An Overview of SMURPH:

an Object-oriented Configurable Simulator for Low-level Communication Protocols

Paweł Gburzyński

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

Abstract

Local area networks and other systems with the perceived functionality of closely coupled distributed computing architectures are characterized by very specific protocol design issues. Unlike in the case of high protocol layers, the design of a low-level protocol must take into account several physical phenomena which are irrelevant at a higher level. The correct interpretation of these phenomena is critical to ensure both the protocol's agreement with its specification and the efficient utilization of the channel bandwidth. We introduce SMURPH—an object oriented simulator for investigating physical communication systems at the medium access control level. SMURPH can be viewed as a combination of a protocol specification system with an emulated environment for executing the protocol specifications. This environment is realistic in the sense that it reflects all the relevant communication phenomena occurring in real physical channels. SMURPH can be used to aid the protocol design process: the designer can evaluate the protocol performance and test the protocol for correctness without resorting to an actual implementation.

1 Introduction

The design of a low-level communication protocol must deal with very specific physical phenomena occurring in the communication channels and the hardware used to interconnect them. Problems like race conditions, signal propagation time, the limited accuracy of independent clocks, etc., are typical examples of issues that must be accounted for by the protocol designer. These issues become even more critical with the advancing technology: high-speed networks call for protocols that can explore to the maximum possible extent the advantageous physical properties of the underlying hardware components.

On the other hand, there seems to be a substantial gap in the assortment of the specification tools available for aiding the protocol design process. Most researchers in this area focus their attention on higher protocol layers which are removed from the hardware far enough to allow for a reasonable degree of abstraction. In particular, the known protocol specification and verification tools as *ESTELLE* ([4, 9]), *LOTOS* ([3, 18]), or *PROMELA/SPIN* ([16]), although useful for describing high-level logical aspects of communication, fail to capture some very important low-level aspects of distributed systems, e.g., the accurate timing of events.

As far as performance evaluation is concerned, the situation is not much better. Approximate analytical methods have exhibited their limitations sufficiently many times to be treated with a limited confidence (cf. [2, 20]). Accurate analytical models exist for very few simple networks and protocols and there is no straightforward way to generalize them into more complex systems. General-purpose simulation packages are not of much help, either. An accurate description of the behavior of a multi-point broadcast channel requires about the same effort in a regular programming language as in a language oriented for simulation. The most tricky part of this description is not modeling events, but predicting their timing. The user of a protocol simulator is not excited about being able to schedule events. The problem is: *how to schedule these events* and, in our opinion, a good protocol modeling system should absorb this problem completely. The interface offered to the user should resemble (in terms of its functionality) a realistic physical environment for protocol execution.

SMURPH is an object-oriented software package for modeling discrete events in communication protocols. It can be viewed as both an implementation of a certain protocol specification language and a process-driven simulator oriented towards investigating medium access control (MAC) level

protocols. In accordance with the above postulate, the structure of the simulator is completely invisible to the user. All the simulation-related operations, as creating and scheduling individual events, maintaining a consistent notion of time, etc., are covered by a high-level interface. A protocol description in SMURPH looks like a program that could be executed on a hypothetical hardware. SMURPH emulates this hardware and thus provides a realistic environment for executing protocols described in its specification language.

Protocol execution in SMURPH can be monitored. One reason for monitoring the behavior of a protocol is to investigate its performance by gathering empirical data. Although SMURPH does not purport to be a protocol verification system, it offers some tools for protocol testing. These tools, the so-called *observers*, look like programmable dynamic assertions describing legitimate sequences of protocol actions.

SMURPH has been programmed in C++. The package is configurable: it is not a single interpreter for a variety of protocols, but it configures itself into a stand-alone modeling program for each particular application.

Although the protocol description language of SMURPH is essentially C++, the standard data types, objects, functions, and macros provided by the package extend this language substantially. With this approach, the user gets the full power of C++ combined with the power of a realistic, emulated environment for programming and executing communication protocols.

SMURPH descends from its predecessor, called LANSF([14]), which was designed and implemented by the authors in 1987. LANSF was programmed in C and has evolved a number of times from its original version. It has been successfully applied to investigating the performance and correctness of a number of protocols for local area networks ([5, 6, 7, 8, 10, 11, 12, 13]) and other distributed systems.

One disadvantage of LANSF was its low flexibility in describing compound data structures. The idea of re-implementing LANSF in C++ was born from our collaboration with the networking group at the Lockheed Space and Missile Company. At first, Objective C was tried, but after a brief preliminary design we decided to use C++ instead. C++ seems to capture all the essential features of Objective C, and at the same time is devoid of some deficiencies of that language.

2 Programming Protocols in SMURPH

We assume that the user is well acquainted with C++ and with the terminology used in object-oriented programming.

The user-supplied type and data definitions together with the code of the protocol processes are contained in a C++ file (or a number of C++ files). They look like a regular program in C++ which has access to the SMURPH libraries of types, data structures, functions, and macro-operations. A special support program (`mks`) is provided whose purpose is to merge the user protocol files with the SMURPH libraries and create a stand-alone version of the simulator.

2.1 Time

Time in SMURPH is discrete which means that there is an *indivisible time unit* (*ITU*) and two events occurring during the same *ITU* are not ordered deterministically with respect to their actual succession. The indivisible time unit may correspond to an arbitrary time interval in the modeled physical system. Time intervals are represented as objects of type `TIME`. The precision of `TIME` is selected by the user. There is no explicit limit on this precision; thus, the *ITU* can correspond to a very small unit of real time[1]. The only penalty for using a very high resolution of `TIME` is the reduced (real) execution speed. Standard arithmetic operations on objects of type `TIME` and combinations of these objects with other numeric entities are available. They are implemented by overloading the usual arithmetic operators: the user does not have to handle `TIME` objects differently from other numbers.

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 the simulation results.

2.2 Hierarchy of compound types

All user-visible compound types[2] (we conveniently assume that `TIME` is a simple type) are organized into a hierarchy presented on figure 1. This hierarchy reflects the inclusion of types in terms of the C++ subclass concept.

[1] Theoretically, one could even get down to the Planck time (about 10^{-45}s). Thus, the discrete nature of protocol modeling in *Smurph* is not really a limitation.

[2] From now on, the words *type* and *class* will be used interchangeably—to denote the same thing.

