMURPH defines another unit of time, the so-called *ETU* which stands for the *experimenter time unit*. This unit is used for presenting simulation results.

2 Hierarchy of built-in SMURPH classes

All user-visible compound types (we assume that `TIME` is a simple type) are organized into a hierarchy presented in Figure 2. This hierarchy reflects the inclusion of types in terms of the C++ subclass concept. We assume that a (nonexistent) type *class* is the root of this hierarchy.

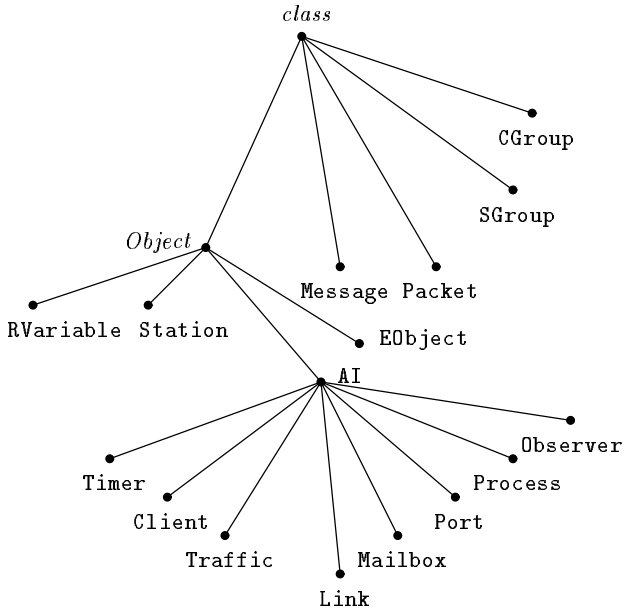

Figure 2 The hierarchy of user-visible compound types.

All objects exhibiting dynamic behavior belong to type *Object* which is not directly visible to the user. Most of the standard subtypes of type *Object* can (and sometimes must) be extended by the user. For example, type `Station` defines a skeleton for creating *stations*—the network processing units (nodes) that run the protocol. Typically, this skeleton has to be augmented by some protocol-specific elements before it can be used to build actual stations.

All *Objects* are *exposable*, i.e., there is a standard way of visualizing information

associated with them. In particular, this information can be displayed in a window (by DSD) and updated automatically while the simulation experiment is in progress.

Type `EObject` provides a base class for defining non-standard dynamic and exposable object types.

There is a special category of dynamic objects called *activity interpreters*, or *AI*s, for short. These objects define the protocol's interface with the outer world. An *AI* can be viewed as a *daemon* that absorbs protocol *activities* and turns them into future events. These events in turn can be perceived by the protocol and influence its activities. By timing future events, activity interpreters advance the modeled time. They also cover the internal event-processing mechanism with a higher-level interface which is perceived by the protocol processes as a realistic reactive environment.

Types of objects that do not act on their own, but instead are subjects of activities of other objects, are straightforward classes. An example of such a type is `Packet` describing a packet skeleton. No packet ever performs any action: packets are passive objects which are handled by active objects.

3 Type declaration

The standard declaration apparatus of C++ (which can also be used in a user-supplied protocol program) has been extended in SMURPH by a set of operations that are used to define extensions of certain standard types from Figure 2. These operations are intercepted by SMPP—the SMURPH preprocessor—which turns them into C++ constructs. The common syntax of SMURPH type declarations is as follows:

```
keyword typename :  suptypename {
   ...
};
```

where `keyword` identifies a category of types describing a common superclass of objects, e.g.: processes, stations, traffic patterns; `typename` is the name of the defined type (class); and `suptypename` is the identifier of the immediate supertype (superclass) in which the new type is contained. This last part need not occur if the defined type is derived directly from the corresponding base type (Figure 2). The opening brace ({) is sometimes preceded by a list of the

so-called argument types encapsulated in parentheses. The argument types associate the new type with other types, without including those types in the inheritance prefix of the new type. For example, processes are owned by stations. Thus, a process type is typically associated with the type of its owning stations. This association reflects the *ownership* relation rather than inheritance (stations and processes are not directly related[5] via shared attributes).

