SIDE/SMURPH:

a Modeling Environment for Reactive Telecommunication Systems

Version 3.21
September 17, 2012

Pawel Gburzynski

Contents

1 Introduction

The package version described by this manual is identified in the title. If the version number of your simulator is higher, some of its features may not be covered by this manual.

1.1 A historical note

The present package is the outcome of an old and long project, which started in 1986 as an academic study of collision-based (CSMA/CD) medium access control (MAC) protocols for wired (local-area) networks. About that time, many publicized performance aspects of CSMA/CD-based networks (like Ethernet) had been subjected to heavy criticism from the more practically inclined members of the community—for their apparent irrelevance and overly pessimistic confusing conclusions. The culprit, or rather culprits, were identified among the popular collection of analytical models, whose cavalier application to describing poorly understood and crudely approximated phenomena had resulted in worthless numbers and exaggerated blanket claims.

Our own studies of low-level protocols for local-area networks, on which we embarked at that time, were aimed at devising novel solutions, as well as dispelling myths surrounding the old ones. Owing to the fact that exact analytical models of the interesting systems were nowhere in sight, the performance evaluation component of our work relied heavily on simulation. To that end, we developed a detailed network simulator, called LANSF,[1] which, in addition to facilitating performance studies, was equipped with tools for asserting the conformance of protocols to their specifications. All the relevant low-level phenomena, like finite propagation speed of signals, race conditions, imperfect synchronization of independent clocks, and so on, were carefully reflected in the system models. In fact, those models had the appearance of programs actually *implementing* the protocols on some abstract hardware. Thus, our system was often referred to as an emulator rather than simulator.

Around 1989 we became involved in a project with Lockheed Missile and Space (now Lockheed-Martin) aimed at the design of a high-performance reliable LAN for space applications. Between 1989 and 2003, LANSF was reorganized, generalized, documented, and re-implemented in C++ receiving the look of a specialized programming language. Under the new name SMURPH,[2] in addition to verifying novel (and old) networking concepts, the package was used for education (in a graduate course on protocol design). SMURPH, with many examples of LAN protocols, was extensively described in a 700+ page book.[3]

[1] Local Area Network Simulation Facility.

[2] A System for Modeling Unslotted Real-time PHenomena.

[3] Pawel Gburzynski, *Protocol Design for Local and Metropolitan Area Networks*, Prentice Hall, 1996.

In mid-nineties, we noticed that the close-to-implementation look of SMURPH models offered a bit more than a powerful simulation environment for low-level network protocols. Namely, it appeared to us that the programming paradigm of SMURPH models might be useful for implementing certain types of real-life applications. The first outcome of this observation was an extension of SMURPH into a programming platform for building distributed controllers of physical equipment represented by collections of sensors and actuators. Under its new name, SIDE,[4] the package encompassed the old simulator equipped with a means of interfacing its programs to real-life equipment. Then, in 2002, we created PicOS—a low-footprint operating system for microcontrollers, whose stack-less threads were directly based on the idea of SMURPH/SIDE *processes*.

The most recent (August 2006) step in the evolution of LANSF, SMURPH, SIDE is the addition of a generic model of a wireless channel, which makes it possible to use the package for modeling wireless networks, especially ad-hoc networks consisting of a potentially large number of possibly mobile nodes. The generic nature of the channel model allows the user to plug into it functions describing the propagation characteristics of the actual wireless medium, e.g., the impact of distance on signal level and interference, as well as the relationship between the signal-to-interference ratio and the probability of a successful packet reception.

Another recent feature is the inclusion of the so-called *visualization mode* for simulation. When run in this mode, the simulator attempts to synchronize the virtual time of the model to the prescribed intervals of real time. This should not be confused with the *real mode*, in which the program controls real equipment. The purpose of the visualization mode is to demonstrate (visualize) the behavior of the modeled system by making intervals of virtual time perceptible by the user as real time intervals of proportional duration. This is only possible if the simulator can "catch up" with real time. In many cases, the program can only render a slow motion visualization of the model.

To avoid unnecessary complication, the package will be called SIDE in the sequel. Also, to avoid too many changes in the last version of the manual, we will introduce the package from its not necessarily most popular "SIDE," namely as a platform for implementing control programs driving real physical equipment. This introduction will not be confusing, if the reader keeps in mind that network simulation is by far the most relevant mission of our system. If simulation is the reader's only concern, then we recommend skipping all paragraphs that refer to the "real mode of operation."

1.2 Reactive systems

By a reactive system we understand any physical system that responds to external *stimuli* and triggers *events* that may be perceived by its observer. This definition is wide enough to encompass all physical devices that exhibit some organized behavior, as well as

[4] Sensors In a Distributed Environment.

interconnections of such devices into possibly large networks of dynamic and interacting components. One example of such a network is a modern factory in which manufacturing equipment is interconnected and organized around a common goal—the production of some goods.

For the purpose of computer control and algorithmic description, a reactive system is viewed as a communication network equipped with some processing power, whose terminal devices are of two basic types: *sensors* and *actuators*—see figure 1. The sensors perceive the world and transform this perception into events. Those events propagate to the processing agents of the network (i.e., programs run on computers), where they are interpreted and transformed into messages sent to the actuators. In response to those messages, the actuators perform specific physical actions that make the system behave in a prescribed way.

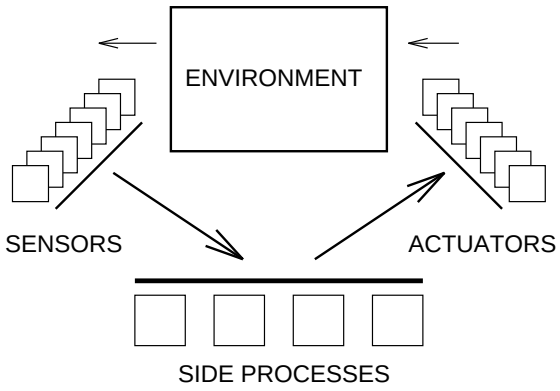

Figure 1: A reactive system.

One immediate application area for SIDE is any industrial system that can be represented by a network of sensors and actuators. The primary goal of SIDE is to provide a platform for developing, testing, and executing control programs for "intelligent" networks of sensors and actuators that may be interconnected or accessed via the Internet.

1.3 Control

SIDE offers an object-oriented programming language for implementing control programs for reactive systems. Programs expressed in this language are executed by the SIDE kernel that interfaces them to networks of sensors and actuators and coordinates their multiple threads.

Besides conventional tools for program debugging, such as local assertions and tracing functions, SIDE offers means for verifying compound dynamic conditions that can be

viewed as an alternative specification of the control program. These tools, the so-called *observers*, are thread-like objects that can be used for expressing global assertions involving the combined behavior of more than one regular thread. 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.

A control program in SIDE can be implemented as a single multithreaded, event-driven module, or as a set of modules run on independent (possibly diverse) machines connected via the Internet. These modules can communicate with operators (human supervisors) on other machines via standard Internet browsers capable of running Java applets. One can enhance the reliability of a control program by providing several copies/versions of a single module. Those multiple copies can be programmed differently, e.g., by different people, and they can be run on different machines at different geographical locations, yet they may end up controlling the same physical fragment of the system.

1.4 Simulation

One distinct feature of SIDE is the possibility of replacing fragments of the interface to a real reactive system with their simulated counterparts. This way, it is possible to develop, test, and debug a control program together with the development, testing, and debugging of the underlying physical system to be controlled. No part of the control program need be aware of whether the program is run in a real or artificial environment.

If we ignore the interface of the SIDE kernel to the physical world (and assume that the environment is fully virtual), we get a simulation package oriented towards modeling communication networks at the implementation level. SIDE has many standard features of simulators, e.g., random number generators and tools for collecting performance data. As we explained in section 1.1, before it became an execution platform, SIDE (or rather its predecessor dubbed SMURPH) had been a pure simulation package. None of its simulation-oriented features have been removed in the present version; consequently, SIDE is 100% compatible with SMURPH and it can be used directly to run simulation experiments expressed in SMURPH (which come included with this package).

1.5 Overview

SIDE is a programming language for describing reactive systems (e.g., communication networks) and specifying event-driven programs (e.g., network protocols) in a natural and straightforward way. Programs in SIDE are run on a virtual "hardware" configured by the user.

SIDE can be viewed both as an implementation of a certain protocol specification language and an "executor" (kernel) for running programs expressed in that language. This kernel

can operate in three modes:

In the *real mode*, at least some elements of the program's environment are real and the program is expected to respond to actual events indicated by physical sensors, and/or trigger actions of some real actuators. In such a case, the time flow is real. Even if there exist simulated components in the environment, they must behave as real components with respect to the timing of events triggered and perceived by them.

In the *simulation mode*, there is no single real element in the environment, and the kernel operates as an event-driven discrete-time simulator. Events triggered and perceived by the control program occur in virtual time that usually has no relation to the execution time as perceived by the user.

In the *visualization mode*, the system is entirely virtual (we have strict simulation); however, the program tries to match virtual time intervals to their real time counterparts, as specified by the user. This way, the on-line behavior of the model *visualizes* the behavior of the "real thing," possibly up to a certain factor. It is possible to interact with a model run in the *visualization mode.* For example, such a model can accept external input (and produce output) in a way essentially the same as a control program interacting with external devices. One fundamental difference between the visualization mode and the real mode is that there are no restrictions in the former on what kind of events make sense. In particular, channel models are illegal (impossible) in the real mode because a channel model cannot correspond to a sensible real object directly perceptible by the control program. On the other hand, modeling channels in the visualization mode is OK because we do not have to worry about the straightforward correspondence of all aspects of the model to the world located outside the simulator.

SIDE has been programmed in C++ and its protocol specification language is an extension of C++. In this manual, we assume that the reader is familiar with C++ and the general concepts of object-oriented programming.

The structure of SIDE is presented in figure 2. A program in SIDE is first preprocessed by `smpp` to become a C++ program. The code in C++ is then compiled and linked with the SIDE run-time library. This operations are organized by `mks`—a generator script that accepts as arguments a collection of program files and options, and creates a stand-alone module, which is an executable control program or simulator for the system described by the source file and input data. If we are interested exclusively in simulation, the resultant program can be fed with some input data and run "as is" to produce performance results. SIDE is equipped with the standard features of simulators, like random number generators and tools for collecting performance data.

The program in SIDE runs under the supervision of the SIDE kernel, which coordinates the multiple threads (*processes*) of the program, schedules events, and communicates the program with the outside world. The program can be monitored on-line via DSD[5]—a Java

[5]DSD stands for *Dynamic Status Display.*

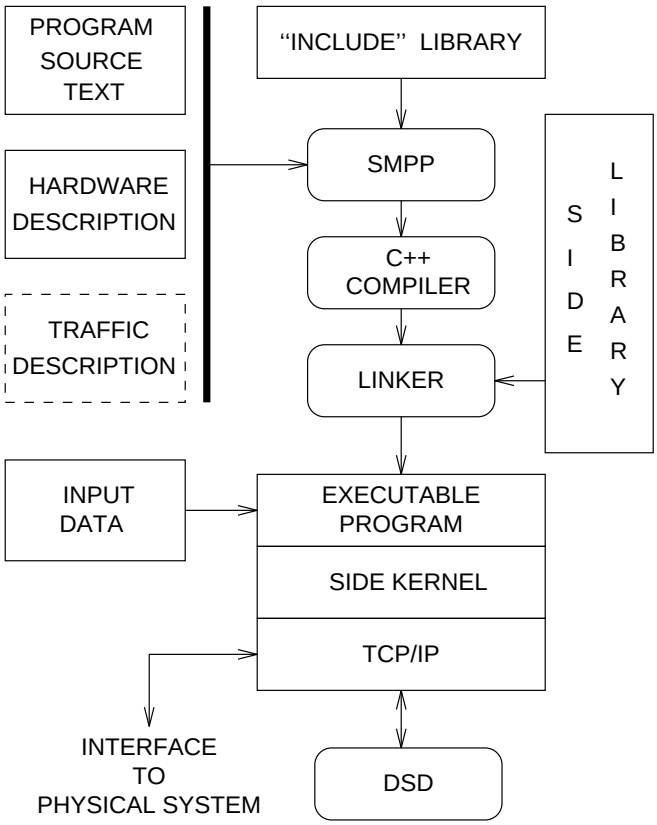

Figure 2: The structure of SIDE.

applet that can be invoked, e.g., from a web browser. This communication is carried out through a monitor server that typically runs on a dedicated host visible by both parties (i.e., the program and DSD). DSD is useful for debugging and peeking at the partial results of a potentially long simulation experiment.

Although SIDE 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 assertions describing sequences of protocol actions. In fact, they provide an alternative (static) way of specifying the protocol; it is checked in run-time whether the two specifications agree.

For the purpose of simulation, the source program in SIDE is logically divided into three parts (see figure 3). The protocol part represents the dynamics of the modeled system. The network part is a logical description of the hardware on which the protocol program is expected to run. Finally, the traffic part describes the load of the modeled system, i.e., the events arriving from the "virtual outside" and their timing.

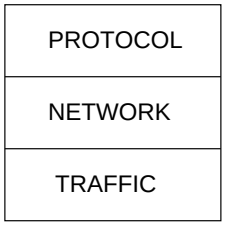

Figure 3: Components of a SIDE program.

The terms "protocol" and "traffic" reflect the fact that the primary application of SIDE's predecessor was simulating communication networks. However, it still makes sense to call a control program driving a network of sensors and actuators a *protocol*, because, owing to its reactive nature, such a program looks like a set of rules prescribing actions to be taken upon some specific events that may be coming from several different (and distant) sources. Similarly, it makes sense to talk about the input *traffic* in a (simulated) fragment of a reactive system, because such a system typically handles some objects arriving from the outside, and it is natural to represent such objects as structured *packets*.