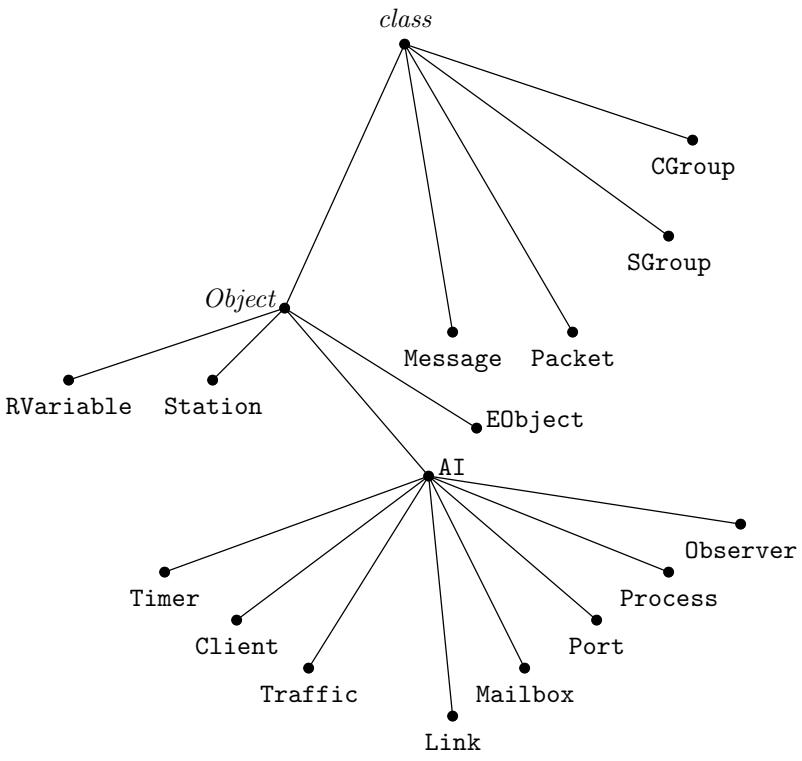

Figure 1: The hierarchy of user-visible compound types.

All compound objects in SMURPH are instances of some classes. This fact is represented by the root of the tree on figure 1. All objects exhibiting dynamic behavior belong to type `Object`. This type declares a number of standard attributes and methods that each `Object` must have. Among them are: numeric and textual identifiers (used for printing and displaying information about objects), the object qualification which determines the actual type of the object, and a virtual function describing the way the object is to be *exposed*. By *exposing* an object, we mean printing some information related to the object or displaying this information in a terminal window[3].

Most of the standard subtypes of type `Object` can (and sometimes must) be extended by the user. For example, the 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 create actual stations.

There is a special category of dynamic objects called *activity interpreters*, or *AI*s, for short. These objects provide the protocol's interface with the outer world. An *AI* can be viewed as a *daemon* that absorbs the protocol *activities* and turns them into future events.

Type `EObject` is the base class for defining non-standard exposable object types.

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 operation: packets are passive objects which are handled by active objects.

2.3 Type declaration

The standard declaration apparatus of C++ (which can be naturally used in the user-supplied protocol program) is extended in SMURPH by a collection of operations that are used to define extensions of certain standard types from figure 1. All these operations obey a common philosophy: the declaration of a SMURPH type extension has the following form:

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

[3]This part is done by a separate program called *DSD*.

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 1). The opening brace ({) is sometimes preceded by additional arguments encapsulated in parentheses. 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).

The declaration body (the text between the braces) contains declarations of new attributes and methods. From the formal point of view, the effect of a type extension operation is equivalent to the following construct of C++:

```
class typename :  public suptypename {
     ... standard prelude inserted by the macro-operation ...
     public:
     ... user-defined 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*[4].

Class types declared by means of the standard type extension macro-operations can be used to generate objects belonging to these types. Again, SMURPH provides its private tool for generating such objects. The following operation:

```
create typename ( ... )
```

creates an object of type `typename`. Operation `create` can be viewed as a function returning the pointer to the created object. The contents of the second pair of parentheses are passed as arguments to the object's `setup` method (the parentheses can be skipped if the method is undefined or if it takes no arguments). Deallocation of objects is performed by the standard C++ operation `delete`.

2.4 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`; sub-

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

types of this type can be defined by the user to describe stations with specific characteristics. Stations represent dynamic elements of the network. The protocol is described as a collection of processes (objects of type `Process`) which are executed by stations.

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.: a (possibly unidirectional) fiber optic carrier, a coaxial cable, or a radio channel.

For each pair of ports connecting some station(s) to the same link, the distance between the ports is specified as the number of *ITU*s needed to propagate a signal from one port to the other[5]. Thus, time is used to express distances in the network; the model abstracts from the actual length and propagation speed of the media.

One attribute of a port is its *transmission rate* which says how fast information can be inserted into the port. This parameter is expressed as the amount of time (in *ITU*s) required to "pump" a single bit into the port. Transmission rates of different ports (even if they are connected to the same link) need not be the same.

Stations, ports, and links are assigned numeric identifiers which generally correspond to the order in which the particular objects have been created. Besides the numeric identifiers (which are always assigned), optional textual names can be associated with these objects—to be used for printing and/or displaying.

2.4.1 Stations

A network modeled in SMURPH may include stations of several types. For example, in a star-shaped network, the central node plays the role of a switching device whose structure and behavior may be completely different from those of a regular station. The basic type for building stations is `Station`. A specific station type is declared in the following way:

```
station stype :  supstype {
    ... attributes and methods ...
};
```

where `stype` is the declared type name and `supstype` is a previously declared

[5]For a bidirectional link, the distance is the same in both directions, whereas for a unidirectional link, only one-way distance is defined.

station type (`supstype`, together with the preceding colon, can be skipped, if it is `Station`—see figure 1).

The attributes and methods of the new station type are combined with the attributes and methods of the supertype—in the regular way determined by the semantics of the C++ class declaration. In particular, the new `setup` method (if defined) subsumes the `setup` method defined in the supertype.

A station owns a number of standard attributes inherited from `Station` and `Object`. The attributes coming from `Object` are mostly protected and hidden by a number of standard methods and macro-operations. The user-visible standard part of a station type consists of two arrays of pointers:

```
Message         **MQHead,
                **MQTail;
```

pointing to message queues, each message queue holding messages belonging to a specific *traffic pattern*. These queues are filled (in most cases automatically) by the programmable traffic generator called the `Client`.

An actual station object is created by invoking `create` with the station type name as the parameter, i.e.:

```
st = create stype ( setup arguments );
```

The second pair of parentheses contains arguments of the station's `setup` function. Stations are assigned numeric identifiers (these identifiers are attributes of `Object`) in the order of their creation. The first created stations is numbered 0. The station identifier is not directly visible to the user and it cannot be changed. Two standard operations:

```
Station         *idToStation (int Id);
```

and

```
int             ident (Station *st);
```

convert station identifiers to object pointers and vice versa.

2.4.2 Links

The predefined link types offered by SMURPH are seldom extended in the protocol program. The only reasonable situation when such an extension could be warranted is the modification of the standard functions for calculating link-related performance measures. These functions, as well as other built-in performance calculating functions of SMURPH, are virtual, which

makes them easily exchangeable by a simple type extension. SMURPH provides the user with four standard link types which are in fact two different implementations of two basic link concepts: the broadcast bidirectional link (types `BLink` and `CLink`) and the unidirectional link (`ULink` and `PLink`). A specific link instance is generated by calling `create` in the following way:

```
lk = create ltype (np, at, sp);
```

where `ltype` is one of the above-mentioned four link types. The three arguments are passed to the standard link `setup` function and they have the following meaning:

- `np` The number of ports connected to the link.
- `at` The archival time limit. Activities leaving a link are kept for some time in the so-called link *archive*. The link archive can be used by the protocol program to inquire the link about its past status, down to the limit determined by the value of `at`.
- `sp` A boolean (`int`-type) flag that tells whether the link-related standard performance measures are to be calculated. This flag can assume one of two values: `ON` or `OFF`.

Only the first argument in the second pair of parentheses is required. The default values for the remaining two arguments are: `0` (archive not maintained) and `ON` (standard performance measures to be calculated), respectively.

Similarly to stations, links are also numbered in the order of their creation. A link can be referenced either via a pointer to the link object or by its numeric identifier. One situation when the user may want to reference a link explicitly is the definition of a port (see below) which must be assigned to a specific link.

2.4.3 Ports

Type `Port` is never extended by the user. A port is created in the following way:

```
pt = create Port (rate);
```

where the argument (passed to the port's `setup` method) gives the port transmission rate—the number of *ITU*s required to insert a single bit into the port.

To describe a network topology, ports, links, and stations must be organized into a single interconnected structure. A port is usually created in the context of a specific station, typically within the station's `setup` method. Such a port belongs to the given station and its purpose is to connect the station to a link. The following operation connects the port pointed to by `pt` to the link pointed to by `lk`:

```
pt->connect (lk);
```

The geometry of a link is described by a *distance matrix* that specifies 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:

```
p1->setDTo (p2, d);
```

where `p1` and `p2` are port pointers and `d` is the distance between the two ports.

2.5 Defining traffic conditions

The traffic in the network consists of *messages* which arrive from "outside" to be processed by stations. By processing a message, we mean splitting it into one or more packets and transmitting it over the network to the destination (a specific station). 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.

2.5.1 Messages and packets

A traffic pattern consists of a message inter-arrival process and two types representing messages and packets to be generated according to the pattern. These types are defined on the basis of two standard classes `Message` and `Packet`. The built-in part of the message structure (inherited from `Message`) has the following contents:

```
int             Receiver,
                TP;
TIME            QTime;
long            Length;
```

where `Receiver` is the identifier of the station that is to receive the message (a special value is used to represent a broadcast message addressed to a set of stations), `TP` identifies the traffic pattern that was used to generate the message, `QTime` tells the time when the message was generated and queued at the station (this attribute is used for measuring the message delay), and `Length` gives the message length in bits.

The user-visible standard contents of a packet have the following layout:

```
int             Sender,
                Receiver,
                TP;
TIME            QTime,
                TTime;
long            ILength,
                TLength,
                Flags;
int             isMy ();
```

Arguments `Receiver`, `TP`, and `QTime` are direct copies of the corresponding arguments from `Message`. `Sender` identifies the packet's original transmitter—the station at which the message was queued for transmission. `TTime` (the so-called *top time*) contains the time when the packet became ready for transmission (this attribute is used for calculating the packet delay). `ILength` and `TLength` specify the length of the information part of the packet (the part acquired from the message) and the total packet length (including the possible header and trailer), respectively. `Flags` is a collection of binary attributes of the packet; usually they are not referenced directly.

The method `isMy` can be used by a station to determine whether the station is the receiver of the packet (or one of the receivers, if the packet is a broadcast one).

A message type is declared in the following way:

```
message mtype :  supmtype {
     ... non-standard attributes ...
};
```

where `supmtype` is an already defined message type (it can be omitted if the new type is derived directly from `Message`). An argument-less `setup` method can be defined for a message type: it will be called whenever a new message is generated and queued at its sender station. Note that messages are seldom generated directly by the protocol program.

Similarly, a packet type can be defined as:

```
packet ptype :  supptype {
     ... non-standard attributes ...
};
```

The `setup` function, if defined, is called whenever a packet of the declared type is acquired from a message. Its purpose is to set up the non-standard packet attributes.

A `setup` function must be declared for a packet type, if this type is non-trivially extended, i.e., it actually defines some non-standard attributes. The function should expect one argument which should be a message object pointer. This argument points to the message from which the packet is being acquired.

Typically, a station defines a number of packet buffers that are used to store packets acquired for transmission, packets relayed to other stations, etc. A packet buffer is just an object of a packet type defined statically within the scope of the station type definition. A packet buffer can be either *full* (if it contains a packet) or *empty* (if no packet is currently stored in the buffer). Type `Packet` defines a collection of methods for determining the status of a packet buffer.

2.5.2 Message arrival process

The traffic in the modeled network is described as a set of independent patterns, each pattern characterized by a specific distribution parameters of the message arrival process. SMURPH offers standard tools for defining traffic patterns—objects of type `Traffic`. Should these tools prove inadequate, the user can program his private traffic patterns as extensions of type `Traffic`.

A standard traffic pattern is described by the following parameters:

- The distribution of senders and receivers. This part determines how often particular stations are selected as senders and receivers for messages generated according to the given pattern.
- Message inter-arrival time distribution. These parameters determine how much time elapses between two consecutive message arrival events.
- Message length distribution. One attribute of a message is its length in bits. For a given traffic pattern, this length can be either fixed or generated as a random number according to the message length distribution.

The distribution of senders and receivers is given as the set of the so-called *communication groups*. A single communication group (an object of type `CGroup`) defines two sets of stations: the set of senders and the set of receivers. Every station within each of the two sets is assigned a weight which can be viewed as the relative probability that this station will be chosen. The entire `CGroup` has also a weight which determines the probability of the group being used to generate the sender and the receiver. The group weight is relevant when the traffic pattern is described by more than one `CGroup`. With this two-level selection mechanism, it is possible to build refined traffic patterns. SMURPH provides shortcuts for defining simple patterns: in many cases the notion of a communication group need not be used.

A communication group is defined in terms of simpler objects called *station groups* which belong to type `SGroup`. An `SGroup` is simply a subset of stations. In particular, the set of receivers for a broadcast traffic pattern is specified as a station group.

The procedure of generating a new message and queuing it at the sending station consists of a number of steps, e.g., selection of a communication group, generation of the sender and the receiver, generation of the message length. Each of these steps is performed by a virtual method defined within type `Traffic`. By extending this type, the user can substitute his own functions for the standard methods. Similarly, an extension of `Traffic` may define private methods for calculating various traffic pattern-related performance measures.

A traffic pattern type is declared in the following way:

```
traffic ttype :  supttype (mtype, ptype) {
     ... non-standard attributes and methods ...
};
```

where `supttype` is an already defined pattern type (this part can be omitted if the new traffic type is derived directly from `Traffic`), `mtype` and `ptype` stand for the types of messages and packets to be generated according to the pattern. If these types are `Message Packet`, respectively, they (and the parentheses enclosing them) can be skipped. The base traffic pattern type `Traffic` can be used directly for traffic generation. It generates messages of type `Message` and packets of type `Packet`.

A traffic pattern type can be used to create specific traffic patterns. The most general format of `create` that serves this end is:

```
create ttype (cgl, ncgl, flags, ...)
```

where `cgl` points to an array of pointers to communication groups, `ncgl` gives the number of communication groups pointed to by `cgl`, and `flags` is a collection of binary flags specifying distribution modes for the message interarrival time and message length. The remaining arguments (their number depends on the contents of `flags`) give the numeric parameters of these distributions.

Very often a traffic pattern is described by a single communication group. In such case, `cgl` can be a direct pointer to the only communication group and `ncgl` should be omitted. It is also possible to define a traffic pattern without specifying any communication group and later build the sets of senders and receivers dynamically. This last method is recommended for simple traffic patterns.

2.6 Processes

The protocol behavior is described by a set of processes (objects of type `Process`) run at stations. A process consists of its private data area, and code which can be shared by a number of processes belonging to the same process type. A process is always associated with some station: we say that the station owns (or runs) the process.

Besides accessing its private data area, a process can reference the attributes of the station owning the process and some global 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 consists of a number of attributes (they can be viewed as local variables of the process), an optional `setup` method (which is typically used to initialize local variables when a process instance is created), and another method defining the process code. The syntax of a process type declaration is:

```
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 an already defined process type, `fptype` is the type of the process' *father*, and `stype` is the type of the station owning the process. Similarly 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 irrelevant from the viewpoint of the new type.

A process is always created by another process which is considered its father. Initially, before any user-defined process is created, there exists a system process called `Kernel` (of type `Process`) which is the father of the first user process.

A process is created similarly as other dynamic objects, by the command:

```
prcs = create ptype ( ... );
```

As usual, the arguments enclosed in the second pair of parentheses are passed to the process' `setup` method.

A process can be terminated either by itself (by executing `terminate ()`) or by another process (which executes `prc->terminate ()`, where `prc` is the pointer to the process to be killed).

The keyword `perform`[6] begins the declaration of the process code method. This method can be defined directly within the process type declaration or it can be just announced there (if `perform` is immediately followed by a semicolon) and defined later in the following way:

```
ptype::perform {
    ...
};
```

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 automatically 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 to 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:

[6] The resemblance to COBOL is accidental.

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

Process code methods are not inherited from supertypes, i.e., each process type declares its code method from scratch.

Two standard pointers (whose declarations are invisible) are available to a code method. They are: `F` (of type `fptype`) pointing to the process' father 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 father and its station.

2.7 Activity interpreters

Operations of protocol processes are driven by events. These events are generated by objects called *activity interpreters*. Before suspending itself, a process declares events that will wake it up in the future. The occurrence of the earliest of these events restarts the process. If two or more of the awaited events occur at the same time (within the same ITU), one of them is selected at random. Thus, a process is always awakened by exactly one event.

Activity interpreters model the time flow. An AI is responsible for determining how much time a specific physical phenomenon would take in the real system and advances the clock of the system model by the corresponding number of ITUs. The input to the AIs is provided by the protocol processes. We say that the protocol processes exhibit activities which are absorbed by AIs. Based on these activities, the AIs predict future events and their timing.

The interface between an AI and the protocol program depends on the AI. One element of this interface is common for all AIs; it is the method:

```
wait (ev, st)
```

which a process can invoke to request to be awakened by a specific future event. The first argument of `wait` identifies the event; its type and range are AI-specific. The second argument is a state identifier: it says that upon

the 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 corresponding `wait` request. The other conditions are then erased; 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.

2.7.1 The `Timer` *AI*

There is exactly one timer *AI* pointed to by the global variable `Timer`. By issuing a `wait` request to the timer, e.g.:

```
Timer->wait (interval, ready);
```

a process declares that it wants to be restarted at state `ready`, `interval` *ITU*s after the current moment. Another part of the timer interface is the global variable `Time` which tells the current virtual time as the number of *ITU*s elapsed since the beginning of simulation.

The `Timer` wait request is implicitly used to implement two operations for unconditional branching to another process state. Namely, the following two operations:

```
proceed stt;
```
and
```
skipto stt;
```

are equivalent to:

```
Timer->wait (0, stt);
sleep;
```

and

```
Timer->wait (1, stt);
sleep;
```

respectively. The first operation is used to branch unconditionally to the indicated state within the same *ITU*. The second operation delays the branching by one *ITU* and is useful for skipping certain events that remain pending until the time is advanced.

It is possible to request that all time intervals be determined with a certain station-dependent tolerance—to simulate the limited accuracy of real independent clocks.

2.7.2 `Traffic` *AI*s and the `Client` *AI*

Each object belonging to type `Traffic` can be viewed as an independent *AI* providing stations with messages to transmit. Moreover, there is one `Client` *AI* which can be viewed as a combination of all individual `Traffic` *AI*s. A process may poll these *AI*s for a packet to be transmitted. If no packet is available, the process may decide to suspend itself until a message arrives at the station. Let us assume that `tp` points to an object of type `Traffic`. The following operation:

```
gotit = tp->getPacket (buf, min, max, frm);
```

attempts to acquire a packet from the first-arrived message belonging to the traffic pattern `tp`. If the attempt is successful, the function returns `YES`; otherwise, if no message of pattern `tp` is queued at the station, `NO` is returned. In case of success, the packet is put into the buffer pointed to by `buf`. The length of the packet's information part is never less than `min` (dummy bits are added if the message is too short to fill the packet) nor is it greater than `max` (only a part of the message is used if the message is longer that `max`; the remaining part remains queued). The last argument of `getPacket` gives the length of the frame part, i.e., the number of additional bits to be added to the packet to furnish it with the pertinent header and trailer.

In the case when the traffic pattern of the packet to be acquired is irrelevant, the `Client` *AI* can be used. For example, by calling:

```
gotit = Client->getPacket (buf, min, max, frm);
```

a process polls the `Client` for a packet belonging to any traffic pattern. In this case, the first-arrived of all messages queued at the station is used, irrespective of its pattern. Thus, this call only fails, if all message queues at the station are empty.

Should an attempt to acquire a packet fail, a `wait` request addressed either to a specific object of type `Traffic` or to the `Client` can be issued to awake the process when a message is generated and queued at the station. In particular, with the following request:

```
Client->wait (ARRIVAL, tryagain);
```

a process declares that it wants to be awakened at state `tryagain` as soon as a message (of any pattern) arrives at the station.

2.7.3 `Port` and `Link` *AI*s

Functionally, `Link` objects are distributed and represented by collections of ports, each port being visible to the protocol program as a separate *AI*. Internally, each link can be viewed as a single *AI* with centralized processing. Protocol processes seldom reference links directly, with exception of the topology initialization phase. A protocol process may access a port *AI* for one of the following three reasons:

- to change the port status, e.g., to start or terminate a packet transmission or a jamming signal;
- to inquire the port about its present or past status;
- to issue a `wait` request to the port, i.e., to await a moment when the port gets into a specific state.