It is also possible to define new types as combinations of already defined types, taking advantage of the *multiple inheritance* apparatus of C++ (in such case `suptypename` is a list of type names). This apparatus has been adopted and modified in SMURPH.

The declaration body (the text between the braces) contains declarations of new attributes and methods. If the type extension declares a method called `setup`, this method will be automatically called when an object of the extended type is created. The `setup` method plays the role of the object *constructor*.[6]

SMURPH provides its private tool for generating objects of the standard types, or their subtypes derived in the way described above. The syntax of this operation, in its two flavors, is as follows:

```
ptr = create typename ( ... );
ptr = create (s) typename ( ... );
```

where `typename` is the type of the created object and `s` is an optional station identifier. Operation `create` can be viewed as a function returning a pointer to the created object. The text enclosed in the parentheses following the type name is passed as a list of arguments to the object's `setup` method (the parentheses can be skipped if no such method is defined or if it takes no arguments).

All *Objects*, built by `create` or declared statically, are assigned identifiers. There are several classes of such identifiers; most of them are used by DSD to identify the objects for the purpose of *exposing* them on the terminal screen. One simple identifier type is the object's serial number which typically reflects the order in which objects have been created.

Some objects, e.g., mailboxes, ports, processes, are owned by stations. An object belonging to one of these types is always created *in the context of some station*. This context can be implicit (e.g., a port created by the station's `setup`

[5] They are related indirectly by having descended from a common ancestor—*Object*.

[6] Standard argument-less constructors of C++ can also be used: the effect of the standard constructor is combined with the effect of `setup`.

method is automatically assigned to the station), or explicit, i.e., specified directly at the moment of creation. One way to indicate that the object being created is to be assigned to a specific station is to use the second variant of `create` (see above) where `s` is either a pointer to a station object or the station's serial number. Deallocation of SMURPH objects is performed by the standard C++ operation `delete`.

4 Describing network topology

The topology of the modeled network is defined as interconnection of *stations*, *ports*, and *links*. Stations are objects belonging to type `Station`; subtypes of this type can be defined to describe stations with specific characteristics. Stations represent dynamic elements of the network: they are computers capable of executing protocol processes, possibly multiple such processes in parallel.

A station is typically attached to a link (or a number of links) via ports. A port (an object of type `Port`) represents a connection of one station to one link. A link (an object of type `Link`) models a simple communication medium, e.g.: an optical fiber, a coaxial cable, or a radio channel. In principle, the standard link type can be extended (virtual functions redeclared in a link subtype may redefine the standard behavior of a link), but this possibility is seldom used. Intentionally, a link is a very simple, uniform, broadcast communication medium whose behavior is well known ([14]). If needed, multiple links can be combined to produce compound channels.

The network configuration is described in a dynamic way, i.e., by a program that explicitly assigns ports to stations and connects them to specific links in specific places. The geometry of a link is determined by a *distance matrix* which determines distances between all pairs of ports connected to the link. There are a number of ways to specify a link distance matrix; the most natural way is to explicitly assign a distance to each combination of two ports connected to the link. The distance between a pair of ports can be assigned by calling the port method `setDTo`—in the following way: `p1->setDTo (p2, d)`, where `p1` and `p2` are port pointers and `d` is the assigned distance between `p1` and `p2`. It is possible to build libraries of network configurations. The multiple inheritance mechanism of C++ (in its SMURPH flavor) is a convenient tool for this purpose.

5 Defining traffic conditions

Traffic in the modeled system consists of *messages* which arrive from outside to be processed by stations. Messages are turned into *packets* which can be transmitted and received by stations on their ports. The traffic distribution is described by a collection of the so-called *traffic patterns* which are objects of type `Traffic`. Each traffic pattern is associated with specific message and packet types. Messages and packets belonging to different traffic patterns may have different, protocol-dependent structures. The behavior of a traffic pattern is described by a set of virtual functions and processes that can be redefined in user-created subtypes of `Traffic`. In particular, libraries of traffic patterns can be built in a natural way.