For the purpose of developing control programs in SIDE, we may adopt a slightly more elaborate view of the source program (see figure 4). The protocol (i.e., the collection of processes) consists now of two parts: the control program proper and the simulator for the virtual components of the driven system. Similarly, the network part is split into the so-called *network map*, i.e., the mapping of logical sensors and actuators perceived by the control program onto their real (or simulated) counterparts, and the description of the modeled fragments of the underlying hardware, i.e., the hardware used by the simulator part of the protocol. The traffic specification only applies to the simulated part of the

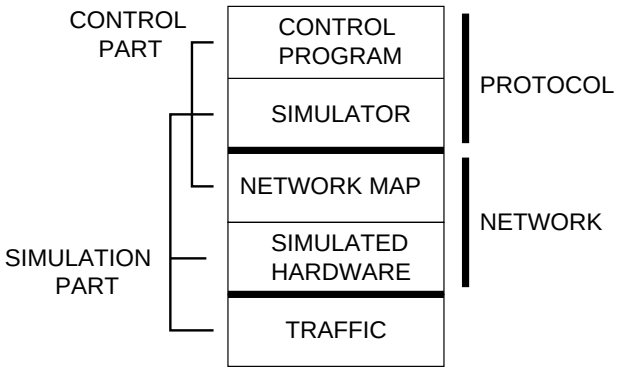

Figure 4: A control program in SIDE.

environment (real fragments handle real traffic that need not be specified).

With the above view, one can separate the software components that belong to the control system from those belonging to the simulator. Thus, the "control program" + "network map" comprise the actual control system (this part represents the target of the development process, with the network map interpreted as a parameterization of the control program), while the remaining components will tend to vanish, until they ultimately disappear altogether in the complete version of the system.

The system controlled by a SIDE program is perceived by that program as a collection of sensors and actuators represented by *mailboxes* (section 13). The actual mapping of a sensor/actuator mailbox into its real physical counterpart consists of two steps. The lower-level portion of this mapping is carried out by a *daemon* that interfaces a physical network of sensors/actuators to the Internet. The daemon acts as a server intercepting all status change events in the sensor network and transforming them into TCP/IP packets sent to the clients. Similarly, it receives status change requests from its Internet clients and transforms them into new values of actuators.

On the SIDE end, the sensor/actuator resulting from this mapping is visible as a mailbox bound to a TCP/IP port.[6] Technically, this perception is sufficient to implement a control program operating directly on such a mailbox. However, it usually makes better sense to impose one more level of mapping, i.e., to transform the raw sensor/actuator mailbox presented by the daemon into a generic and more uniform object. The second level of mapping is performed by the network map portion of the protocol program in SIDE, and its advantages are listed below.

- The control program can be built in a way independent of the technical idiosyncrasies

[6] A mailbox can also be bound to a device, i.e., a serial port. Such a mailbox directly represents the device and makes it visible to SIDE as a reactive object.

of different sensors/actuators manufactured by different vendors.

- It is possible to have a single logical sensor/actuator mapped into several real sensors/actuators and vice versa.
- It is easy to change the mapping of a logical sensor/actuator without affecting the control program. For example, the same logical sensor/actuator may be mapped differently in different versions of the control program (e.g., it may be simulated in some versions or mapped to a physical sensor/actuator in others).

2 Lexical notes: naming conventions

SIDE constitutes a layer of types, objects, operations and library functions imposed on C++. Thus, the protocol specification language accepted by the package consists of C++ augmented by some additional rules and constructs. We assume that the reader is familiar with C++ and the features of this language will not be discussed in this manual, unless they are essential for understanding some SIDE-specific notions.

The following rules were followed when introducing SIDE-specific names into the protocol specification language.

The names of global variables and user-visible object attributes not being methods start with a capital letter and may contain both lower and upper case letters. If the name of a variable has been obtained by putting together a number of words, then each word starts with a capital letter and all other letters are in lower case. Examples of such names are: `Time`, `Itu`, `s->MQHead`.

The names of functions and object methods start with a lower case letter and may contain both upper and lower case letters. If the name of a function has been obtained by putting together a number of words, each word, except the first one, starts with a capital letter and all other letters are in lower case. Examples: `setLimit(100)`, `pkt->frameSize()`.

The names of operations that are intentionally new keywords added to the language start with lower case letters and contain lower case letters only. Examples: `station A : B {`, `proceed NewState;`. The names of macros that are supposed to look as functions or methods obey the same rules as the names of functions or methods. Similarly, the names of macros providing aliases for global variables obey the same rules as the names of variables. Examples: `idToStation(i)`, `TheStation`, `Timer`.

The name of a symbolic constant (defined as a macro) starts with a sequence of capital letters optionally followed by a "_" (underscore) and a sequence of lower case letters (and possibly digits). Examples: `YES`, `MIT_exp`, `BIG_precision`, `PF_usr3`.

There exist some global variables, types, and functions that are not intended to be visible to the user. The names of such variables and types begin with "`zz_`" or "`ZZ_`". Most of the user-invisible methods are made private or protected within their objects.

3 Extensions of standard arithmetic

Having been built on top of C++, SIDE naturally inherits all the arithmetic machinery of that language. Most of the extensions to this machinery result from the need for a high precision representation of the modeled discrete time. This issue is usually less serious in control programs (where the time resolution of the order of microseconds is typically sufficient) than in simulators (where the fine granularity of time helps identify race conditions and other timing problems).

3.1 Simple integer types

Different machines may represent types `int` and `long` using different numbers of bits. Moreover, some machines/compilers may offer multiple non-standard representations for integer numbers.[7] To alleviate the portability problems that may result from different representation of the integer types on different computers, SIDE defines its private aliases for the most critical of those types. It is recommended that SIDE programs use these aliases instead of the standard types, at least in the cases when a non-trivial minimum precision must be guaranteed for a particular object. Specifically, the package defines the following types:

`Long`
: This type is intended to represent numbers with a sufficient precision to cover the range of object `Id`s (see section 5.2). Generally, type `Long` should be used for integer numbers (possibly signed) which require more than 16 but no more than 32 bits of precision. (SIDE assumes that numbers belonging to the standard type `int` are represented on at least 16 bits.) The guaranteed precision of type `Long` is 24 bits.[8]

`LONG`
: This type represents the maximum-precision signed integer numbers available on a given machine. Type `LONG` is used as the base type for creating numbers of type `BIG` (see section 3.3).

`IPointer`
: This is the smallest signed integer type capable of holding a pointer, i.e., an address.

`LPointer`
: This is the smallest type capable of holding both a pointer and a `LONG` integer number.

[7] E.g., type `long long`.
[8] Typically, type `Long` is defined as `long` and its actual precision is 32 bits.

The distinction between the above integer types becomes meaningful for architectures that offer 64-bit integer arithmetic. For example, on a 64-bit system, type `int` may be represented on 32 bits, whereas both `long` integers and pointers are 64-bit long. On such a machine, `Long` is equivalenced to `int` and the remaining three types (i.e., `LONG`, `IPointer`, and `LPointer`) are all defined as `LONG`.

On a standard 32-bit Intel system, where types `int` and `long`, as well as the pointer type, are all 32-bit long, types `Long`, `LONG`, and `LPointer` are set to `long`, and type `IPointer` is declared as `int`. With the machines/compilers that offer the type `long long`, the user may select this type as the base type for creating type `BIG` (see section 19.1). In such a case, types `LONG` and `LPointer` will be declared as `long long`, type `Long` will be equivalenced to `long`, and type `IPointer` will be defined as `int` (unless the pointer size is wider than the size of `int`). Other combinations are conceivable, depending on the subtleties of the machine architecture and the assortment of integer types offered by the compiler.

3.2 Time in SIDE

As all simulators, SIDE provides tools for modeling the flow of time in a class of physical systems. Time in SIDE is discrete, which means that integer numbers are used to represent time instants and intervals. In the light of modern physics, the discrete nature of the modeled time is not really a limitation,[9] provided that the granularity of time sampling is not too coarse.

In SIDE, the user is responsible for choosing the granularity of the modeled time. This granularity is determined by the physical interpretation of the time unit—the interval between two consecutive time instants. This interval is called the *indivisible time unit* and is denoted by *ITU*. In the virtual (simulation) mode, the correspondence between the *ITU* and an interval of real time is not defined explicitly. All time-dependent quantities are assumed to be relative to the *ITU*, and it is up to the interpreter of the input data and output results to assign a meaning to it. In the real mode, by default, one *ITU* is equal to one microsecond of real time. This binding can be redefined by the user. In the visualization mode, the user must set up the correspondence between the internal time unit and an interval of real time. This correspondence must be explicitly established when the visualization mode is entered (section 7.10).

If two events modeled by SIDE occur within the same *ITU*, there is generally no way to order them with respect to their "actual" succession in time. In many cases (especially in any of the two simulation modes) the order in which such events are presented (trigger some operations in the user program) is nondeterministic. It is possible, however, to assign priorities to awaited events. This way, when multiple events occur at the same time, they will be presented in the order of their priorities. It is also possible (see section 19.2)

[9] In fact, real time is also discrete, namely, for two events separated in time by less than the so-called Planck interval (about 10^{-45}s) the concept of chronological ordering is meaningless.

to switch off the nondeterminism in SIDE (which is generally much less desirable in control than in simulation). Of course, events triggered by actual physical components of a controlled system always occur at some specific real-time moments.

Besides the *ITU*, there exist two other time-related units, whose role is to hide the internal time unit and simplify the interpretation of input and output data. One of them is *ETU*, i.e., the *experimenter time unit*, which determines the primary unit of time being of relevance to the user. The user declares the correspondence between the *ITU* and the *ETU* by calling one of the following two functions:

```
void setEtu (double e);
void setItu (double i);
```

The argument of `setEtu` specifies how many *ITU*s are in one *ETU*; the argument of `setItu` indicates the number of *ETU*s in one *ITU*; The call

```
setEtu (a);
```

is equivalent to the call:

```
setItu (1/a);
```

One of the two functions should be called only once, at the beginning of the network creation phase, before the first object of the network is created (section 6). Two global, read-only variables `Itu` and `Etu` of type `double` tell the relationship between the *ITU* and *ETU*. `Itu` contains the number of *ETU*s in one *ITU* and `Etu`, the reciprocal of `Itu`, contains the number of *ITU*s in one *ETU*.

The third unit is called *DU*, which stands for *distance unit*. Its purpose is to establish a convenient measure of distance (typically propagation distance between network nodes), which internally translates into a specific number of *ITU*s required by the signal to travel the given distance. The function:

```
void setDu (double i);
```

sets the *DU* to the specified number of *ITU*s. The corresponding global variable (of type `double` is called `Du`, and it contains the number of *ITU*s in one unit of propagation distance.

For illustration, suppose that you want to model a mobile wireless network operating at 100Mb/s, with the mobility grain of 10cm. Assuming the signal propagation speed of 3×10^8m/s, the amount of time corresponding to 10cm is $1/(3 \times 10^9) = 0.33 \times 10^{-9}$s. You want the *ITU* to be at least as small as this number. Also, you have to keep in mind that the transmission rate of a transceiver (or port) should be expressible as an entire number

of *ITU*s per bit (see sections 6.5.1, 6.3.1). At 100Mb/s, one bit translates into 10^{-8}s; thus, it may make sense to assume that 1 *ITU* corresponds to 0.33×10^{-9}s. By issuing the following two calls:

```
setEtu (3000000000.0);
setDu (10.0);
```

you set the *ETU* to 1s and the *DU* to 1m. Also, the transmission rate of transceivers (section 6.5.1) should then be set to 30, which is the number of *ITU*s needed to insert a single bit into the wireless channel. Note that the *ITU* is never defined explicitly: its meaning is determined by the values of the remaining two units, as well as by the transmission rate assigned to ports and/or transceivers.

In the real mode, the *DU* is not used (the concept of propagation distance only applies to the models of communication channels), and the *ETU* stands for one real second. There exists an alias for `Etu`, namely `Second`; instead of `setEtu`, you can use `setSecond` with the same result. These functions give you a means to define the internal granularity of time. By default, `Second` (`Etu`) is set to 1,000,000 (and `Itu` contains 0.000001), which means that one *ITU* is equal to one microsecond.

The visualization mode brings no restrictions compared to the virtual mode; however, it makes sense to agree to interpret the *ETU* as one second. This is because the operation of entering the visualization mode (section 7.10) specifies the number of real milliseconds to be used as the synchronization grain (at those intervals the simulator will be synchronizing its virtual time to real time) and associates with it a given fraction of *ETU*. For clarity, *ETU* should correspond to some well-known interval, and one second seems to be a safe bet.

To continue the above example, suppose that you would like to visualize the behavior of your network with the granularity of 1/4s, assuming that one virtual second spreads over ten seconds of real time. This is the operation to enter the visualization mode:

```
setResync (250, 0.025);
```

The first argument gives 250 millisecond as the "resync granularity," and the second one maps these 250 milliseconds (of real time) to 0.025 *ETU*s. Thus, assuming that 1 *ETU* stands for 1 second of virtual time, one second of real time will correspond to 1/10 of a second in the model. This means that the visualization will be carried out in slow motion with the factor of 10.

Note: The functions introduced above are not available when the simulator (or rather the control program) has been compiled with `-F` (section 19.2). In such a case, the *ETU* is "hardwired" at 1,000,000 *ITU*s (`Etu`, `Itu`, and `Second` are not available to the program).

In both simulation modes, `Itu`, `Etu`, and `Du` all start equal to 1.0, and there is no implicit interpretation of the internal time unit with respect to real time. It is illegal to redefine the

ETU after the visualization mode has been entered. Note, however, that the visualization mode can be exited and entered dynamically (section 7.10).

If the same program is supposed to run in a simulation mode as well as in the control mode, it usually makes sense to put the statement:

```
setEtu (1000000.0);
```

somewhere in the initialization sequence (section 7.9).