There are two types of activities that can be inserted into ports: *packet transmissions* and *jamming signals*. The latter are used in protocols based on *collision detection* to enforce the so-called *collision consensus* (cf. [19]). Jamming signals are in some sense redundant: they can be simulated by special packets; however, their presence makes it easier to program an important class of MAC-level protocols.

An activity, be it a packet transmission or a jamming signal, must be explicitly started and explicitly terminated. Assuming that `pt` represents an object of type `Port`, the following operation:

```
pt->transmit (buf, done);
```

initializes a packet transmission on port `pt`. The packet is taken from the buffer pointed to by `buf`. After the packet has been completely transmitted, the process will be awakened at state `done`. The fact that the packet has been completely transmitted does not imply that its transmission has stopped. The process is restarted at state `done` after the amount of time equal to the

total length of the packet multiplied by the port transmission rate. Then, the process is supposed to stop the transmission by calling:

```
pt->stop ();
```

usually followed by:

```
buf->release ();
```

which empties the packet buffer and updates certain performance measures.

Typical port inquiries are functions returning the time when the port was last found in some specific state. For example:

```
t0 = pt->lastBOT ();
```

assigns to `t0` the time when the last beginning of packet was heard on port `pt`.

A port `wait` request may identify one of the following future events:

`SILENCE`	the earliest beginning of a silence period (the event occurs in the current *ITU* if the port is already silent);
`ACTIVITY`	the earliest beginning of an activity period;
`COLLISION`	interference of two or more packet transmissions or the beginning of a jamming signal;
`BOT`	the beginning of a packet;
`EOT`	the end of a packet;
`BMP`	the beginning of *my* packet, i.e., a packet addressed to the station whose process issues the `wait` request;
`EMP`	the end of *my* packet;

and a few others.

If a process is awakened by an event that has been triggered by a packet transmission, the variable `ThePacket` (of type `Packet`) points to packet object and another variable, `ThePort`, points to the port on which the packet is heard. These two global variables (as well as a few other variables settable by various events) belong to the *process environment*. Their purpose is to pass to the awakened process some information related to the restarting event.

2.7.4 `Mailbox` *AI*s

`Mailbox` *AI*s provide means for inter-process communication. A mailbox, which is always owned by some specific station, can be viewed as a depository for messages passed among the processes running at the station. The capacity of a mailbox is limited and determined at its creation. Depending how a mailbox is defined and created (the standard type `Mailbox` can be extended by the user), its functionality may resemble a simple signal (interrupt) passing mechanism or a FIFO-type storage for compound objects. The standard type `Mailbox` (without extension) can be used to pass simple signals with no information content.

A mailbox can be declared statically, as part of a station type definition, or created dynamically within the context of the station to which it is supposed to belong. One `setup` argument that can be specified upon mailbox creation is the capacity. If no capacity is specified when the mailbox is created, default capacity 0 is assumed. This means that no items will ever be stored in the mailbox: if no process is waiting for an item and one arrives, it is ignored.

Besides the `wait` method, the `Mailbox` *AI* offers two operations: `put`—to send a new item to the mailbox and `get`—to retrieve the front element from the mailbox queue.

By issuing the following mailbox `wait` request:

```
mb->wait (NEWITEM, gotit)
```

a process awaits the moment when a new item is put into the mailbox. By using `RECEIVE` instead of `NEWITEM`, the process can request to automatically accept the item when it arrives. In such case, the value of the item will be returned to the process via the environment variable `TheItem`.

Another example of an event that can be specified as the first argument of the `wait` method is a nonnegative number n. The event is triggered when the number of elements stored in the mailbox reaches n.

2.7.5 `Process` *AI*s

One more way of communicating processes is to take advantage of the fact that each protocol process is an independent *AI*. Assume that `prc` is a pointer to a process object. By issuing the following request:

```
prc->wait (prcstate, mystate);
```

a process requests to be awakened at `mystate` as soon as the process pointed to by `prc` reaches `prcstate`. One special event that can be specified as the first argument of the `Process wait` method is `DEATH` which represent the termination of the process in question. For example, by executing the following sequence:

```
prc = create mychild ( ... );
prc->wait (DEATH, done);
sleep;
```

a process creates a child process and blocks itself until the child is terminated. Thus, the child process can be viewed as a *subroutine* called by its creator, as opposed to the situation when the child and its father operate concurrently.

2.8 User interface

Typically, the protocol program reads some input data, although one can imagine a situation in which no input data is required, i.e., all parameters are hard-coded into the program. SMURPH offers private operations for reading numbers from input (the standard C++ tools can be used as well) which ignore all non-numeric contents of the input file and provide shortcuts for reading sequences of related numbers. It is easy to insert textual comments into the data file: they are just bypassed and ignored.

A similar collection of basic output tools is provided. In most cases, however, the output results are produced by exposing, i.e., calling high-level output methods associated with particular objects. For example, assuming that `tpt` points to a traffic pattern, the call:

```
tpt->printPfm ();
```

writes to the output file the set of performance measures associated with the pattern `tpt`.

Usually, a single object defines a number of *exposing* methods; some of them can be used for debugging. In most cases, these methods can be subsumed or augmented by user-defined methods in type extensions.

Another part of the SMURPH user interface is the *dynamic status display* which allows the user to monitor the protocol behavior on-line. The display is handled by a separate, in principle exchangeable, program which communicates with SMURPH using a well-defined and reasonably simple protocol. In particular, 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.

2.9 Performance measures

The global performance of a network is usually expressed as the correlation of *throughput* and *delay*. By default, the performance statistics collected by SMURPH during simulation include three different measures of delay for each traffic pattern; they are: the packet delay (excluding the message queuing time), the absolute message delay which includes the message queuing time, and the weighted message delay which includes the message queuing time and reflects the fact that long messages may be split into multiple packets reaching their destination at different times. Each measure is presented in the form of a random variable (an object of type `RVariable`) with a number of parameters including the minimum, maximum, mean, and the standard deviation. The throughput is measured separately for each link and globally for the entire network.

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 printing out or displaying the parameters of a random variable.

2.10 Observers

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

Observers are process-like objects that can be used for expressing global assertions that involve the combined behavior of more than one regular process. An observer may specify that it is to be awakened whenever a process is restarted at a specific state. A similar approach to implementing self-checking distributed programs was proposed earlier ([1]) and recently refined ([15]). Observers can be seen as a tool for writing non-executable protocol specifications in terms of dynamic formulas describing possible successions of protocol states.

3 An implementation of Hubnet

In this section, we present a complete example of a network and its protocol programmed in SMURPH. This network is *Hubnet* (cf. [17]) which is a star-

shaped architecture based on broadcast-type communication. The central station in this star is a switching device responsible for resolving contention to the broadcast channel.