6 Processes

A process consists of its private data area and a possibly shared code. Besides accessing its private data, a process can reference the attributes of the station owning the process, and some other variables constituting the so-called *process environment*. Processes can communicate in several ways, even if they do not belong to the same station.

A process type usually defines a number of attributes (they can be viewed as the local data area of the process), an optional `setup` method (used to initialize the data area when a process instance is created), and one special method defining the process code. A process type declaration has the following syntax:

```
process ptype :  supptype (fptype, stype) {
   ... attributes and methods ...
  setup ( ... ) {
     ...
  };
  states {s0, s1,  ... , sk};
  perform {
     ...
  };
};
```

where `ptype` is the name of the declared process type, `supptype` is a previously defined process type, `fptype` is the type of the process' *parent*, and `stype` is the type of the station owning the process. As for the other SMURPH

types, `supptype` can be omitted if the new process type is derived directly from `Process`. The two arguments in parentheses are also optional: they can be skipped if they are not interesting to the process.

A process is always created by another process which is considered its parent. Initially, before any user-defined process is created, there exists a built-in system process called `Kernel`. This process creates one instance of a process called `Root` whose definition must be present in a complete user-suplied protocol code. The role of `Root` can be compared to the role of function `main` in a C (or C++) program.

The process code method resembles the description of a finite state machine. The `states` declaration assigns symbolic names to the states of this machine. The first state on the list is the initial state: the process gets into this state immediately after it is created.

The operation of a process consists in responding to various events triggered by its environment. The occurrence of an event awaited by a process wakes the process up and forces it into a specific state. Then the process (its code method) performs some operations and suspends itself. Typically, among these operations are indications of future events that the process wants to perceive. Thus, the process code method can be viewed as the transition function of a finite state machine. Most often, this method has the following structure:

```
perform {
  state s0:
     ...
  state s1:
     ...
   ...
  state sk:
     ...
};
```

Two standard pointers (whose declarations are implicit and invisible) are available to the code method. They are: `F` (of type `fptype`) pointing to the process' parent and `S` (of type `stype`) pointing to the station owning the process. This way, besides having natural access to its private objects, a process can reference public attributes of its parent and its station.

7 Activity interpreters

Events that drive processes are generated by objects called *activity interpreters* (*AI*s) which model the time flow. One common element of the interface between an *AI* and a process is provided by the *AI*'s `wait` method accepting two arguments, i.e., called as `ai->wait (ev, st)`. A process invokes the `wait` method of an *AI* to request to be awakened by a specific future event generated by the *AI*. The first argument of `wait` identifies the event; its type and range are *AI*-specific. The second argument is a process state identifier: upon the nearest occurrence of the indicated event the process wants to be awakened at the given state.

A restarted process usually performs a sequence of operations (possibly exhibiting activities addressed to some *AI*s) and then it issues a number of `wait` requests describing its future waking conditions. Eventually, the process suspends itself—either by exhausting the list of statements associated with its current state or by executing `sleep`. A process that puts itself to sleep without specifying at least one waking condition becomes terminated. All the `wait` requests issued by a process at one state are combined into an *alternative* of waking conditions. It means that as soon as one of these conditions is fulfilled, the process is restarted at the state previously indicated by the second argument of the corresponding `wait` request. The other pending wait request are then forgotten; thus, if the process decides later to wait for the same (or a similar) configuration of events, all the `wait` requests must be issued from the beginning.

While a process is running, i.e., from the moment the process was restarted until it suspends itself, the simulated time does not flow. This way multiple processes can be active *simultaneously*—within the same *ITU*. One of the basic principles of SMURPH is that the modeled time only flows when it is advanced by one of the *AI*s.

The collection of standard *activity interpreters* consists of the `Timer` (used to implement alarm clocks), `Traffic` objects (responsible for traffic generation), the `Client` which can be viewed as a combination of all `Traffic` *AI*s, `Port` objects (triggering communication events that originate in links), `Mailbox` objects (providing means for inter-process communication), and `Process` objects (which are also *activity interpreters*).