While operating in the real or visualization mode, a SIDE program maps intervals of virtual time to the corresponding intervals of real time. The actual knowledge of real time (the precise date and time of the day) is usually less relevant. One place where actual time stamps are used is journaling (section 13.5), which is available in the real as well as the visualization mode. Such time stamps are represented as the so-called UNIX time, i.e., the number of seconds and microseconds that have elapsed since 00:00:00 on January 1, 1970. The following function:

```
Long getEffectiveTimeOfDay (Long *usec = NULL);
```

returns the number of seconds in UNIX time. The optional argument is a pointer to a `Long` number which, if specified, will contain the number of microseconds after the last second. Function `tDate` (section 4.8) converts the value returned by `getEffectiveTimeOfDay` into a textual representation of the current date and time.

It is possible to call a SIDE program operating in the real or visualization mode in such a way that its real time is shifted to the past or to the future. This may be useful together with journaling (section 13.5), e.g., to synchronize the program to a past journal file. The `-D` call argument (section 19.3) can be used for this purpose. If the real time has been shifted, both `getEffectiveTimeOfDay` and `tDate` return the shifted time.

3.3 Type `BIG` and its range

In many cases the standard range of integer numbers available on the given machine is insufficient to represent long intervals of simulated time with a satisfying granularity. Therefore, SIDE comes with its own type for representing potentially huge **nonnegative** integer numbers and performing standard arithmetic operations on them.

Time instants and other potentially huge nonnegative integers should be declared as objects of type `BIG` or `TIME`. These two types are absolutely equivalent. Intentionally, `TIME` should be used in declarations of variables representing time instants or intervals and `BIG` can be used to declare other big integers, not necessarily related to time.

When a SIDE program is created by `mks` (see section 19.2), the user may specify the precision of type `BIG` with the `-b` option. The value of this option (a single decimal digit)

is passed to SIDE (and also to the user program) in symbolic constant `BIG_precision`. This value selects the implementation of type `BIG` by specifying the minimum guaranteed precision of `BIG` numbers as `BIG_precision`×31 bits. If `BIG_precision` is 1 or 0, type `BIG` is equivalent to the maximum-length integer type available on the given machine (type `LONG`—see section 3.1) and all operations on objects of this type are performed directly as operations on integers of type `LONG`. Otherwise, depending on the size of type `LONG`, several objects of this type may be put together to form a single `BIG` number. For example, if the size of `LONG` is 32 bits, a single `BIG` number is represented as `BIG_precision LONG` numbers. If the `LONG` size is 64 bits, the number of `LONG` objects needed to form a single `BIG` number is equal to[10]

$$\lfloor \frac{\texttt{BIG\_precision} + 1}{2} \rfloor \quad .$$

The tacit limit on the precision of type `BIG` imposed by `mks` (section 19.2) is 9 which corresponds to 279 significant bits. Using this precision one could simulate the life of the universe with the granularity of time corresponding to the Planck interval.

The default precision of type `BIG` is 2 (in both modes). For the real mode, assuming that one *ITU* is equal to one microsecond, the amount of real time covered by the range of type `BIG` (and `TIME`) is almost 150,000 years (but it is only slightly more that 30 minutes for precision 1).

3.4 Arithmetic operations and conversions

The actual implementation of type `BIG` is completely transparent to the user. When the precision of `BIG` numbers is different from 1, the standard arithmetic operators `+`, `-`, `*`, `/`, `%`, `++`, `--`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `+=`, `-=`, `*=`, `/=`, `%=` are automatically overloaded to operate on objects of type `BIG`. Combinations of `BIG` operands with types `int`, `long`, and `double` are legal. If `BIG` is equivalent to `long`, the conversion rules for operations involving mixed operand types are defined by C++. Otherwise, if an operation involves a `BIG` operand mixed with a numerical operand of another type, the following rules are used:

- If the type of the other operand is `char`, `int`, or `long`, the operand is converted to `BIG` and the result type is `BIG`. One exception is the modulo (`%`) operator with the second operand being of one of the above types. In such a case, the second operand is converted to `long` and the result type is `long`.

- If the type of the other operand is `float` or `double`, the `BIG` operand is converted to `double` and the result type is `double`.

[10] Thus, for `BIG_precision` equal 1 and 2, `BIG` numbers will be represented in the same way, using the standard integer type `long`.

- An assignment to/from `BIG` from/to other numerical type is legal and involves the proper conversion. A `float` or `double` number assigned to `BIG` is truncated to its integer part.

Overflow and error conditions for operations involving `BIG` operands are only checked for precisions 2–9.[11] You can switch off those checks (e.g., for efficiency) with the `-m` option of `mks` (see section 19.2). Note that in principle it is illegal to assign a negative value to a `BIG` variable. However, for precisions 0 and 1 such operation passes unnoticed and the problem will surface later.

It is possible to check whether a `BIG` number is convertible to `double`. The following function:

```
int convertible (BIG a);
```

returns 1, if the `BIG` number `a` can be safely converted to `double`, and 0 otherwise.

Explicit operations exist for converting a character string (an unsigned sequence of decimal digits) into a `BIG` number and vice versa. The following function turns a `BIG` number into a sequence of characters:

```
char *btoa (BIG a, char *s = NULL, int nc = 15);
```

where `a` is the number to be converted to characters, `s` points to an optional character buffer to contain the result, and `nc` specifies the length of the resulting string. If `s` is not specified (or `NULL`) an internal (static) buffer is used. In any case, the pointer to the encoded string is returned as the function value.

If the number size exceeds the specified string capacity (15 is the default), the number is encoded in the format: `dd...ddEdd`, where `d` stands for a decimal digit and the part starting with `E` is a decimal exponent.

The following function can be used to convert a string of characters into a `BIG` number:

```
BIG atob (const char *s);
```

The character string pointed to by `s` is processed until the first character that is not a decimal digit is encountered. Initial spaces and an optional plus sign are ignored.

3.5 Constants

One way to create a constant of type `BIG` is to convert a character string into a `BIG` object. If the constant is not too big, one can use a conversion from `long int`, e.g., `b = 22987`

[11] If the size of type `LONG` is 64 bits, the overflow and error conditions are only checked for precisions higher than 2.

(we assume that `b` is of type `BIG`) or `double`. This works also in declarations; thus, the following examples are legal:

```
BIG  b = 12;
TIME tc = 10e9;
const TIME cnst = 20000000000.5;
```

are legal. In the last case, the fractional part is ignored.

Three `BIG` constants: `BIG_0`, `BIG_1`, and `BIG_inf` are available directly. The first two of them represent 0 and 1 of type `BIG`, the last constant stands for an *infinite* or *undefined* value and is equal to the maximum `BIG` number representable with the given precision. This maximum value is reserved to denote *infinity* (or an *undefined* value) and it should not be used as a regular `BIG` number. For completeness, there exist constants `TIME_0`, `TIME_1`, and `TIME_inf` which are exactly equivalent to the three `BIG` constants.

These two predicates tell whether a `BIG` number is *defined* (or *finite*):

```
int def (BIG a);
```

returning 1 when `a` is defined (or finite) and 0 otherwise, and

```
int undef (BIG a);
```

which is a simple negation of `def`.

The following arithmetic-related symbolic constants are available to the protocol program:

`TYPE_long`
: equal to 0. This is a type indicator (see section 3.6).

`TYPE_short`
: equal to 1. An unused type indicator existing for completeness.

`TYPE_BIG`
: equal to 2. This is the `BIG` type indicator (see section 3.6).

`BIG_precision`
: telling the the precision of `BIG` numbers (section 3.3).

`MAX_long`
: equal to the maximum positive number representable with type `long int`. For 32-bit 2's complement arithmetic this number is 2147483647.

`MIN_long`
: equal to the minimum negative number representable with type `long int`. For 32-bit 2's complement arithmetic this number is −2147483648.

`MAX_short`
: equal to the maximum positive number representable with type `short int`. For 16-bit 2's complement arithmetic this number is 32767.

`MIN_short`
: equal to the minimum negative number representable with type `short int`. For 16-bit 2's complement arithmetic this number is −32768.

`MAX_int`
: equal to the maximum positive number representable with type `int`. In most cases this number is equal to `MAX_long` or `MAX_short`.

`MIN_int`
: equal to the minimum negative number representable with type `int`. In most cases this number is equal to `MIN_long` or `MIN_short`.

There also exist the following constants: `MAX_Long`, `MIN_Long`, `MAX_LONG`, and `MIN_LONG` which represent the boundary values of types `Long` and `LONG`, according to the way these types have been aliased to the corresponding basic integer types (see section 3.1).

3.6 Other non-standard numerical types

Some numbers, not necessarily representing time instants or intervals, may also be potentially big. On the other hand, using type `BIG` to represent all numbers that could potentially exceed the capacity of standard types `int` or `long` would be too costly in typical situations. Therefore, there exist three non-standard numerical types that can be used to represent nonnegative integer variables that typically fit into the `long int` range, but sometimes one would prefer to store them as `BIG` numbers. Each of these types is an alias (`typedef`) for either `LONG` or `BIG`, depending on the setting of some options of `mks` (see section 19.2). The three "flexible" types are listed below.

`DISTANCE`
: This type is used to represent (internally) propagation distances between *ports* or *transceivers* (see sections 6.2.1, 6.3.2, 6.5.2). A propagation distance is in fact a time interval; however, in most cases this interval is reasonably small and there is no need to use type `BIG` for its representation.

`BITCOUNT`
: This type is used to declare variables counting individual information bits, e.g., transmitted or received globally or at a specific station. Numerous such counters are used internally by SIDE for calculating various performance measures.

`RATE`
: This type is used to represent port *transmission rates* (see section 6.3.1). The transmission rate of a port is a time interval which in most cases is rather small and therefore `BIG` representation would be too costly for it.

Five macros are associated with each of the above three types. For example, the following macros are related to type `DISTANCE`:

`TYPE_DISTANCE`
: This is a symbolic constant which can have one of two values: `TYPE_long` (0) meaning that type `DISTANCE` is equivalent to `LONG`, or `TYPE_BIG` (2) which means that `DISTANCE` is equivalent to `BIG`.

`MAX_DISTANCE`
: This macro is defined as `BIG_inf` if type `DISTANCE` is equivalent to `BIG` or as `MAX_LONG` (equal to the maximum `LONG` number representable on the machine), otherwise.

`DISTANCE_inf`
: The same as `MAX_DISTANCE`.

`DISTANCE_0`
: This macro is defined as `0` if type `DISTANCE` is equivalent to `LONG`, or as `BIG_0`, otherwise.

`DISTANCE_1`
: This macro is defined as `1` if type `DISTANCE` is equivalent to `LONG`, or as `BIG_1`, otherwise.

To obtain the corresponding macros for the other two types, replace the word `DISTANCE` with `BITCOUNT` or `RATE`.

By default, all three types `DISTANCE`, `BITCOUNT`, `RATE` are equivalent to `LONG`. You should use your judgment, based on the interpretation of *ITU* and the expected range of values accumulated in bit counters (like the total length of all messages received by a node in the

course of a simulation run) to decide whether the default definitions are fine. For example, if the *ITU* corresponds to 1fs (femto-second, i.e., 10^{-15}s), the maximum propagation distance covered by a 32-bit `LONG` integer is slightly over 600m (assuming $c = 3 \times 10^8$m/s). Thus, if longer distances are not unlikely to show up in your model (and `LONG` happens to be 32-bit long), `DISTANCE` should be declared as `BIG`. Type `RATE` is generally less prone to this type of problems. As a rate determines the number of *ITU*s needed to transmit a single bit, the lower the actual transmission rate, the higher the value that must be stored in a `RATE` object. For illustration, with 1 *ITU* corresponding to 1fs, the maximum interval stored in a 32-bit number is ca. 2^{-6}s, which translates into the minimum rate of 500kb/s.

Note: Rates, bit counts, and distances are only useful when SIDE is used as a network simulator (possibly in the visualization mode). They are irrelevant in the real mode.

The way distances are specified (and usually interpreted) by the user (e.g., when configuring a network, see section 6.3.3) involves `double` numbers, which are internally transformed into type `DISTANCE` according to the setting of the *distance unit DU* (section 3.2). SIDE defines the following `double` constants: `Distance_0`, `Distance_1`, `Distance_inf`, which translate into/from the respective `DISTANCE` values in the conversion from/to `double` involving the assumed distance unit. Similarly, as the internal time in *ITU*s is often converted to double precision time in *ETU*s, the following `double` constants: `Time_0`, `Time_1`, `Time_inf` represent *ETU* equivalents of the respective `TIME` constants. Even though the conversion between *ITU*, *ETU* and *DU* is straightforward (and involves multiplication/division by `Itu`, `Etu` or `Du`), SIDE provides the following functions whose consistent usage may help to avoid confusion or ambiguity:

```
double ituToEtu (TIME);
TIME etuToItu (double);
double itoToDu (TIME);
TIME duToItu (double);
```

In addition to performing the required scaling, these functions also correctly convert the special value *inf*, e.g., `ituToEtu (TIME_inf) == Time_inf`.

4 Auxiliary operations and functions

In this section we list those functions provided by SIDE that are only superficially related to its operation as a network simulator or a system controller. They cover random number generation, non-standard i/o, and a few other items that are not extremely important, as the user cold easily substitute for them some standard tools from the C++ library.

In those early days when the first (C++) version of SIDE (SMURPH) was being developed, there was little consensus regarding the exact shape of the C++ i/o library and

other common functions, which tended to differ quite drastically across platforms. Consequently, we wanted to cover them as much as possible by our own tools, such that the same SIDE program could compile and run without modifications on all popular platforms. These tools are still present in the most recent version of our package, although these days the standard libraries are incomparably more consistent.

4.1 Random number generators

All random number generators offered by SIDE are based on the following function:

```
double rnd (int seed);
```

which returns a pseudo-random number of type `double` uniformly distributed in [0,1). By default, this function uses its private congruential algorithm for generating random numbers. The user can select the standard random number generator (the `drand48` family) by creating the program with the "`-8`" option (see section 19.2).