3.1 The protocol

Hubnet consists of a number of regular stations connected to a central switching device—the so-called *Hub*. Each regular station is connected to the *Hub* via two channels: the broadcast channel and the selection channel. All the broadcast channels are connected together and form a uniform broadcast medium (a single link). The selection channels are not connected: each selection channel is a separate link attaching one regular station to the *Hub*.

The broadcast channel can be in one of two states which are recognized and toggled by the *Hub*. These states are: *busy*, meaning that a packet is being transmitted on the broadcast medium, and *idle*, when, according to the *Hub*'s perception, the broadcast channel is inactive.

A station willing to transmit a packet sends it on the selection channel. When the packet arrives at the *Hub*, the *Hub* determines the state of the broadcast link. If the broadcast link is *idle*, the *Hub* connects the selection channel to the broadcast link and marks the broadcast link as *busy*. Thus, the packet is relayed on the broadcast channel and it will be heard by all stations in due time. As soon as the transmission is complete, i.e., the *Hub* detects silence on the selection channel, the channel is disconnected from the broadcast link and the broadcast link status is changed to *idle*.

If a packet arrives on a selection link while the broadcast channel is *busy*, the packet is simply ignored. The transmitting station listens for the echo of its packet on the broadcast channel. If the echo does not arrive after some time (depending on the distance of the station from the *Hub*), the station assumes that its transmission has been blocked and tries again.

3.2 Startup and termination

SMURPH expects that the user-supplied protocol description defines a special process type named `Root`. The simulator will create exactly one process of this type. This process is responsible for building the network model, creating the protocol processes, and detecting the end of simulation. This part of our implementation of *Hubnet* is listed below.

```
process Root {

  Traffic *TPat;
```

```
  Hub     *TheHub;

  void  initTopology (), initTraffic (), initProtocol ();

  states {Start, Exit};

  perform {
     state Start:
       setEtu (1000);
       setLimit (200, 1000000000);
       initTopology ();
       initTraffic ();
       initProtocol ();
       Kernel->wait (DEATH, Exit);
     state Exit:
       TPat -> printPfm ();
  };
};
```

The `Root` process has two states. The first state (`Start`) is triggered automatically when the process is created. At this state, the process sets up the relation between the ETU and ITU (in this case one ETU is declared to be equal to 1000 ITUs), defines the simulation exit condition (the simulation will stop as soon as 200 messages are completely received at their destinations or the modeled time reaches $1,000,000,000$ ITUs, whichever happens first), and calls three user-defined functions. The first of these functions builds the network, the second defines the traffic conditions, and the third starts the protocol execution. Then, the process awaits the `DEATH` of an internal process called `Kernel`. This event marks the end of the simulation run and when it happens the `Root` process is restarted at state `Exit`.

Variable `TPat` is set by the function `initTraffic` (see below) to point to the only pattern describing the traffic conditions in the network. By invoking the `printPfm` method of this object, `Root` prints out the standard set of performance measures associated with the traffic pattern. Then the simulation run is actually finished.

3.3 Station types

The network consists of two different types of stations: regular stations (which are assumed to be homogeneous), and the *Hub* acting as a switching device. The following initial part of the protocol program file defines (among other things) the station data types:

```
identify (Hubnet);

int     NUsers;
TIME    EchoTimeout,
        HdrRecTime;
long    MinPL, MaxPL, FrameL, TRate;

enum {Idle, Busy};

station Hub {

  int    Status;
  Port   **HPorts;

  void setup () {
      HPorts = new Port* [NUsers+1];
      for (int i = 0; i <= NUsers; i++)
        HPorts [i] = create Port (TRate);
      Status = Idle;
  };
};

station User {

  Port    *SPort, *BPort;
  Mailbox *StartEW, *ACK, *NACK, *Timeout;
  Packet  Buffer;

  void setup () {
      StartEW = create Mailbox (1);
      ACK = create Mailbox (1);
      NACK = create Mailbox (1);
      Timeout = create Mailbox (1);
      SPort = create Port (TRate);
      BPort = create Port (TRate);
  };
};
```

The operation `identify` is used to assign a name to the protocol. This name will be included in the header of the output file.

Variable `NUsers` (its contents are read from the input file by `initTopology`) contains the number of regular stations in the network. A regular station is represented by an object of type `User`. Type `Hub` is used

to represent the central switching station.

Each regular station has two ports: one to the selection link (pointed to by `SPort`), the other (`BPort`) to the broadcast link. One packet buffer (`Packet`) is used to store a packet awaiting transmission.

The four mailboxes created at each regular station will be used to synchronize the station's processes. All the mailboxes are created with capacity 1. They represent simple signal passing devices, each device capable of storing one pending signal.

The *Hub* has `NUsers`+1 ports pointed to by the entries in array `HPorts`. Port number 0 connects the *Hub* to the broadcast channel, the remaining ports represent the *Hub*'s perception of the selection links connecting it to the regular stations. One additional attribute of the *Hub* is the integer variable `Status` assuming two enumeration values `Idle` and `Busy`.

Ports (and also mailboxes) are created upon `setup` of their stations. Variable `TRate` (initialized from the input data file) is the transmission rate of the network (common for all ports). The role of the remaining global variables declared within the above code fragment is explained below.

`EchoTimeout` represents the amount of time that elapses until the transmitter of a packet assumes that the packet has not made it through the *Hub*, if the packet echo does not appear in the meantime on the broadcast port. `HdrRecTime` denotes the time needed to recognize the packet's sender. A station awaiting the echo of its packet must wait for `HdrRecTime` since the moment it detects the beginning of a packet, until it can definitely tell whether the packet was transmitted by the station. `MinPL`, `MaxPL`, and `FrameL` determine the packet framing, i.e., the minimum packet length, the maximum packet length, and the combined length of the packet header and trailer, respectively.

3.4 Network configuration

The network configuration is defined by the following function:

```
void    Root::initTopology () {

  long    LinkLength;
  Link    *slk, *blk;
  User    *s1, *s2;

  readIn (NUsers);
  readIn (LinkLength);
  readIn (TRate);
```

```
  blk = create Link (NUsers+1);
  TheHub = create Hub;
  TheHub->HPorts [0]->connect (blk);

  for (int i = 1; i <= NUsers; i++) {
    slk = create Link (2);
    s1 = create User;
    s1->BPort->connect (blk);
    s1->SPort->connect (slk);
    TheHub->HPorts [i]->connect (slk);
  }

  for (i = 1; i <= NUsers; i++) {
    s1 = (User*) idToStation (i);
    s1->BPort->setDTo (TheHub->HPorts [0], LinkLength);
    s1->SPort->setDTo (TheHub->HPorts [i], LinkLength);
    for (int j = i+1; j <= NUsers; j++) {
      s2 = (User*) idToStation (j);
      s1->BPort->setDTo (s2->BPort, LinkLength+LinkLength);
    }
  }
};
```

The function starts with reading in the number of regular stations (`NUsers`), the length of a link segment connecting a regular station with the *Hub* (we assume that all these segments are of the same length), and the transmission rate (common for all ports).