Processes running at the same station communicate by exchanging signals via `Mailbox` *AI*s. A process may suspend itself awaiting the occurrence of a signal

on a `Mailbox`. Then some other process may supply this signal and restart the waiting process. Depending on the complexity of a `Mailbox`, multiple pending signals can be queued; they can also carry some information. Another (direct) way to synchronize processes is to take advantage of the fact that each process is itself an activity interpreter. Thus, it is possible for a process A to issue a `wait` request to another process B, to be awakened when B gets into some specific state (or terminates itself).

8 User interface

Typically, the protocol modeling program reads some input data and produces some output. SMURPH offers private operations for reading input data and private structural tools for producing output results. Most of the output results are generated by *exposing* objects, i.e., calling special high-level output methods associated with objects.

Another part of the SMURPH user interface is DSD—a separate program which communicates with the simulator via IPC tools. It is possible to run the simulator on one machine, e.g., a CPU server, and monitor its execution on another computer, e.g., a graphic workstation.

DSD can be used to monitor the protocol execution "on-line" and is also very useful for debugging. Windows displayed by DSD can be "stepped." In this mode, the simulator halts after every event that has "something to do" with the contents of the selected windows. An explicit user action is required to advance the simulator to the next "stepping" event. This way the user can trace the protocol, e.g., to find a reason for its misbehavior or just to understand its operation.

9 Performance measures

Some standard performance measures are computed automatically while the protocol program is running. SMURPH offers tools for collecting non-standard statistics. Objects of type `RVariable` can be created by the user and processed by a collection of standard methods implementing typical operations on random variables, e.g., adding a new sample, combining two random variables into one, and exposing (i.e., printing out or displaying) the parameters of a random variable.

10 Observers

Besides conventional tools for program debugging, such as local assertions and protocol tracing functions, SMURPH offers means for verifying compound dynamic conditions that can be viewed as an alternative specification of the modeled protocol.

Observers are process-like objects that can be used for expressing global assertions involving the combined behavior of more than one regular process. They are somewhat reminiscent of the tools with the same name described in [15].

A regular process specifies its waking conditions (by calling `wait`) in terms of events triggered by *AI*s. An observer specifies its waking conditions by invoking a sequence of `inspect` requests, each such request identifying a class of waking scenarios for a regular protocol process. By calling `inspect (s, p, ps, os)`, the observer indicates that it wants to be restarted in state `os` when a "regular" process of type `p` owned by station `s` is awakened in state `ps`. The symbol `ANY` used in place of any of the first three arguments is a wildcard—it stands for an arbitrary value. In such case, the observer is restarted by any event that matches the other parameters of `inspect`.

Configurations of `inspect` requests describe the observer's transition function which can be viewed as a description of legal state transitions in the entire collection of protocol processes. As regular assertions verify some Boolean properties of a program, observers verify state transitions in a distributed system and make sure that these transitions conform to a set of specifications. In this sense, observers are conformance testing tools.

4 ALTERNATING-BIT PROTOCOL

In this section we present a SMURPH implementation of the well-known alternating-bit protocol [16].[7] The network forms a simple ring: it consists of two stations connected by a pair of channels. One station is the *sender* and the other is the *recipient* of the traffic. One of the two channels is used for transfers from the sender to the recipient; the other channel is used by the recipient to send *acknowledgement packets* to the sender. Both channels are faulty, i.e., a packet inserted at one end of a channel may not arrive at the other end in a good shape. Packets and acknowledgements carry single-bit flags which represent their serial numbers modulo 2. These serial numbers (serial bits) are used

[7] Due to the limited space, we confine ourselves to protocol specification, leaving out network creation and traffic description.

by the sender to match acknowledgements with transmitted packets and by the recipient—to detect packet duplicates. In our implementation, we declare the following two packet types:

```
packet PacketType { int SequenceBit; };
packet AckType { int SequenceBit; };
```

to represent packets and acknowledgements. Besides the non-standard attribute `SequenceBit` each packet has a collection of standard attributes, e.g., identifying its sender, receiver, and size.