The argument of `rnd` which must be `SEED_traffic` (0), `SEED_delay` (1), or `SEED_toss` (2), identifies one of three *seeds*. Each seed represents a separate pattern of pseudo-random numbers. The initial values of the seeds (which are of type `long`) can be specified when the SIDE program is called (see section 19.3); otherwise, some default (always the same) values are assumed. This way simulation runs are replicable, which is important from the viewpoint of debugging.

Intentionally, each of the three seeds represents one category of objects to be randomized, in the following way:

`SEED_traffic`
: Traffic related randomized values (see section 10), e.g., message interarrival intervals, message lengths, selection of the transmitter and the receiver.

`SEED_delay`
: Values representing lengths of various (possibly randomized) delays not related to traffic generation. For example, randomization of `Timer` delays for modeling inaccurate clocks (section 8.3) is based on this seed.

`SEED_toss`
: Tossing (multi-sided) coins in situations when SIDE must decide which of a number of equally probable possibilities to follow, e.g., selecting one of two or more waking events scheduled at the same *ITU* (see section 7.4).

All random numbers needed by SIDE internally are generated according to the above rules, but the user is not obliged to obey them. In most cases, the user need not use `rnd` directly.

When the program is restarted with the same value of a given seed, all randomized objects belonging to the category represented by this seed will be generated in exactly the same way as previously. In particular, two simulation runs with identical values of all three seeds will produce exactly the same results, unless the declared CPU time limit is exceeded (section 15.3).

The following two functions can be used to generate exponentially distributed pseudo-random numbers:

```
TIME tRndPoisson (double mean);
LONG lRndPoisson (double mean);
```

When the precision of `TIME` is 1, the two functions are identical. The first function should be used for generating objects of type `TIME`, whereas the second one generates (`LONG`) integer values. The parameter specifies the mean value of the distribution. In both cases the result is generated from a uniformly distributed pseudo-random value obtained by a call to `rnd` with seed 0 (`SEED_traffic`). The result is a double precision floating point number which is rounded to an object of type `TIME` or `LONG`.

Below we present four functions that generate uniformly distributed random numbers of type `TIME` and `LONG`.

```
TIME tRndUniform (TIME min, TIME max);
TIME tRndUniform (double min, double max);
LONG lRndUniform (LONG min, LONG max);
LONG lRndUniform (double min, double max);
```

In all cases the result is between `min` and and `max`, inclusively. The functions call `rnd` with seed 0 (`SEED_traffic`).

Sometimes one would like to randomize a certain, apparently constant, parameter, so that its actual value is taken with some *tolerance*. The following functions serve this end:

```
TIME tRndTolerance (TIME min, TIME max, int q)
TIME tRndTolerance (double min, double max, int q)
LONG lRndTolerance (LONG min, LONG max, int q)
LONG lRndTolerance (double min, double max, int q)
```

The above functions generate random numbers according to the *Beta* distribution which is believed to describe technical parameters that may vary within some tolerance. The functions call `rnd` with seed 1 (`SEED_delay`) and transform the uniform distribution into distribution

$$\beta(q, q)$$

which is scaled appropriately, such that the resultant random number is between `min` and `max` inclusively. The parameter `q`, which must be greater than 0, can be viewed as the

"quality" of the distribution. For higher values of `q` the generated numbers have better chances to be closer to (`min`+`max`)/2. Reasonable values of `q` are between 1 and 10. Slightly more time is taken for generating a random number for a higher value of `q`.

The following function generates a normally distributed (Gaussian) random number:

```
double dRndGauss (double mean, double sigma);
```

with the specified mean and standard deviation. It calls `rnd` with seed 2 (`SEED_toss`).

Here is a function to generate a randomized number of successes in a Bernoulli experiment involving tossing a possibly biased coin:

```
Long lRndBinomial (double p, Long n);
```

where `p` is the probability of success in a single experiment, and `n` is the total number of trials.

Various instances of practical Zipf-style distributions can be produced with this function:

```
Long lRndZipf (double q, Long max = MAX_Long, Long v = 1);
```

which generates integer random values between 0 and `max`–1, inclusively, in such a way that the probability of value k being selected is proportional to:

$$P_k = \frac{1}{(v+k)^q} \quad (k = 0, \ldots, \texttt{max} - 1)$$

Note that `v` must be greater than zero, and `q` must be greater than 1.

The following function simulates tossing a multi-sided coin:

```
Long toss (Long n);
```

It generates an integer number between 0 and `n`–1 inclusively. Each number from this range occurs with probability 1/`n`. The function calls `rnd` with seed 2 (`SEED_toss`).

There is an abbreviation for the most frequently used variant of `toss`, namely, the following function:

```
int flip ();
```

returns 0 or 1, each value with probability 0.5. This function is slightly more efficient than `toss (1)`.

4.2 Input/Output

Two standard files, dubbed the *data* file and the *results* file, are automatically opened by a SIDE program at the beginning of its execution. For a simulation experiment, the results file is supposed to contain the final performance results at the end of run. In a control session, the results file may not be needed, although it may still make sense to produce some "permanent" output, e.g., a log or some measurement data.

The *data* file, represented by the global variable `Inf` of type `istream`, is opened for reading and it may contain some numerical data parameterizing the system architecture and the protocols, e.g., the number of stations, TCP/IP ports of the daemons, the mapping of virtual sensors/actuators to their physical counterparts, etc.

Note: The data file is only accessible during the initialization phase, i.e., while the `Root` process is in its first state (section 7.9). The file is closed before the protocol execution is started. If the program requires a continuous stream of data during its execution, it should open a non-standard input file with that data, or, perhaps, read the data from a bound mailbox (if it executes in the real or visualization mode—section 13.1). The same applies to non-standard output files, which, needless to say, a program may open as many as it needs.

4.2.1 Reading and writing simple numerical data

All standard functions and methods of C++ related to input/output are available from SIDE. There are, however, some additional functions introduced specifically by SIDE, which handle input from the data file and output to the results file. The most natural way to write some information to the *results* file is to *expose* some objects (section 17). In this section we discuss more elementary i/o operations offered by SIDE.

The standard C++ operators `<<` and `>>` with a stream object as the first argument have been overloaded to handle `BIG` numbers. Thus, it is legal to write

```
sp >> b;
```

or

```
sp << b;
```

where `sp` is a stream object (it should be an `istream` in the first case and an `ostream` in the second) and `b` is a number of type `BIG` (or `TIME`). In the first case, a `BIG` number is read from the stream and stored in `b`. The expected syntax of that number is the same as for `atob` (section 3.4), i.e., a sequence of decimal digits optionally preceded by white spaces. It is also legal to put a "+" sign immediately before the first digit. Note the there are no negative `BIG` numbers. The `<<` operation encodes and writes a `BIG` number to the

stream. The number is encoded by a call to `btoa` (section 3.4) with the third argument (digit count) equal to 15. If the number has less than 15 digits, the initial spaces are stripped off.

The following functions can be used to read numbers from the (standard) data file:

```
void readIn (int&);
void readIn (Long&);
void readIn (LONG&);
void readIn (BIG&);
void readIn (float&);
void readIn (double&);
```

Each of the above functions ignores in the data file everything that cannot be interpreted as the beginning of a number. A number expected by any of the above input functions can be either integer (the first three functions) or real (the last two functions). An **integer number** begins with a digit or a sign (note that the minus sign is illegal for a `BIG` number) and continues as long as the subsequent characters are digits. A sign not followed by a digit is not interpreted as the beginning of a number. A real number may additionally contain a decimal point followed by a sequence of digits (the fraction) and/or an exponent. The syntax is essentially as that accepted by the standard UNIX parsing tools, e.g., `strtod`. A decimal point encountered in an expected integer number stops the interpretation of that number, i.e., the next number will be looked for starting from the first character following the decimal point.

The size of an integer number depends on the size of the object being read. In particular, the range of variables of type `BIG` (`TIME`) may be very big, depending on the declared precision of this type. All values corresponding to time intervals must not be negative.

There are three simple features that help organize complex data files in a more legible way. If a number read by one of the above functions is immediately followed by a sign (‘`+`’, ‘`-`’) and another number, then the two numbers are combined into one. For example, 120+90 will be read as 210; similarly 1–0.5 will be read as 0.5, if a real number is expected, or as 1 (1–0) otherwise. The rule applies iteratively to the result and thus 1+2+3+4–5 represents a single number (5). Another feature is the symbolic access to the last-read number. Namely, the character ‘`%`’ appearing as the first (or as the only) item of a sequence of numbers separated by signs stands for the “last-read value.” For example, 15, %–4 will be read as 15, 11. If the type of the expected number is different from the type of the last-read number, the value of ‘`%`’ is undefined. The last feature provides an abbreviation for multiple consecutive occurrences of the same number. Namely, if a number m is immediately followed by a slash (‘/’), followed in turn by an unsigned integer number n, the entire sequence is interpreted as n occurrences of number m. Again all the expected numbers should be of the same type, otherwise the results are unpredictable.

The fact that only numerical data (together with a few other characters) are relevant,

and everything else is skipped, makes it easy to include comments in the input data file. It is also possible to put a number in a comment. Namely, whenever an asterisk ('*') is encountered in the *data* file, the rest of the current line is ignored, even if it contains some numbers.

Here is one more function useful for scanning character strings that may contain numbers:

```
int parseNumbers (const char *str, int n, nparse_t *res);
```

where `nparse_t` is a structure declared as follows:

```
typedef struct {
   int type;
   union {
     double  DVal;
     LONG    LVal;
     int     IVal;
   };
} nparse_t;
```

The second argument, `n`, gives the expected number count, which determines the size of the `res` array (whose entries are records of type `nparse_t`). The function will try to fill up to `n` entries in `res` with subsequent numbers located in the input string `str`. It will skip all useless characters in front of a number, and will stop interpreting the number as soon as it cannot be correctly continued. The exact format of an expected number depends on `type`, which should be preset to one of the following values:

`TYPE_double`
: The expected number is `double`. When found, it will be stored in `DVal`.

`TYPE_LONG`
: The expected number is `LONG`. When found, it will be stored in `LVal`. If the number begins with `0x` or `0X`, possibly preceded by a sign, it will be decoded as hexadecimal.

`TYPE_int`
: The expected number is `int`. When found, it will be stored in `IVal`. If the number begins with `0x` or `0X`, possibly preceded by a sign, it will be decoded as hexadecimal.

`TYPE_hex`
: The expected number is an integer-sized value coded in hex, regardless of whether it begins with `0x` or `0X`. When found, it will be stored in `IVal`.

`ANY`
: If the actual number is `double`, but not `LONG`, i.e., it has a decimal point or/and an exponent, it will be stored in `DVal`, and `type` will be set to `TYPE_double`. Otherwise, the number will be decoded as `LONG` and stored in `LVal`, while `type` will be set to `TYPE_LONG`. In the latter case, the number can also have the hex prefix (`0x` or `0X`) possibly preceded by a sign.

The function value tells how many numbers have been located in the string. This can be more than `n`, which means that some numbers were ignored because the `res` array was too short. If any of the numbers encountered by the function overflows the limitations of its format, the function returns `ERROR` (-127).

The following functions can be used to write simple data items to the *results* file:

```
void print (LONG n, const char *h = NULL, int ns = 15, int hs = 0);
void print (double n, const char *h = NULL, int ns = 15, int hs = 0);
void print (BIG n, const char *h = NULL, int ns = 15, int hs = 0);
void print (const char *n, int ns = 0);
void print (LONG n, int ns);
void print (double n, int ns);
void print (BIG n, int ns);
```

The first argument identifies the data item to be written: it can be a number or a character string. In the most general case (the first three functions), there are three more arguments denoting:

`h`
: A textual header to precede the data item. If this argument is `NULL`, no header is printed.

`ns`
: The number of character positions taken by the encoded data item. If the value of this argument is greater than the actual number of positions required, the encoded item is right-justified and spaces are inserted on the left.

`hs`
: The number of character positions taken by the header. The header is left-justified and the appropriate number of spaces are added on the right. The total length of the encoded item (together with the header) is `ns`+`hs`.

The last three functions are abbreviations of the first three ones with the second argument equal `NULL` and `hs` equal 0.

4.2.2 The XML parser

To facilitate interfacing the simulator to external programs that may act as the suppliers of simulation data and, possibly, absorbers of the output, SIDE has been equipped with a simple XML parser.[12] Being a rather independent add-on, the parser is a self-contained feature, which can be used independently of its intended role as an input/output helper.

The document structure is represented as a tree of nodes, with every node pointing to a complete XML element (tag) of the document, which may include subtrees, i.e., subordinate tags. The simple idea is to store the entire document in memory for processing. Type `sxml_t` describes a node (tag) pointer. Here is the complete list of functions available from the parser:

`sxml_t sxml_parse_str (char *s, size_t len);`
: Given a string `s` of length `len` representing an XML document, the function transforms it into a tree of nodes and returns a pointer to the root node. For economy, the input string is modified and recycled; this is why it isn't passed as `const`. The tree is stored in dynamically allocated memory, which can be freed later by `sxml_free` (see below).

`int sxml_ok (sxml_t node);`
: The function should be called after a conversion by `sxml_parse_str` to tell whether the operation has succeeded (the function returns `YES`) or failed (the function returns `NO`). The argument should point to the root node of the XML tree as returned by `sxml_parse_str`.

`const char *sxml_error (sxml_t node);`
: If `sxml_ok` returns `NO` on the root node produced by `sxml_parse_str`, this function (invoked on the same node) will return a NULL-terminated error description string. The function returns an empty string (not the NULL pointer) if there was no error.

`char *sxml_txt (sxml_t node);`
: This function returns the character string representing the textual content of the node. For example, if `node` describes a tag that looks like this:

```
<tolerance quality="3" dist="u">0.000001</tolerance>
```

the function will return the NULL-terminated string "`0.000001`".

[12] Adapted from the publicly available (MIT license) code by Aaron Voisine.