Then `initTopology` creates the broadcast link (with `NUsers`+1 ports) and the *Hub* station. The broadcast port of the *Hub* is connected to the broadcast link.

The first `for` loop creates the `NUser` selection links and regular stations. Each selection link has two ports. This loop also connects the two station ports to the appropriate links and the other end of the selection link to the *Hub*.

The next two loops assign distances between all pairs of ports of each link. The outer loop executes for all regular stations: it defines the distance between a regular station and the *Hub*. The inner loop defines the distance between a pair of selection ports of different regular stations. In the symmetric star topology, this distance is equal twice the distance of a regular station from the *Hub*.

3.5 Traffic definition

The traffic in our network is described by one traffic pattern (an object of type `Traffic`)—in the following way:

```
void    Root::initTraffic () {

  SGroup *Snd, *Rcv;
  CGroup *Cgr;
  double MeanMIT, MeanMLE;

  readIn (MinPL);
  readIn (MaxPL);
  readIn (FrameL);
  readIn (MeanMIT);
  readIn (MeanMLE);

  Snd  = create SGroup (-1, &TheHub);
  Rcv  = Snd;
  Cgr  = create CGroup (Snd, Rcv);
  TPat = create TPattern (Cgr, MIT_exp+MLE_exp, MeanMIT, MeanMLE);

  readIn (EchoTimeout);
  readIn (HdrRecTime);
};
```

The function starts with reading traffic-related protocol parameters from the input file. Then, it creates a group of stations: this group consists of all the stations in the network with exception of the `Hub`, i.e., of all regular stations. Note that the `Hub` does not receive any messages to transmit from outside: it only relays packets arriving from regular stations.

The only traffic pattern (pointed to by `TPat`) created by the function is based on a communication group consisting of two identical station groups, each containing all the regular stations. With this definition, all regular stations contribute the same amount of traffic to the network load and the global traffic pattern is uniform. Both the mean message inter-arrival time and the message length are exponentially distributed with the mean values read from the input file.

The last operation performed by `initTraffic` is reading the two delay values `EchoTimeout` and `HdrRecTime`.

3.6 The protocol code

Each regular station runs three "permanent" processes (`Xmitter`, `Receiver`, and `Monitor`) and occasionally spawns one additional process (called `Trumpet`) which disappears after a while.

3.6.1 The transmitter

The `Xmitter` process transmits packets on the station's selection port and determines whether the transmission has been successful. Its definition is listed below.

```
process Xmitter (User) {

  Port    *SPort;
  Packet  *Buffer;

  setup () {
      SPort  = S->SPort;
      Buffer = &(S->Buffer);
  };

  states {NewPacket, Retransmit, Done, Confirmed, Lost};

  perform {
    state NewPacket:
      if (Client->getPacket (Buffer, MinPL, MaxPL, FrameL))
        proceed Retransmit;
      Client->wait (ARRIVAL, NewPacket);
    state Retransmit:
      SPort->transmit (Buffer, Done);
      S->StartEW->put ();
      S->NACK->wait (RECEIVE, Lost);
    state Done:
      SPort->stop ();
      S->NACK->wait (RECEIVE, Retransmit);
      S->ACK->wait (RECEIVE , Confirmed);
    state Confirmed:
      Buffer->release ();
      proceed NewPacket;
    state Lost:
      SPort->abort ();
      proceed Retransmit;
   }
```

```
};
```

The two local attributes of the process, `SPort` and `Buffer`, are set by the `setup` function to identify the station's selection port and the only packet buffer, respectively.

If a packet becomes ready for transmission, the transmitter wakes up at state `Retransmit`. Having acquired a packet from the `Client`, the process sends a signal to the `Monitor` (by putting a dummy item into the `StartEW` mailbox) and starts to transmit the packet on the station's selection port. Note the reference to the `S` attribute which points to the station owning the process. Thus, `S->StartEW` identifies one of the mailboxes declared at the station.

Having notified the `Monitor` about the beginning of a packet transmission, the transmitter issues a wait request to the `NACK` mailbox for a possible "negative acknowledgement" signal coming from the `Monitor`. If the packet is long enough, it may happen that the `Monitor` decides to send the `NACK` signal before the packet has been entirely transmitted. In such case, there is no point in continuing the transmission and the process aborts the transfer. Note that nothing will be lost if the "positive acknowledgement" signal arrives from the `Monitor` before the transfer is complete. The signal will simply remain pending in the `ACK` mailbox and it will be immediately `RECEIVE`d when the transmitter issues a wait request for it at state `Done`.

3.6.2 The monitor

The purpose of the `Monitor` is to detect echo timeouts and notify the transmitter about the success or failure of the last transfer attempt. The `Monitor` is implemented with the help of an auxiliary process which is created and killed by the `Monitor` dynamically. To understand the rationale of such a solution let us discuss the expected behavior of the `Monitor`.

The process has nothing to do until it is awakened by the `StartEW` signal from the transmitter. Then, it has to set up an alarm clock for `EchoTimeout` time units. While waiting for the timer to go off, the `Monitor` is supposed to listen to the broadcast port. Whenever a packet arrives there, the process must wake up, examine the packet header to identify the sender, and, if the sender happens to be different from the process' station, the waiting must continue until either the right packet arrives or the alarm clock goes off. One natural way to implement the alarm clock is to issue a simple timer request. Let us note, however, that whenever the `Monitor` is interrupted

by a packet on the broadcast port and finds out that the packet has been sent by some other station, it will have to set the alarm clock again for an appropriately updated time interval. Thus, the process will have to keep track of how much time it has been waiting so far and how much time still remains to wait.

It seems to be a somewhat simpler approach to create a special process to implement an independent alarm clock that can wait for the requested amount of time without unsolicited interruptions. This is the role of the `Trumpet` process which is defined as follows:

```
process Trumpet {

  TIME  Delay;

  states {Start, Play};

  perform {
    state Start:
      Timer->wait (EchoTimeout, Play);
    state Play:
      S->Timeout->put ();
      terminate ();
   }
};
```

When the `Trumpet` is created, it issues a wait request to the timer for `EchoTimeout` time units. When that delay elapses, the process is restarted at `Play` where it sends a timeout signal (by storing a dummy item in the `Timeout` mailbox) and terminates.