1 The sender protocol

We start from specifying the behavior of the sender station. Its structure is described by the following station type declaration:

```
station Sender {
  PacketType PacketBuffer;
  Port *IncomingPort, *OutgoingPort;
  Mailbox *SignalMailbox;
  int LastSent;
  void setup () {
    IncomingPort = create Port;
    OutgoingPort = create Port (TransmissionRate);
    SignalMailbox = create Mailbox (1);
    LastSent = 0;
  }
};
```

The station is equipped with one packet buffer (to store the packet to be transmitted to the other party) and two ports, one for each channel. Additionally, the station defines a mailbox (to communicate the two processes running at the station) and an integer counter telling the contents of the *serial bit* of the last transmitted packet. The ports and the mailbox are represented by pointers; the actual objects are created by the station's `setup` method, when the station itself is created. The `setup` argument for `OutgoingPort` gives the port's transmission rate as the number of *ITU*s required to insert a single bit into the port. This port attribute will be used to calculate the amount of time needed to transmit a packet through the port. As `IncomingPort` is only used to receive

packets, its transmission rate is irrelevant and we leave it unspecified. The `setup` argument of `SignalMailbox` indicates the capacity of the mailbox, i.e., the maximum number of *signals* that may be queued in the mailbox awaiting acceptance.

The station runs two processes communicating via the station's mailbox. One process, called the *transmitter*, transmits packets on the output port. The other process receives acknowledgements from the input port and notifies the transmitter about their arrival. The transmitter process is defined as follows:

```
process Transmitter (Sender) {
  Port *Channel;
  PacketType *Buffer;
  Mailbox *Signal;
  TIME Timeout;
  void setup (TIME tmout) {
    Channel = S->OutgoingPort;
    Buffer = &(S->PacketBuffer);
    Signal = S->SignalMailbox;
    Timeout = tmout;
  };
  states {NextPacket, EndXmit, Acked, Retransmit};
  perform {
    state NextPacket:
      if (Client->getPacket (Buffer)) {
        Buffer->SequenceBit = S->LastSent;
        Channel->transmit (Buffer, EndXmit);
      } else
        Client->wait (ARRIVAL, NextPacket);
    state EndXmit:
      Channel->stop ();
      Signal->wait (RECEIVE, Acked);
      Timer->wait (Timeout, Retransmit);
    state Retransmit:
      Channel->transmit (Buffer, EndXmit);
    state Acked:
      Buffer->release ();
      S->LastSent = 1 - S->LastEvent;
```

```
        proceed NextPacket;
    };
};
```

Upon setup, the process copies the relevant station attributes to its private pointers—for handy access. One parameter specific to the process (which must be specified when the process is created) is the acknowledgement timeout (`tmout`), i.e., the maximum waiting time for the acknowledgement of the last transmitted packet.

The process starts its operation in state `NextPacket` where it attempts to acquire a packet for transmission from the `Client` (the traffic generator). If this operation in successful, the process sets the serial bit of the packet to `LastSent` and starts transmitting the packet (operation `transmit`) on the station's output port. Otherwise, the transmitter issues a `wait` request to the `Client`—to be awakened when a new message arrives at the station. When the transmission is completed, the process wakes up in state `EndXmit` where it stops the transfer and issues two `wait` requests: one to the mailbox—for a signal from the acknowledgement receiver process (see below), and the other to the `Timer`—to detect the acknowledgement timeout. If the acknowledgement arrives before the `Timer` goes off, the transmitter wakes up in state `Acked` where it empties the packet buffer (operation `release`), flips `LastSent` to make it ready for the next packet, and returns to state `NextPacket` to acquire another packet for transmission. If the `Timer` goes off, the transmitter is restarted in state `Retransmit` where its simply retransmits the last packet (the contents of `Buffer`).