`char *sxml_attr (sxml_t node, const char *attr);`
: This function returns the value of the attribute identified with `attr` and associated with the tag pointed to by `node`. For example, if `node` points to the "tolerance" tag from the previous illustration, `sxml_attr (node, "quality")` will return the string "`3`". Note that there is a difference between a non-existent attribute and an empty one. In the former case, the function returns `NULL`, while in the latter case it returns an empty string.

`sxml_t sxml_child (sxml_t node, const char *name);`
: This function returns the first child tag of `node` (one level deeper) with the given name, or `NULL` if no such tag is found. For example, if `node` points to the following tag:

```
<goodies>This is a list of goodies:
    <goodie>First</goodie>
    <other>Not a goodie</other>
    <goodie>Second</goodie>
    <goodie>Third</goodie>
</goodies>
```

  `sxml_child (node, "goodie")` returns a pointer to the tree node representing `<goodie>First</goodie>`.

`sxml_t sxml_next (sxml_t node);`
: This function returns a pointer to the next tag (at the same level) with the same name as the one pointed to by `node`, or `NULL` if there are no more tags. For example, if `node` points to the first "goodie" tag in the above document, `sxml_next (node)` will return a pointer to the second "goodie" (not to the "other" tag). Note that to get hold of "other," you have to call `sxml_child (node, "other")` on the "goodies" node.

`sxml_t sxml_idx (sxml_t node, int idx);`
: This function returns the `idx`'th tag (counting from zero) with the same name as the tag pointed to by `node` and at the same level. In particular, for `idx` = 0, the function returns `node`. If `idx` is larger than the number of tags with the same name following the one pointed to by `node`, the function returns `NULL`. For example, `sxml_idx (node, 2)` with `node` pointing to the first "goodie" in the above example, returns a pointer to the third "goodie."

`char *sxml_name (sxml_t node);`
: This function returns the name of the tag pointed to by `node`. For example, with `node` pointing to the "other" tag in the above example, `sxml_name (node)` returns the string "`other`".

`sxml_t sxml_get (sxml_t node, ...);`
: This function traverses the tree of nodes rooted at `node` to retrieve a specific subitem. It takes a variable-length list of tag names and indexes, which must be terminated by either an index of −1 or an empty tag name. For example,

```
title = sxml_get (library, "shelf", 0, "book", 2, "title", -1);
```

retrieves the title of the 3-rd `book` on the 1-st `shelf` of `library`. The function returns `NULL` if the indicated tag is not present in the tree.

`const char **sxml_pi (sxml_t node, const char *target);`
: This function returns an array of strings (terminated by NULL) consisting of the XML *processing instructions* for the indicated target (entries starting with "`<?`"). Processing instructions are not part of the document character data, but they are available to the application.

The above functions extract components from an existing XML structure, e.g., built from a parsed input string. It is also possible to create (or modify) an XML structure and then transform it into a string, e.g., to be printed to the output file. This can be accomplished by calling functions from the following list. Some of these functions exist in two variants distinguished by the presence or absence of the trailing "`_d`" in the function name. The variant with "`_d`" copies the string argument into new memory, while the other variant uses a pointer to the original string, thus assuming that the string will not be deallocated while the XML tree is being used.

`sxml_t sxml_new[_d] (const char *name);`
: This function initializes a new XML tree and assigns the specified name to its root tag. The tag has no attributes and its text is empty.

`sxml_t sxml_add_child[_d] (sxml_t node, const char *name, size_t off);`
: The function adds a child tag to `node` and assigns `name` to it. The `off` argument specifies the offset into the parent tag's character content. The returned value points to the child node. The offset argument can be used to order the children, which will appear in the resulting XML string in the increasing order of offsets. This will also happen if the character content of the parent is shorter than the specified offset(s).

`sxml_t sxml_set_txt[_d] (sxml_t node, const char *txt);`
: This function sets the character content of the tag pointed to by `node` and returns `node`. If the tag already has a character content, the old string is overwritten by the new one.

`sxml_t sxml_set_attr[_d] (sxml_t node, const char *name, const char *value);`
The function resets the tag attribute identified by `name` to `value` (if already set) or adds the attribute with the specified value. A value of `NULL` will remove the specified attribute. Note the difference between attribute removal and setting it to an empty string. The function returns `node`.

`sxml_t sxml_insert (sxml_t node, sxml_t dest, size_t off);`
This function inserts an existing tag `node` as a child in `dest` with `off` used as the offset into the destination tag's character content (see `sxml_add_child`).

`sxml_t sxml_cut (sxml_t node);`
This function removes the specified tag along with its subtags from the XML tree, but does not free its memory, such that the tag (whose pointer is returned by the function) is available for subsequent operations.

`sxml_t sxml_move (sxml_t node, sxml_t dest, size_t off);`
This is a combination of `sxml_cut` and `sxml_insert`. First the tag (`node`) is removed from its present location and then inserted as for `sxml_insert`.

`void sxml_remove (sxml_t node);`
As `sxml_cut`, but the memory occupied by the tag is freed.

`char *sxml_toxml (sxml_t node);`
The function transforms the XML tree pointed to by `node` into a character string ready to be printed.

These two operations complete the set:

`void sxml_free (sxml_t node);`
The function deallocates all memory occupied by the XML tree rooted at `node`. It only makes sense when applied to the root of a complete (stand-alone) XML tree.

`sxml_t sxml_parse_input (char del = '\0', char **data = NULL);`
The function reads the standard data file (stream `Inf`—section 4.2) and treats it as an XML string to be parsed and transformed into a tree. It returns the root node of the tree, which should be checked for successful conversion with `sxml_ok` and/or `sxml_error` (see above). If the first argument is not `'\0'`, its occurrence as the first character of a line will terminate the interpretation of the input file. If the second argument is not `NULL`, then the (null-terminated)

string representing the complete data file (before XML parsing) will be stored at the pointer passed in the argument. The string will be complete in the sense that the contents of any *included* files (see below) will have been inserted in their respective places, i.e., there will be no "include" tags in the string.

The last function is the only one that interprets *includes* in the parsed file, i.e., tags of these forms:

```
<xi:include href="filename">...</xi:include>
<xi:include href="filename"/>
```

where *filename* gives the name of the file whose contents are to be inserted into the current place of the processed XML data, replacing the entire `<xi:include>` tag. The tag may have a text content, which will be ignored, but is not allowed to have sub-tags. This feature implements a trivial subset of the full capabilities of the official XML `<xi:include>` tag, in particular, `href` is the only expected and parsed attribute. The included file is interpreted as XML data (being a straightforward continuation of the source file), which is allowed to contain other `<xi:include>` tags. In the SIDE variant, the tag's keyword can be abbreviated as `include`, e.g., these two constructs:

```
<xi:include href="somefile.xml"/>
<include href="somefile.xml"></include>
```

are equivalent.

4.3 Operations on flags

In a number of situations, it is desirable to set, clear, or examine the contents of a single bit (flag) in a bit pattern. SIDE defines the type `FLAGS`, which is equal to `Long` or `LONG`,[13] and is to be used for representing 32-element flag patterns. In particular, a collection of flags is associated with a packet (section 10.2). The following simple functions[14] provide elementary operations on binary flags:

```
FLAGS setFlag (FLAGS flags, int n);
FLAGS clearFlag (FLAGS flags, int n);
int flagSet (FLAGS flags, int n);
int flagCleared (FLAGS flags, int n);
```

The first two functions respectively set and clear the contents of the n-th bit in `flags`. The updated value of `flags` is returned as the function value. Bits are numbered from 0 to 31, 0 is the number of the rightmost (least significant) bit.

[13] The guaranteed precision of type `FLAGS` is 32 bits.

[14] These functions are in fact macros.

The last two functions examine the contents of the n-th bit in `flags` and return either 0 or 1 depending on whether these contents are 0 or 1, respectively, for `flagSet`; and 1 and 0, for `flagCleared`.

4.4 Type `Boolean`

SIDE defines type `Boolean` (also named `boolean`) as `char`. This type is intended to represent simple binary flags that can have one of two values: 0 represented by the symbolic constant `NO` (standing for "false") and 1 represented by the symbolic constant `YES` (which stands for "true").

4.5 Pools

In several places, SIDE uses internally the so-called *pools*, which are (possibly ordered) sets of items stored in doubly-linked lists. Sometimes it may be convenient to use those tools (which are made available to the protocol program) to implement a pool of non-standard objects. In order to be managed that way, an object should define two attributes `prev` and `next` to be used as list links, for example:

```
struct my_pool_item_s {
    struct my_pool_item_s *prev, *next;
    ... other stuff ...
};
typedef struct my_pool_item_s my_pool_item;
```

A pool of such objects is represented by a single pointer, e.g.,

```
my_pool_item *Head;
```

which should be initialized to `NULL` (for an empty pool). Then, the following operations (macros) are available:

```
pool_in (item, head);
```

to add the new item pointed to by the first argument (an item pointer) to the pool represented by the second argument (the head pointer);

```
pool_out (item);
```

to remove the indicated item from the pool (`item` is a pointer);

```
for_pool (item, head)
```

to traverse the pool as in this example:

```
my_pool_item *find (int ident, my_pool_item *Head) {
    my_pool_item *el;
    for_pool (el, Head) {
        if (el->Ident == ident)
            return el;
    }
    return NULL;
}
```

and finally,

```
trim_pool (item, head, cond);
```

to remove from the pool all items satisfying the provided condition, e.g.,

```
trim_pool (el, Head, el->Expiry > Time);
```

It may be useful to know that the pool is organized in such a way that new items (added with `pool_in`) are stored at the head. The list is not looped: the `prev` pointer of the first element points to the head (or rather to a dummy item whose `next` pointer coincides with the head pointer. The `next` pointer of the last (oldest) element in the pool is `NULL`.

4.6 Error handling

SIDE is equipped with a standard error handling mechanism, which can also be used by the user protocol program. This simple mechanism offers the following three functions:

```
void excptn (const char *string)
void assert (int cond, const char *string);
void Assert (int cond, const char *string);
```

The first function is used to terminate the session due to a fatal error condition. The text passed as `string` is written to the standard output, standard error, and the results file. When the run is aborted by `excptn`, SIDE prints out the description of the context in which the error has occurred.

A call to `assert` or `Assert` is semantically equivalent to

```
if (!cond) excptn (string);
```

The difference between `assert` and `Assert` is that when the program is created with the "`-a`" option (see section 19.2), all references to `assert` are disabled (removed from the program), whereas references to `Assert` are always active.

4.7 Identifying the session

The following declarative operation can be used to assign a name to a SIDE session:

```
identify name;
```

where *name* can be any piece of text that does not contain blanks. This text will be printed out in the first line of the results file, together with the current date and time (section 4.8).

If the protocol identifier contains blanks, it should be encapsulated in parentheses, e.g.,

```
identify (Expressnet version B);
```

or in quotes, e.g.,

```
identify "Conveyor belt driver";
```

A parenthesis within an identifier encapsulated in parentheses or a quote within a quoted string can be escaped with a backslash.

4.8 Telling time and date

The following function returns the number of seconds of the CPU time used by the program from the beginning of run:

```
double cpuTime ();
```

Note: If the program has been compiled with `-F` (section 19.2), the type of `cpuTime` is `Long`, and the function returns an entire number of seconds.

To get the current date, you can call this function:

```
char *tDate ();
```

which returns a pointer to the character string containing the date in the standard format `www mmm dd hh:mm:ss yyyy`, e.g.,

```
Fri Jan 27 13:07:31 2006
```

Date/time in this format is included in the header of the results file, together with the experiment identifier (section 4.7).

5 SIDE types

By a SIDE type we mean a compound, predefined, user-visible type declared as a class with some standard properties. We conveniently assume that `BIG` (and `TIME`), although sometimes declared as a class, is a simple type. In this section we will be concerned with SIDE types only—in the sense of the above definition. The words "class" and "type" will be used interchangeably—to denote the same concept.

5.1 Type hierarchy

Figure 5 presents the hierarchy of built-in basic SIDE types. We assume that all of them are derived from a common ancestor called *class* which reflects the fact that they are all compound types.

All objects exhibiting dynamic behavior belong to type *Object*, which is an internal type, not visible directly to the user. Its role is to bind together its descendant types by furnishing them with a small collection of common attributes and methods that each *Object* must have. Type `EObject` can be used to prefix user-defined subtypes of *Object*.

The most relevant property of an *Object* (or `EObject`) is that it can be *exposed*. By *exposing* an object (see section 17) we mean presenting some information related to the object in a standard way. This information can be *printed*, i.e., included in the output file produced by the SIDE program, or *displayed*, i.e., shown in a window on the terminal screen.

`Timer` and `Client` stand for specific objects rather than types. These objects represent some important elements of the protocol environment (see section 7.1) and are static, in the sense that they exist throughout the entire execution of the protocol program. Each of them occurs in exactly one copy. Therefore, the actual types of the above objects are uninteresting and are hidden from the user. Other *Objects* may exist in multiple copies; some of them may be dynamically created and destroyed during a simulation run.

Generally, all objects rooted at `AI` (which is another internal type binding the so-called *activity interpreters*) are models of some entities belonging to the protocol environment. They are responsible for modeling or perceiving the flow of time, which is discussed in (section 7.1).

5.2 *Object* naming

Each *Object* has a number of attributes that identify it from the outside and a number of methods for accessing these attributes. The reason for so many identifiers mostly results from the fact that the dynamic display applet (DSD) responsible for exposing *Objects* on the screen (see section 18) must be able to identify individual *Objects* and recognize some of their general properties (see section 17).