The `Monitor` code looks as follows:

```
Monitor::perform {
    state WaitSignal:
      S->StartEW->wait (RECEIVE, WaitEcho);
    state WaitEcho:
      AClock = create Trumpet;
      proceed Waiting;
    state Waiting:
      BPort->wait (BOT, Packet);
      S->Timeout->wait (RECEIVE, NoEcho);
    state Packet:
      if (ThePacket -> Sender == ident (S)) {
        timer->wait (HdrRecTime, Echo);
        S->Timeout->wait (RECEIVE, NoEcho);
```

```
      } else
        skipto Waiting;
    state Echo:
      if (S->Timeout->erase () == 0) AClock -> terminate ();
      S->ACK->put ();
      proceed WaitSignal;
    state NoEcho:
      S->NACK->put ();
      proceed WaitSignal;
};
```

Upon reception of the `StartEW` signal from the transmitter, the `Monitor` creates an instance of `Trumpet` (pointed to by `AClock`). Then the process examines all packets appearing on the station's broadcast port until it either gets the `Timeout` signal or finds a packet sent by the current station. Note that the `Monitor` simulates the operation of recognizing the packet's sender. We assume that the sender can be determined `HdrRecTime` time units since the moment when the beginning of the packet was heard. Intentionally, this delay corresponds to receiving (a part of) the packet header.

If the awaited packet echo arrives before the alarm clock goes off, the `AClock` process is killed, but only if the `Timeout` signal is not already pending at the station. The `erase` method empties the mailbox and returns the number of removed items. The pending `Timeout` signal means that `AClock` has already killed itself: the two events have occurred at the same *ITU*.

3.6.3 The receiver

The `Receiver` process (run by a regular station) is completely independent of the other processes: its sole purpose is to listen to the broadcast port and detect packets addressed to its owner.

```
Receiver::perform {
    state Wait:
      BPort->wait (EMP, Packet);
    state Packet:
      client->receive (ThePacket, BPort);
      skipto Wait;
};
```

When started for the first time, the process issues a `wait` request to the broadcast port specifying that it wants to be restarted by the earliest `EMP` event—the end of a packet addressed to its station. This corresponds to the complete reception of a packet and is the only interesting event from the

viewpoint of the `Receiver`. Upon reception of a packet, the process calls the `Client`'s method `receive` that updates some performance measures.

Having received the packet, the process is ready to await the arrival of the next one. However, before branching back to `Wait`, the `Receiver` has to make sure that the last `EMP` event has disappeared from the port. This is the reason for using `skipto` instead of `proceed`. Otherwise, the process would loop infinitely on the same event: the modeled time does not flow automatically while a process is running.

3.6.4 The *Hub* process

The *Hub* station runs `NUsers` identical copies of the same process, each copy servicing one selection port. The type of this process is declared as:

```
process HubProcess (Hub) {

  Port  *SPort, *BPort;

  void setup (int pn) {
      SPort = S->HPorts [pn];
      BPort = S->HPorts [0];
  };

  states {Wait, NewPacket, Done};

  perform {
    state Wait:
      SPort->wait (BOT, NewPacket);
    state NewPacket:
      if (S->Status == Busy) skipto Wait;
      S->Status = Busy;
      BPort->transmit (ThePacket, Done);
    state Done:
      BPort->stop ();
      S->Status = Idle;
      proceed Wait;
  };
};
```

Each of the *Hub* processes has access to one selection port and to the broadcast port. The selection port serviced by the process is specified as the `setup` parameter when the process is created (see below).

The Hub process awaits a packet arrival on its selection port (event BOT). If the packet arrives while the broadcast medium is busy (the *Hub* is already relaying a packet from another selection port), the packet is ignored (the BOT event is *skipped*) and the process awaits another packet arrival. If the broadcast channel is idle, it is immediately marked as busy and the packet is re-transmitted on the Hub's broadcast port. When the transfer is complete, the Hub status is reset to idle and the process resumes waiting for another packet to relay.

3.6.5 Protocol startup

The last element of the startup action performed by the Root process is the creation of the protocol processes; this part is done by the following function:

```
void    Root::initProtocol () {

  for (int i = 0; i < NUsers; i++) {
    TheStation = idToStation (i);
    create Xmitter;
    create Receiver;
    create Monitor;
  }
  TheStation = TheHub;
  for (i = 1; i <= NUsers; i++) create HubProcess (i);
};
```

With the first loop, the three processes are created for each regular station. Before the create operations are issued, the context variable TheStation is set to point to the station object which is supposed to own the created processes. The Hub runs NUsers versions of the same HubProcess. The parameter passed to the setup function of this process identifies one of the Hub's selection ports.

4 Summary

We have presented a software package for specifying low-level communication protocols in a realistic executable environment. The package, SMURPH, has descended from an earlier product, designed and implemented by the present authors, called LANSF. In the three years since LANSF was created, it has proven to be more flexible than planned. It has found applications in

investigating a much wider class of physical systems than MAC-level protocols for local area networks.

The success of LANSF seems to be due to its ability to model physical events in communication media at an arbitrarily low level. We believe that the new version of the system will prove more useful and user friendly than its predecessor.

References

[1] J. M. Ayache, P. Azéma, and M. Diaz. Observer: a concept for on-line detection of control errors in concurrent systems. In *Proceedings of the 9th Symposium on Fault-Tolerant Computing*, pages 1–8, Madison, WI, June 1979.

[2] D. Boggs, J. Mogul, and K. C.A. Measured capacity of an ethernet: Myths and reality. Wrl research report 88/4, Digital Equipment Corporation, Western Research Laboratory, 100 Hamilton Avenua, Palo Alto, California, 1988.

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

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

[5] W. Dobosiewicz and P. Gburzyński. Ethernet with segmented carrier. In *Proceedings of IEEE Computer Networking Symposium*, pages 72–78, Washington, DC, Apr. 1988.

[6] W. Dobosiewicz and P. Gburzyński. Improving fairness in CSMA/CD networks. In *Proceedings of IEEE SICON'89*, Singapore, July 1989.

[7] W. Dobosiewicz, P. Gburzyński, and P. Rudnicki. An Ethernet-like CSMA/CD protocol for high speed bus LANs. In *IEEE INFOCOM'90*, pages 238–245, 1990.

[8] W. Dobosiewicz, P. Gburzyński, and P. Rudnicki. On two collision protocols for high speed bus LANs. *Computer Networks and ISDN Systems*, 1990. (Submitted.).

[9] 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.

[10] P. Gburzyński and P. Rudnicki. A better-than-Token protocol with bounded packet delay time for ethernet-type LAN's. In *Symposium on the Simulation of Computer Networks*, pages 110–117, Colorado Springs, Co., Aug. 1987. IEEE.

[11] P. Gburzyński and P. Rudnicki. Using time to synchronize a Token Ethernet. In *Proceedings of CIPS Edmonton '87*, pages 280–288. Canadian Information Processing Society, Nov. 1987.

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

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

[14] P. Gburzyński and P. Rudnicki. LANSF: a protocol modelling environment and its implementation. *Software Practice and Experience*, 21(1):51–76, Jan. 1991.

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

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

[17] A. Hopper, S. Temple, and R. Williamson. *Local Area Network Design*. Addison-Wesley, 1986.

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

[19] R. Metcalfe and D. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[20] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, Mar. 1986.