The acknowledgement receiver process is shown below.

```
process AckReceiver (Sender) {
  Port *Channel;
  Mailbox *Signal;
  states {WaitAck, AckArrival};
  void setup () {
    Channel = S->IncomingPort;
    Signal = S->SignalMailbox;
  };
  perform {
    state WaitAck:
      Channel->wait (EOT, AckArrival);
```

```
        state AckArrival:
          if (((AckType*)ThePacket)->SequenceBit == S->LastSent)
            Signal->put ();
          skipto WaitAck;
      };
    };
```

The process starts in state `WaitAck` where it awaits a packet arrival (this can only be an acknowledgement) on the station's input port. The `EOT` (*End Of Transmission*) event is triggered on the port by the last bit of a packet. This corresponds to a complete reception of an acknowledgement packet. If this event occurs, the process wakes up in state `AckArrival` where it compares the `SequenceBit` of the acknowledgement with `LastSent`. If the two values match, the acknowledgement is for the last transmitted packet in which case a signal is deposited in the mailbox (operation `put`). Otherwise, the acknowledgement is ignored and the process moves back to state `WaitAck` to await another `EOT` event on the input port.

Note that `skipto` and `proceed` are two different operations for moving directly to a specific state. Unlike `proceed`, `skipto` advances the simulated time by one *ITU*, i.e., the process wakes up in the new state one *ITU* after the moment when the operation is executed. This way events that remain pending until time is advanced (e.g., `EOT`) are given a chance to disappear. With `skipto`, the acknowledgement receiver makes sure that when it gets back to state `WaitAck`, it will not be immediately restarted by the previous `EOT` event.

Variable `ThePacket` referenced by the process in state `AckArrival` belongs to the process environment. Whenever a process is awakened by a port event caused by a packet, `ThePacket` points to the packet responsible for triggering the event. Other *environment variables* are set by events occurring in other *AI*s. The mechanism of environment variables is used by all events that beside their occurrence carry some information that may be useful to the awakened process.

2 The recipient protocol

In the light of the previous section, the structure of the recipient part of our implementation hardly needs an explanation. The recipient station is described by the following type:

```
station Recipient {
  AckType AckBuffer;
  Port *IncomingPort, *OutgoingPort;
  Mailbox *SignalMailbox;
  int Expected;
  void setup () {
    IncomingPort = create Port;
    OutgoingPort = create Port (TransmissionRate);
    SignalMailbox = create Mailbox (1);
    AckBuffer.fill (NONE, NONE, AckLength);
    Expected = 0;
  }
};
```

Its structure is very similar to the structure of the sender station. The acknowledgement buffer `AckBuffer` is preloaded by the station's `setup` method (operation `fill`) with a packet representing an acknowledgement. `Expected` tells the sequence bit of the next packet expected to arrive from the sender.

Similarly as the sender station, the recipient station runs two processes communicating via `SignalMailbox`. One of them, taking care of incoming packets, is defined as follows:

```
process Receiver (Recipient) {
  Port *Channel;
  Mailbox *Signal;
  TIME Timeout;
  states {WaitPacket, PacketArrival, ReAck};
  void setup (TIME tmout) {
    Channel = S->IncomingPort;
    Signal = S->SignalMailbox;
    Timeout = tmout;
  };
  perform {
    state WaitPacket:
      Channel->wait (EOT, PacketArrival);
      Timer->wait (Timeout, ReAck);
```

```
    state PacketArrival:
      if (((PacketType*)ThePacket)->SequenceBit ==
       S->Expected) {
        Client->receive (ThePacket, Channel);
        S->Expected = 1 - S->Expected;
      }
      Signal->put ();
      skipto WaitPacket;
    state ReAck:
      Signal->put ();
      skipto WaitPacket;
  };
};
```

The process awaits an `EOT` event on the station's input port. If no such event occurs before the timeout expires, the process moves to `ReAck` to deposit a signal in the mailbox. This will be an indication for the other process (the acknowledgement sender) to re-send the last acknowledgement. Upon a packet arrival, the process checks whether the packet is the expected one. If it is the case, the packet is received (operation `receive`) and `Expected` is flipped—to indicate the expected serial bit of the next packet to arrive from the sender. In either case, a signal is put in the mailbox which results in an acknowledgement being sent to the sender. This is done by the second process with the following structure:

```
process AcknowledgerType (RecipientType) {
  Port *Channel;
  AckType *Ack;
  Mailbox *Signal;
  states {WaitSignal, SendAck, EndXmit};
  void setup () {
    Channel = S->OutgoingPort;
    Ack = &(S->AckBuffer);
    Signal = S->SignalMailbox;
  };
  perform {
    state WaitSignal:
      Signal->wait (RECEIVE, SendAck);
```

```
    state SendAck:
      Ack->SequenceBit = 1 - S->Expected;
      Channel->transmit (Ack, EndXmit);
    state EndXmit:
      Channel->stop ();
      proceed WaitSignal;
  };
};
```