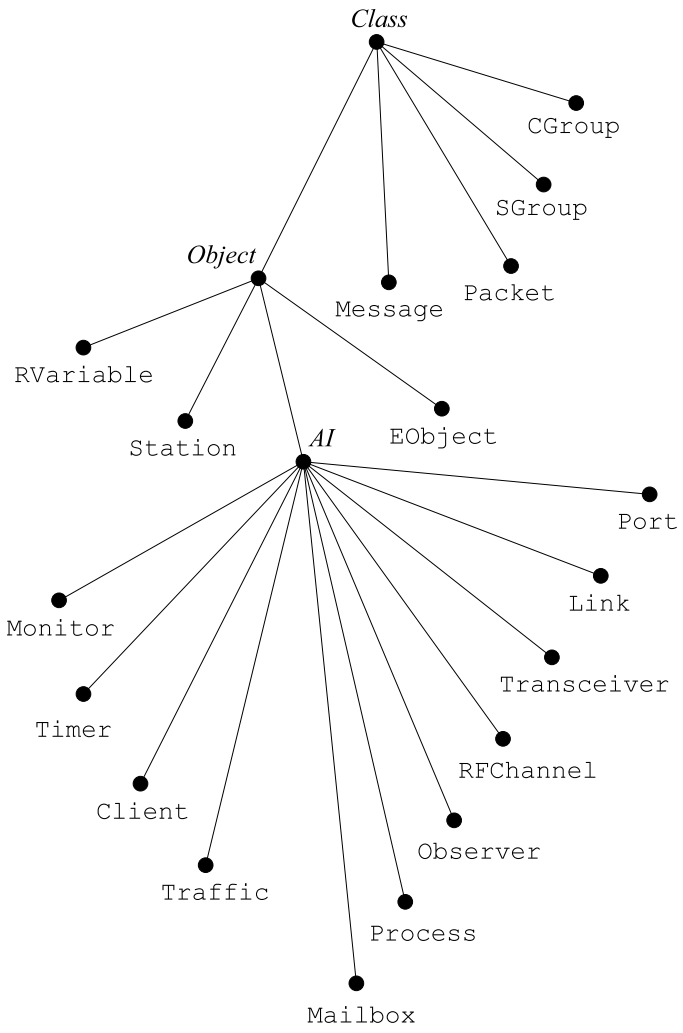

Figure 5: Hierarchy of user-visible compound types.

The following seven identification attributes are associated with each *Object*:

- the class identifier;
- the serial number;
- the type name;
- the standard name;
- the nickname;
- the base standard name;
- the output (print) name.

The *class identifier* of an *Object* is a number identifying the base SIDE type to which the *Object* belongs. This attribute can be accessed by the parameter-less method `getClass`, e.g.,

```
int cl;
...
cl = obj->getClass ();
```

which returns the following values:

`AIC_timer`	if the object is the `Timer` *AI*
`AIC_client`	if the object is the `Client` *AI*
`AIC_link`	if the object is a `Link`
`AIC_port`	if the object is a `Port`
`AIC_rfchannel`	if the object is an `RFChannel`
`AIC_transceiver`	if the object is a `Transceiver`
`AIC_traffic`	if the object is a `Traffic`
`AIC_mailbox`	if the object is a `Mailbox`
`AIC_process`	if the object is a `Process`
`AIC_observer`	if the object is an `Observer`
`AIC_rvariable`	if the object is a `RVariable`
`AIC_station`	if the object is a `Station`
`AIC_eobject`	if the object is an `EObject`

The *serial number* of an *Object*, also called its `Id`, tells apart different *Objects* belonging to the same *class*. This attribute is accessed by the method `getId`, e.g.,

```
int id;
...
id = obj->getId ();
```

With exception of `Ports`, `Transceivers` and `Mailboxes`, all dynamically created *Objects* are assigned their `Id`s in the order of creation, such that the first created *Object* in its *class* is numbered 0, the second object is numbered 1, etc. The numbering of `Ports`, `Transceivers` and `Mailboxes` is described in sections 6.3.1, 6.5.1 and 13.2.1, respectively. The `Id` attribute of the `Timer` and the `Client` (which never occur in multiple copies) is equal to `NONE` (–1).

The *type name* of an *Object* is a pointer to a character string storing the textual name of the most restricted type (C++ `class`) to which the object belongs. This pointer is returned by the method `getTName`, e.g.,

```
char *tn;
...
tn = obj->getTName ();
```

The purpose of the *standard name* is to identify exactly one *Object*. The standard name is a character string consisting of the object's *type name* concatenated with the encoded *serial number*, e.g. `MyProcess 245`. The two parts are separated by exactly one space. For a `Port`, a `Transceiver` and a station `Mailbox` (section 13.2.1), the standard name is built in a slightly more tricky way. Namely, it contains as its part the standard name of the `Station` owning the object (see sections 6.3.1, 6.5.1, 13.2.1). For an object that always occurs in a single instance, the numerical part of the standard name is absent, i.e., the standard names of the `Timer` and the `Client` are just "`Timer`" and "`Client`," respectively. The pointer to the object's standard name is returned by the method `getSName`, e.g.,

```
char *sn;
...
sn = obj->getSName ();
```

The *nickname* is an optional character string which can be assigned to the object by the user. This assignment is usually made when the object is created (see section 5.8). The nickname pointer is returned by the method `getNName`, e.g.,

```
char *nn;
...
nn = obj->getNName ();
```

If no nickname has been assigned to the object, `getNName` returns `NULL`.

The nickname is the only name that can be changed after it has been assigned (all the other names are assigned by the system and they cannot be changed). The following method of *Object* can be used for this purpose:

```
void setNName (const char *nn);
```

where `nn` is the assigned nickname. If the *Object* didn't have a nickname, one is assigned by the method. Otherwise, the previous nickname is discarded and the new one becomes effective.

The *base standard name* is a character string built similarly to the *standard name*, with the exception that the name of the base SIDE type from which the type of the object has been derived is used instead of the object's type name. For example, assume that `ms` points to an *Object* of type `MyStation` which has been derived directly or indirectly from `Station`. The (regular) *standard name* of this object is "`MyStation` *n*", whereas its base standard name is "`Station` *n*". The pointer to the base standard name is returned by the method `getBName`, e.g.,

```
char *bn;
...
bn = obj->getBName ();
```

The base standard names of objects that are direct instances of base types are the same as their (regular) standard names.

The *output name* of an *Object* is a character string which is the same as the *nickname*, if the nickname is defined; otherwise, it is the same as the *standard name*. When an *Object* is exposed *on paper* (see section 17), its output name is used as the default header of the exposure. The method `getOName` returns a pointer to the object's output name, e.g.,

```
char *on;
...
on = obj->getOName ();
```

Note: The character strings returned by `getSName`, `getBName`, and `getOName` are not constants. Thus, if a string returned by one of these functions is to be stored, it must be copied to a safe area. On the other hand, the strings pointed to by `getTName` and `getNName` are constants and they do not change for as long as the object in question is alive (or, in the case of nickname, the name is not changed explicitly).

5.3 Type derivation

Some of the types introduced in section 5.1 are templates that can be used for creating problem-specific types defined by the user. For example, type `Process` can be viewed as a

frame for defining types representing processes to be run at stations. One element of the process that must be provided by the user is a method describing the process *code*, i.e., its behavior. This element is specified in the user's part of the process type definition.

SIDE provides a special way of deriving new types from the built-in ones. Here is the simplest format of the operation that should be used for this purpose:

declarator typename {
    ...
    attributes and methods
    ...
};

where *declarator* is a keyword corresponding to a base SIDE type (section 5.1) and *typename* is the name of the newly defined type.

Example

The following declaration:

```
packet Token {
    int status;
};
```

defines a new packet type called `Token`. This type is built as an extension of the standard type `Packet` and contains one user-defined attribute—the integer variable `status`.

The declarator keyword is obtained from the base SIDE type by changing the first letter to the lower case. The base types that can be extended this way are: `Message`, `Packet`, `Traffic`, `Station`, `Link`, `RFChannel`, `Mailbox`, `Process`, `Observer`, and `EObject`.

In most cases, it is possible to define the new type as a descendant of an already defined type derived from the corresponding base type. In such a case the name of the inherited type should appear after the new type name preceded by a colon, i.e.,

declarator typename : *itypename* {
    ...
    attributes and methods
    ...
};

where `itypename` identifies the inherited type, which must have been declared previously with the same *declarator*. In fact a declaration without *itypename* can be viewed as an abbreviation for an equivalent declaration in which *itypename* identifies the base type. In particular, the previous declaration of type `Token` is equivalent to

```
packet Token : Packet {
    int status;
};
```

For the base types `Traffic`, `Process`, and `Mailbox`, a subtype declaration (sections 10.6.1, 7.2, 13.2.1) may optionally specify one or two *argument types*. If present, these argument types are given in parentheses preceding the opening brace. Thus, the format of a subtype definition for these three base types is

declarator typename : *itypename* (*argument types*) {
    ...
    attributes and methods
    ...
};

where the parts "`:` *itypename*" and "(*argument types*)" are independently optional.

Examples

The following process type declaration:

```
process Guardian (Node) {
    ...
};
```

defines a process type `Guardian` descending directly from the base type `Process`. The argument type should be a defined (or announced—see section 5.6) station type; it tells the type of stations that will be running processes of type `Guardian`.

With the following declaration:

```
process SpecializedGuardian :  Guardian (MyNode) {
    ...
};
```

you can build a subtype of `Guardian`. Processes of this type will be owned by stations of type `MyNode`.

The exact semantics of the SIDE type extension operations will be described individually for each extensible base type. Every such operation is expanded into the definition of a class derived either from the corresponding base type or from the specified supertype. Some standard attributes (mostly invisible to the user) are automatically associated with this class. These attributes will be used by SIDE to keep track of what happens to the objects created from the defined type.

The inherited supertype class is made `public` in the subclass. Following the opening brace of the type definition, the user can define attributes and methods of the new type. By default, all these attributes are `public`. The C++ keywords `private`, `protected`, and `public` can be used to associate specific access rights with user-defined attributes.

5.4 Multiple inheritance

Sometimes it may be convenient to define a SIDE subtype as a direct descendant of more than one supertype. This may be especially useful when creating libraries of types. The standard concept of *multiple inheritance* (present in C++) is more or less naturally applicable in such situations; however, SIDE adds to the issue a specific flavor resulting from the following two problems:

- It is meaningless to define types derived simultaneously from different base types. Thus there should be a way of controlling whether a SIDE type extension makes sense.
- An object of a type derived from multiple supertypes must have exactly one frame of its base type.

The first fact is rather obvious: what would it mean to have an object that is both a `Station` and a `Message` at the same time? The second fact is a consequence of the way in which objects derived from the base SIDE types are handled by the kernel. Internally, each such object is represented by various pointers and tags kept in its base type part. Having two or more copies of this part, besides wasting a substantial amount of space, would have the disastrous effect of perceiving a single object as multiple objects.

In C++ it is possible to indicate that the frame corresponding to a given class occurs exactly once in the object frame, even if the class occurs several times in the object's type derivation. This is achieved by declaring the class as `virtual` in the derivation sequence. Thus, one possible solution to be adopted in SIDE would be to force each base type to occur as `virtual` in the derivation list of a user-defined type. This solution, however, would be quite costly. To make it work, one should always have to make the base types virtual, even if no multiple inheritance were used.[15] This would be expensive because the frame of a virtual subclass is referenced via a pointer, which substantially increases the cost of such a reference. Therefore, another solution has been adopted.

If the name of a newly defined SIDE supertype is followed by `virtual`, e.g.,

```
station VirPart virtual {
    ...
};
```

[15] Note that the SIDE program may be contained in several files and the fact that no multiple inheritance is used in one file does not preclude it from being used in another one.

it means that formally the type belongs to the corresponding base type, but the frame of the base type will not be attached to the type's frame.

The *itypename* part of a SIDE subtype definition (section 5.3) may contain a sequence of type names separated by commas, e.g.,

```
packet DToken :  AToken, BToken, CToken {
    ...
};
```

This sequence represents the inheritance list of the defined subtype. The following rules must be obeyed:

- The inheritance list may contain at most one non-virtual type name (in the sense described above). If a non-virtual type name is present, it must appear first on the list.
- If the name of the defined type is followed by `virtual` (this keyword must precede the colon), the inheritance list must not include a non-virtual type.
- If the inheritance list contains virtual types only, but the keyword `virtual` does not follow the name of the defined type, the defined type **is not virtual**, i.e., the base type is implicitly added to the inheritance sequence (as the first element). Thus only the types explicitly declared as `virtual` are made `virtual`.

One disadvantage of this solution is that a virtual (in the SIDE sense) type cannot reference attributes of the base type. This is not a big problem, as very few of these attributes may be of interest to the user. Virtual functions can be used to overcome this difficulty, if it ever causes a problem.

5.5 Abstract types

A SIDE type can be explicitly declared as incomplete (and thus unusable for creating objects) by putting the keyword `abstract` into its declaration, e.g.,

```
traffic MyTrafficInterface abstract {
    ...
};
```

An abstract type can be derived from other (possibly non-abstract) types, but it cannot be virtual (section 5.4). Thus, the keywords `abstract` and `virtual` cannot occur in the same type declaration.

If an abstract type is derived from other types, then those types must be listed after a colon following the `abstract` keyword, e.g.,

```
traffic MyTrafficInterface abstract :  XTraffic (Msg, Pkt) {
    ...
};
```

Essentially, an abstract type has all the features of a regular type, except that `create` for such a type (section 5.8) is illegal. Moreover, a type *must* be declared `abstract` if it declares abstract virtual methods. For example, this declaration:

```
traffic MyTrafficInterface abstract (Msg, Pkt) {
    virtual Boolean check_it (Pkt *p) = 0;
    ...
};
```

would trigger errors if the `abstract` keyword following the traffic type name were absent.

5.6 Announcing types

A SIDE subtype can be announced before it is actually defined. This may be needed occasionally, if the name of the type is required (e.g., as an *argument type*—section 5.3) before the full definition can take place. The following type declaration format is used for this purpose:

declarator typename`;`

where *declarator* has the same meaning as in section 5.3. The type name can be optionally followed by `virtual` in which case the type is announced as virtual in the sense of section 5.4. Note that a full type definition must precede the usage of this type in the inheritance sequence of another type.

5.7 Subtypes with empty local attribute lists

There are many cases when the local attribute list of a SIDE subtype is empty, e.g.,

```
traffic MyTPattern (MyMType, MyPType) { };
```

This situation occurs most often for traffic patterns (subtypes of `Traffic`) and mailboxes (subtypes of `Mailbox`). In such a case, it is legal to skip the empty pair of braces, provided that the new type declaration cannot be confused with a type announcement (section 5.6). In particular, the above definition of `MyTPattern` can be shortened to

```
traffic MyTPattern (MyMType, MyPType);
```

as, due to the presence of the argument types, it does not look like a type announcement. On the other hand, in the declaration

```
packet MyPType { };
```

the empty pair of braces cannot be omitted, as otherwise the declaration would just announce the new packet type without actually defining it. Generally, an abbreviated type declaration (without braces) is treated as an actual type definition rather than an announcement, if at least one of the following conditions is true:

- the declaration contains argument types
- the declaration specifies explicitly the supertype

Thus, the above declaration of `MyPType` can be written equivalently as

```
packet MyPType : Packet;
```

which, however, is hardly an abbreviation. On the other hand, the most common cases when the attribute list of a newly defined type is empty deal with subtypes of `Traffic` and `Mailbox` and then argument types are usually present.

5.8 Object creation and destruction

If an object belonging to a base SIDE type or its derived type is to be created explicitly, the following operation must be used:

```
obj = create typename ( setup args );
```

where *typename* is the name of the object's type. The arguments in parentheses are passed to the object's `setup` method. If the object has no setup method, or the list of arguments of this method is empty, the part in parentheses does not occur (the empty parentheses can be skipped as well). The operation returns a pointer to the newly created object.

The purpose of the `setup` method is to perform object initialization. If it is needed, it should be declared as

```
void setup (...);
```

within the object type definition. Argument-less C++ constructors can also be used in user-defined derived SIDE types; however, there is no way to define and invoke upon object creation a constructor with arguments. The role of such a constructor is played

by `setup`. Non-extensible types (section 5.3) and types that need not be extended define appropriate standard/default `setup` methods. Their arguments and semantics will be discussed separately for each basic type.

It is possible to assign a nickname (see section 5.2) to the created object. The following version of create can be used for this purpose:

```
obj = create typename, nickname, ( setup args );
```

where *nickname* is an expression that should evaluate to a character pointer. The string pointed to by this expression will be duplicated and used as the object's nickname. If the *setup args* part does not occur, the second comma together with the parentheses may be omitted.

Note: The comma following *nickname* is needed (it is not a mistake).

The *typename* keyword (in both versions of `create`) can be optionally preceded by an expression enclosed in parentheses, i.e.,

```
obj = create (expr) typename ( setup args );
```

or

```
obj = create (expr) typename, nickname, ( setup args );
```

where *expr* must be either of type `int` (in which case it will be interpreted as a station `Id`) or a pointer to an object belonging to a `Station` subtype. It identifies the station in the context of which the new object is to be created and is relevant for some object types only (e.g., see section 7.3). The switchover to the context of the indicated station only affects the `create` operation, as well as any operations triggered by it (e.g., the execution of the created object's setup method).

Objects belonging to types `SGroup`, `CGroup`, `Station`, `Traffic`, `Link`, `RFChannel`, `Mailbox`,[16] `Port`, and `Transceiver` can only be created and never explicitly destroyed. Other dynamically created objects can be destroyed with the standard C++ operator `delete`.

User-defined subtypes of *Object* must belong to type `EObject`, i.e., be actually declared as derivatives of `EObject` or its descendants. A declaration of such a subtype should start with the keyword `eobject` and obey the rules listed in sections 5.3 and 5.4. An `EObject` subtype may declare a `setup` method (or a number of such methods). Object instances of `EObject` subtypes should be generated by `create` and deallocated by `delete`.

[16] This only concerns those mailboxes that are owned by stations. Mailboxes owned by processes can be destroyed (see section 13.2.1).

`Id` attributes of `EObjects` reflect their order of creation starting from 0. This numbering is common for all subtypes of `EObject`.

Note: it is illegal to `create` an object of a virtual type (section 5.4).

6 Defining system geometry

The logical structure of the modeled/controlled system (we shall call it a network), although static from the viewpoint of the protocol program, is defined dynamically by explicit object creation and calls to some functions and methods. As perceived by SIDE, a network consists of *stations* which are conceptually the processing units of the network. A station can be viewed as a hardware (a parallel computer) that runs a collection of *processes*. These processes implement a part of the protocol program which is run by the entire network.

Sometimes (especially in network simulators) stations are interconnected via *links* or/and *radio channels*. Links, represented by type `Link` (and its few built-in variants), are models of simple communication channels that can be implemented in real life on the basis of some broadcast-type communication media, e.g., a piece of wire, a coaxial cable, an optical fiber, or even a wireless link (although `RFChannel`s are generally more useful for the last purpose).

Stations are interfaced to links via *ports*, which can be viewed as some specific points (taps) on the links that the stations may use to send and receive information. Links introduce propagation delays; therefore, one application of a link in a control program is to implement a delayed communication between a pair of stations.

Radio channels, represented by objects of type `RFChannel`, are similar to links, excepts that they allow for more dynamic (in particular mobile) configurations of stations interconnected by them. Also, radio channels are more open-ended: they come with hooks whereby users can describe (program) their favorite characteristics related to transmission power, receiver gain (sensitivity), transmission range, signal attenuation, interference, i.e., all the interesting attributes of real-life radio channels relevant from the viewpoint of modeling signal propagation and packet reception. Owing to the large and unforeseeable set of possible channel models that people may want to plug into SIDE simulators, the package does not presume any built-in characteristics, except for the timing of signals according to the distance among different geographical points.

The radio channel equivalent of a port is called a *transceiver*. Similar to a port, it represents an interface of a station to a radio channel. Multiple (independent) radio channels can coexists within a single network model. Also, links can coexist with radio channels. Thus, hybrid networks can be modeled in SIDE.

6.1 Stations

A station is an object belonging to a type derived from `Station` (see section 5.1). It is possible to create a station belonging directly to type `Station`, but such a station is not very useful.

The standard type `Station` defines only two attributes visible to the user, which are practically never accessed directly. These attributes describe the queues of *messages* awaiting transmission at the station and are discussed in section 10.7.

6.1.1 Defining station types

The definition of a new station type starts with the keyword `station` and obeys the rules listed in sections 5.3 and 5.4. The `setup` method for a station type needs only be defined if the type declares attributes that must be initialized when a station object of this type is created.

Examples

The following declaration:

```
station Hub {
    Port **Connections;
    int  Status;
    void setup () { Status = IDLE; };
};
```

defines a new station type called `Hub` derived directly from `Station`, with two attributes: an array of pointers to ports and an integer variable. When a `Hub` object is created, the `Status` attribute will be set to `IDLE`. The array of port pointers is not initialized in the `setup` method, so, presumably, it will be created outside the object.

The `Hub` type defined above can be used to derive another station type, e.g.,

```
station SuperHub : Hub {
    TIME *TransferTimes;
    void setup (int hsize) {
        Hub::setup ();
        TransferTimes = new TIME [hsize];
    };
};
```

The `SuperHub` type inherits all attributes from `Hub` and declares one private attribute—an array of `TIME` values. The `setup` method has now one argument. Note the call to the

`setup` method of the supertype—to initialize the `Status` attribute.

Stations are the most natural candidates to be defined from virtual supertypes, according to the rules described in section 5.4. For example, consider the following declaration:

```
station TwoBusInterface virtual {
    Port *Bus0Port, *Bus1Port;
    void setup (RATE tr) {
        Bus0Port = create Port (tr);
        Bus1Port = create Port (tr);
    };
};
```

which defines a station component. This component consists of two ports which are created by the `setup` method (the details are explained in section 6.3.1). The two ports may provide a station interface to a two-bus network without specifying any other details of the station architecture. The next declaration

```
station ThreeBuffers virtual {
    Packet Buf1, Buf2, Buf3;
};
```

can be viewed as a description of a `Client` interface (section 10.3) consisting of three packet buffers. The two interfaces can be used together in a single derived definition, e.g.,

```
station MyStationType : TwoBusInterface, ThreeBuffers {
    RVariable *PacketDelay;
    void setup () {
        RATE MyTRate;
        PacketDelay = create RVariable;
        readIn (MyTRate);
        TwoBusInterface::setup (MyTRate);
    };
};
```

as independent building blocks for a complete (non-virtual) station type.

6.1.2 Creating station objects

Given a station type, a station object is built by executing `create` (see section 5.8). The serial number (the `Id` attribute—see section 5.2) of a station object reflects its creation order. The first-created station is assigned number 0, the second one is numbered 1, etc.

The global read-only variable `NStations` of type `int` contains the number of all stations that have been created so far.

As the system geometry must be completely determined before the protocol execution is started, it is illegal to create new stations after starting the protocol. Also, once created, a station can never be destroyed. At first sight, this may look like a limitation reducing the expressing power for the model's dynamics, e.g., in ad-hoc wireless networks, where one may want to study mobility and varying populations of stations that come and go. Note, however, that a station's behavior can be arbitrarily dynamic and independent of the behavior of other stations. Station can remain dormant, then wake up, then become dormant again. In a wireless network, they can also change their locations, move out of range, and so on. The operation of pre-building the model's virtual hardware should be viewed as creating a virtual universe. Once created and set in motion, the "substance" of this universe is not allowed to change, but it can behave in a highly dynamic way.

There exists one special (and typically inconspicuous) station, which is created automatically before anything else. Its purpose is to run a few internal processes of SIDE and also the user's `Root` process (see section 7.9). This station is pointed to by the global variable `System` whose type (a subtype of `Station`) is hidden from the user. The `Id` attribute of the `System` station is `NONE` and its *type name* is `SYSTEM`.

Sometimes it is convenient and natural to reference a station by its serial number, i.e., the `Id` attribute, rather than the object pointer. For example, station numbers are used to identify senders and receivers of messages and packets (see section 10.2). Quite often, especially at the initialization stage, one would like to perform some operation(s) for all stations, or perhaps for their large subset. In such a case, it may be natural to use a loop of the following form:

```
for (int i = 0; i < NStations; i++)  ...
```

To make such operations possible, the following function[17] converts a station `Id` into the pointer to the station object:

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

The function checks whether the argument is a legal station number and, if it is not the case, the execution is aborted with a pertinent error message.

Note: For symmetry, the following macro looks like a function that returns the `Id` attribute of a station (or any *Object*) whose pointer is specified as the argument:

```
int  ident (Object *a);
```

This macro expands into a call to `getId` (see section 5.2).

[17] Implemented as a macro.

6.1.3 Current station

At any moment when a protocol thread (process) is run, as well as during the network initialization phase, it is known which station is the one that is currently *active*. One can say that anything that ever happens in a SIDE program always happens within the context of some station. This (current) station is pointed to by the global variable `TheStation` of type `Station`, which belongs to the so-called *process environment* (see section 7.6). When SIDE starts, and the user program is given control for the first time, `TheStation` points to the `System` station. Then, whenever a new user-defined station is created, `TheStation` points to that last created station. Many other objects that get created along the way (e.g., ports, transceivers, processes) are supposed to belong (be owned) by specific stations; thus, they are assigned by default to the station pointed to by `TheStation` at the moment of their creation. This means that sometimes it may make sense to set `TheStation` explicitly—to assume the context of a specific station and make sure that the object (or objects) subsequently created will be assigned to the right owner. There exists a version of `create` with an extra argument specifying the station that should be made current (assigned to `TheStation`) before the actual object creation (see section 5.8).

6.2 Links

Links are objects belonging to type `Link`, which exists in four standard subtypes: `BLink`, `ULink`, `PLink`, and `CLink`. The primary purpose of a link is to model a simple wired communication channel. Two types of wired channel models, broadcast and unidirectional links, are built into SIDE, each of the two types occurring in two marginally different flavors.

6.2.1 Propagation of information in links

A broadcast link is a uniform information passing medium with the property that a signal entered into any port connected to the link reaches all other ports of the link in due time. A typical real-life example of the broadcast link model is a piece of wire (e.g., a coaxial cable). In a unidirectional link, information travels in one direction only: for any two ports A and B on such a link, there is either a path from A to B or from B to A, but never both. A typical physical implementation of a unidirectional link is a single, segmented fiber-optic channel.

Irrespective of the link type, the actual geometry of the link is described by the *link distance matrix* which specifies the propagation distance between each pair of ports connected to the link. The propagation distance from port A to port B is equal to the (integer) number of *ITU*s required to propagate information from A to B. If an activity, typically a packet transmission (see section 11.1.2), is started on port A at time t, that activity will arrive

on port B at time *t + D[A,B]*, where *D* is the link distance matrix. For a broadcast bidirectional link (types `BLink` and `CLink`), the propagation distance is independent of the direction, i.e., the propagation distance from *A* to B is the same as the propagation distance from B to *A*. In other words, the distance matrix of a broadcast link is symmetric.

For a unidirectional link (types `ULink` and `PLink`), the order in which particular ports have been connected to the link (see section 6.3.2) determines the propagation direction, i.e., if port *A* was connected earlier than B (we assume that the two ports belong to the same link), then signals can propagate from *A* to *B*, but not from B to *A*. It means that an activity inserted at time *t* into *A* will reach B at time *t + D[A,B]*—as for a broadcast link, but no activity inserted into B will ever arrive at *A*. Thus, the distance matrix of a unidirectional link is triangular.[18]

If a unidirectional link has a strictly linear topology, i.e., for every three ports *A*, *B*, *C* created in the listed order *D[A,C] = D[A,B] + D[B,C]*, the link should be declared as a `PLink`, which type can only be used to represent strictly linear, unidirectional links. Semantically, there is no difference between types `PLink` and `ULink` used to represent such a link; however, the `PLink` representation is usually much more efficient, especially when the link is very long compared to the duration of a typical activity inserted into it.