The process is driven by the signals stored in the mailbox by the receiver. Each signal forces the process to state `SendAck` where it sends and acknowledgement with the serial bit opposite to the contents of `Expected`. This way, as long as the expected packet is not received, the process will keep acknowledging the previous packet. As the receiver flips `Expected` after each reception of an expected packet, the acknowledgement sent after such reception acknowledges the received packet.

REFERENCES

[1] Budkowski, S. and Dembinski, P., "An introduction to ESTELLE a specification language for distributed systems," Computer Networks and ISDN Systems, vol. 14, pp. 3–23, 1987.

[2] "User guide for the NBS prototype compiler for Estelle." Report No. ICST/SNA - 87/3, U.S. Dept. of Commerce, National Bureau of Standards, Oct. 1987.

[3] Bolognesi, T. and Brinksma, E., "Introduction to the ISO specification language LOTOS," Computer Networks and ISDN Systems, vol. 14, pp. 25–59, 1987.

[4] Logrippo, L., Obaid, A., Briand, J. P., and Fehri, M. C., "An interpreter for LOTOS, a specification language for distributed systems," Software Practice and Experience, vol. 18, pp. 365–385, Apr. 1988.

[5] Holzmann, G., Design and Validation of Computer Protocols. Prentice-Hall Int., 1991.

[6] Mills, R. and Skinner. S., "COMNET III: The new enterprise-wide performance analysis tool," in Proceedings of MASCOTS'93, (San Diego, California), pp. 349–350, Jan. 1993.

[7] Palmer, C., Naghshineh, M., and Chen, J., "The GILDA LAN design tool," in Proceedings of MASCOTS'93, (San Diego, California), pp. 353–354, Jan. 1993.

[8] Jump, J. and Lakshmanamurthy, S., "NETSIM: A general-purpose interconnection network simulator," in Proceedings of MASCOTS'93, (San Diego, California), pp. 121–125, Jan. 1993.

[9] Gburzyński, P. and Rudnicki, P., "LANSF: a protocol modelling environment and its implementation," Software Practice and Experience, vol. 21, pp. 51–76, Jan. 1991.

[10] Bertan, B. R., "Simulation of MAC layer queuing and priority strategies of CEBus," IEEE Transactions on Consumer Electronics, vol. 35, pp. 557–563, Aug. 1989.

[11] Gburzyński, P. and Rudnicki, P., "A note on the performance of ENET II," IEEE Journal on Selected Areas in Communications, vol. 7, pp. 424–427, Apr. 1989.

[12] Gburzyński, P. and Rudnicki, P., "On executable specifications, validation, and testing of MAC-level protocols," in Proceedings of the 9th IFIP WG 6.1 Int. Symposium on Protocol Specification, Testing, and Verification, June 1989, Enschede, The Netherlands (E. Brinksma, G. Scollo, and V. C. A., eds.), pp. 261–273, North-Holland, 1990.

[13] Gburzyński, P. and Rudnicki, P., "A virtual token protocol for bus networks: Correctness and performance," INFOR, vol. 27, pp. 183–205, 1989.

[14] Gburzyński, P. and Rudnicki, P., "On formal modelling of communication channels," in IEEE INFOCOM'89, pp. 143–151, 1989.

[15] Groz, R., "Unrestricted verification of protocol properties in a simulation using an observer approach," in Proceedings of the IFIP WG 6.1 6th Workshop on Protocol Specification, Testing, and Verification, pp. 255–266, North-Holland, June 1986.

[16] Bartlett, K., Scantlebury, R., and Wilkinson, P., "A note on reliable full-duplex transmission over half-duplex lines," Communications of the ACM, vol. 12, no. 5, pp. 260–265, 1969.