More information about links is contained in section 11.2 where the concept of a collision is discussed. For the purpose of this section, it is enough to say that there are two different methods of recognizing collisions in a bidirectional link. The existence of two types for representing a bidirectional link is a direct consequence of this fact. The two types `BLink` and `CLink` are semantically identical, except for the difference in interpreting collisions. Type `BLink` is equivalent to the simplest generic link type `Link`.

6.2.2 Creating links

Link objects are seldom, if ever, referenced directly by the protocol program, with exception of the network creation phase when the links must be built and configured. It is technically possible (but seldom useful) to define a new link type with the following operation:

```
link newtypename : oldtypename {
    ...
};
```

One use for this possibility is to augment the standard performance measuring link methods by user-defined functions (see section 14.3), or to declare a non-standard link method for generating erroneous packets (section 11.4). If the *oldtypename* part is absent, the new link type is derived directly from `Link`.

[18] In fact, distance matrices do not occur explicitly: they are distributed into *distance vectors* associated with ports rather than links.

A link object is created by `create`—in a way similar to any other *Object* (see section 5.8). The standard link `setup` method is declared with the following header (the same for all link types):

```
void setup (Long np, TIME at = TIME_0, int spf = ON);
```

Thus, the `create` operation for a link expects at least one and at most three setup arguments. The first (mandatory) argument (`np`) specifies the number of ports that will be connected to the link. The second argument (`at`) gives the so-called link *archival time* in *ITU*s. The archival time determines for how long descriptions of activities that have formally disappeared from the link are to be kept in the link data base (see section 11.1.1). The default value (0) means that any activity leaving the link is immediately forgotten (this is how things appear in real life). The last argument indicates whether standard performance measures are to be calculated for the link. With the default value `ON` (1), standard performance measures will be computed; the value `OFF` (0) turns them off.

Note: If you have provided non-standard performance measuring methods for a link (see section 14.3), they will be called even if the link has been created with `spf = OFF`.

A link created according to the above rules is a raw link, in the sense that it is not configured into the network. The value of `np` specifies only the number of slots for ports that will be eventually connected to the link, but these ports are neither created nor anything is said about their geometric distribution. In fact, nothing is known at this moment about the geometry of the link: as we said in section 6.2.1, this geometry is determined by distances between pairs of ports, and these distances must be assigned explicitly (section 6.3.3).

The `Id` attributes of links (and also their standard names—see section 5.2) reflect the order in which the links have been created. The first created link gets number 0, the second link—number 1, etc., until n–1, where n is the total number of links in the network. In many cases when a link is to be referenced, it is possible to use either a pointer to the link object or the link number (its `Id` attribute). The following function:

```
Link *idToLink (Long id);
```

converts a numerical link `Id` to the Link pointer. The global variable `NLinks` of type `int` stores the number of all links created so far.

6.3 Ports

Ports are objects of the standard type `Port`, which is not extensible by the user. Therefore, there is no `port` operation that would declare a `Port` subtype.

6.3.1 Creating ports

There are two ways to create a port. One is to use the standard `create` operation (see section 5.8). The port `setup` method for this occasion is declared as follows:

```
void setup (RATE rate = RATE_0);
```

where the optional argument defines the port transmission rate (attribute `TRate`) as the number of *ITU*s required to insert a single bit into the port. This attribute is relevant if the port will ever be used for *transmission* (see section 11.1) and determines the amount of time required to transmit a packet through the port. If nothing is ever transmitted via the port, i.e., it is used for reception only, the argument can be skipped, in which case the port's transmission rate is undefined (set to 0).

Ports are naturally associated with stations and links. From the viewpoint of the protocol program, the association of a port with a station is more important than its association with a link, as links are not perceived directly by the protocol processes. Therefore, a port can be declared statically within a station type and then created automatically when the station is created, i.e.,

```
Port portname;
```

Note that such a port has no assigned transmission rate. The rate can be set separately by the `setRate` method of `Port`. For illustration, in the following code:

```
station BusStation {
 ...
  Port P;
 ...
  void setup () {
   ...
    P.setTRate (myTRate);
    P.setNName (myNickname);
   ...
  };
 ...
};
```

the statically declared port `P` is assigned a transmission rate as well as a nickname.

In addition to setting the transmission rate of its port, `setTRate` returns the previous setting of the rate. The method is declared with the following header:

```
RATE setTRate (RATE r);
```

This method:

```
RATE getTRate ();
```

just returns the current setting of the transmission rate. As a general rule, most "set" methods have their "get" counterparts.

No matter which way a port has been created, from the beginning of its existence it is associated with a specific station. If the port has been declared statically as a station attribute, the situation is clear: as soon as the station has been created, the port comes into existence and is automatically assigned to the station. When a port is created by `create`, it is assigned to the *current* station pointed to by the environment variable `TheStation` (see section 6.1.3).

All ports belonging to one station are assigned numerical identifiers starting from 0 up to n–1, where n is the total number of ports owned by the station. This numbering of ports reflects the order of their creation and is used to construct the `Id` attributes of ports (section 5.2). Namely, the `Id` attribute of a port consists of two numbers: the `Id` attribute of the station owning the port and the station-relative port number determined by its creation order. These numbers are shown separately in the port's standard name (see section 5.2) which has the following format:

`Port` *pid* `at` *stname*

where *pid* is the relative number of the port at its station, and *stname* is the standard name of the station owning the port (note that it includes the station's `Id`).

The following `Station` method:

```
Port *idToPort (int id);
```

can be used to convert the numerical station-relative port identifier into a pointer to the port object. An alternative way to do the same is to call a global function declared exactly as the above method, which assumes that the station in question is pointed to by `TheStation`.

The following two `Port` methods:

```
int getSID ();
int getYID ();
```

return the numerical `Id` of the port's owning station and the station-relative port `Id`, respectively.

6.3.2 Connecting ports to links

A port that just has been created, although it automatically belongs to some station, is not yet configured into the network. Two more steps are required to specify the port's place in the network structure. The first of these steps is connecting the port to one of the links. The link to which the port is to be connected must have been created previously. One of the following two port methods can be used to connect the port to a link:

```
void connect (Link *l, int lrid = NONE);
void connect (int lk, int lrid = NONE);
```

In the first variant of `connect`, the first argument points to a link object. The second variant allows the user to specify the link number (its `Id` attribute—see section 6.2.2) instead of the link pointer.

A port connected to a link is assigned a link-relative numerical identifier which, in most cases, is of no interest to the user. This identifier is a number between 0 and n–1, where n is the total number of ports connected to the link, and normally reflects the order in which the ports were `connect`ed. For a unidirectional link (types `ULink` and `PLink`), the link-relative numbering of ports (which normally coincides with the connection order) determines the link direction. Namely, if A and B are two ports connected to the same unidirectional link, and the link-relative number of A is smaller than that of B, there is a path from A to B, but not in the opposite direction.

It is possible to assign to a port being connected to a link an explicit link-relative number by specifying this number as the second argument of `connect`. By default, this argument is `NONE` (–1) which means that the next unoccupied link-relative number is to be assigned to the port. The user can force this number to be any number unused so far, but the resulting numbering of all ports that eventually get connected to the link must be continuous and start from 0.

6.3.3 Setting the distance between ports

The last operation required to configure the network is assigning distances to all pairs of ports that have been connected to the same link. One way to set a distance between a pair of ports is to call one of the following two global functions:

```
void setD (Port *p1, Port *p2, double d);
void setD (Port p1, Port p2, double d);
```

which set the distance between ports `p1` and `p2`. The distance `d` is expressed in *distance units* (section 3.2) and is converted internally into ITUs. Typically, the ITU is (much) finer than the distance unit (DU); however, one should remember that the internal representation of the specified distance is always an integer number. In particular, with

the default setting of 1 *1DU* = 1 *ITU*, any fractional components of `d` are rounded (not truncated) to the nearest integer value.

The order in which the two ports occur in the argument list of `setD` is immaterial. For a bidirectional link (types `Link`, `BLink`, `CLink`), the distance from `p1` to `p2` must be the same as the distance from `p2` to `p1`. For a unidirectional link (types `ULink` and `PLink`), only one of these distances is defined, as determined by the connection order (see section 6.3.2).

The following distance setting functions specify only one port each; the other port is implicit:

```
void setDFrom (Port *p, double d);
void setDFrom (Port p, double d);
void setDTo (Port *p, double d);
void setDTo (Port p, double d);
```

Each of the two variants of `setDFrom`, sets the distance **from** port `p` to the port that was last used as the second argument of `setD`. For a bidirectional link, the direction is immaterial: the two-way distance **between** the two ports is set to `d`. For a unidirectional link, SIDE checks whether the direction coincides with the link-relative ordering of the two ports (see section 6.3.2) and if not, the program is aborted with a pertinent error message. Similarly, the two variants of `setDTo` set the distance **to** port `p` from the port that last occurred as the first argument of `setD`. The rules are the same as for `setDFrom`.

The four functions listed above also exist as methods of type `Port`. For such a method, the implicit port is the one determined by `this`, e.g.,

```
myPort->setDTo (otherPort, 12.45);
```

sets the distance from `myPort` to `otherPort` according to the same rules as for the corresponding (global) function `setDTo`.

It is not illegal to define the same distance twice or more times, provided that all definitions specify the same value. For a strictly linear unidirectional link (type `PLink`—see section 6.2.1), there is no need to define distances between all pairs of ports. As soon as enough distances have been provided to describe the entire link, SIDE is able to find the other distances on its own.

It is possible to look up the distance between two ports (connected to the same link) from the protocol program. The following port method:

```
double distTo (Port *p);
```

returns the distance (in *DU*s) from the current (`this`) port to `p`. This distance is undefined (value `Distance_inf` is returned by `distTo`) if the two ports belong to different links or if the link is unidirectional and `p` is situated upstream from the current port.

6.4 Radio channels

In principle, links and ports (section 6.2), in their purely broadcast flavor (type `BLink`) can be used to model radio channels. However, they miss certain rather basic aspects of functionality of such channels, which would require the user to construct too much of their models by hand. Two problems with links are particularly difficult to overcome in this context:

- There is no built-in notion of neighborhood (determined by the propagation range), which means that all stations (ports) connected to the link always see all traffic. While the protocol program can ignore its parts based on a programmed notion of range, sensitivity, and so on, the model of a large network with relatively small neighborhoods would incur a lot of overhead.
- It is extremely difficult to implement any kind of node mobility. The problem is that once the topology/geometry of a link is determined (at the initialization stage), it is meant to remain fixed for the entire simulation run.

These problems get in the way of any attempts to tweak the link models to offer the kind of flexibility one would like to have with interesting models of RF channels. Even ignoring the fact that links lack any explicit tools for modeling signal attenuation and interference (and the programmer would have to be responsible for creating them all), the rigid and global structure of links makes them practically useless as models of wireless links (even though they have proved extremely useful for modeling wired networks).

Consequently, SIDE provides a separate collection of tools for expressing the behavior of wireless channels. Those tools are comprised of *radio channels* (type `RFChannel`) and *transceivers* (type `Transceiver`), which can be viewed as the wireless equivalents of links and ports. Many general concepts applicable to links and ports directly extend over their wireless counterparts.

6.4.1 The concept

Similar to links and ports, the role of a radio channel in SIDE is to interconnect transceivers, which (like ports) provide the interface between stations and radio channels. In contrast to a link, a radio channel need not guarantee that a signal originating at one transceiver will reach all other transceivers Another important difference is that the built-in channel type `RFChannel` is more open-ended than `Link` (or any of its built-in subtypes). Although it does provide a complete functionality of sorts, that functionality is practically useless. This is because the issue or modeling real-life wireless channels (with a meaningful accuracy) is considerably more complex than modeling a piece of wire. Owing to the proliferation of wireless channel models in the literature, it would be unwise to restrict our model to one (or some) of them. Consequently, `RFChannel` should be viewed

as a generic parent type for building actual channel types, whose exact behavior is fully specified by a collection of virtual methods provided by the user.

The primary role of those methods is to determine how a signal attenuates over distance, and how the levels of multiple signals perceived by the same recipient determine whether any of those signals can be recognized as a valid packet. Also, they determine the propagation distance (the so-called *cut-off distance*) beyond which the signal becomes irrelevant. This is the distance that determines which receivers stand a chance of ever perceiving a signal originating from a given point.

The network map, i.e., the notion of distance used as a parameter of some of the virtual methods mentioned above, is derived from the assumption of the two- or three-dimensional Euclidean geometry of the space in which the network nodes are deployed. Specifically, we are talking about Cartesian distance in two or three dimensions. The dimensionality of the network's deployment area is selected at compilation time (see section 19.2). As most practically interesting cases are expressible in two dimensions, the default is 2-D, i.e., node locations are described by pairs of Cartesian coordinates, which simplifies network parameterization and reduces the execution time of the model. With the 3-D option, certain methods receive different headers—to accommodate one more coordinate needed to specify node locations.

This Euclidean component of the generic signal propagation model does not preclude specific models in which attenuation/interference is determined by rules that take other factors into consideration. It is up to the user to implement the virtual methods that ultimately decide the fate of packets, and their decisions can be based on arbitrary criteria. The distance-driven built-in behavior of radio channels provides a vehicle for describing and timing events caused by signals (representing packets) propagated among transceivers. Some of those events (e.g., a packet reception) are affected by a user-provided set of methods that determine when a packet is deemed receivable, e.g., based on the distance traveled by it, and/or the presence of other signals perceived by the transceiver at the same time.

In contrast to the distance matrix of a link (section 6.2.1), the distance matrix of a radio channel is flexible (and it is not really a matrix). Transceivers can change their position in a practically unrestricted way. A natural way to model node mobility is to have a special process (section 7) that modifies the locations of nodes according to some (e.g., stochastic) pattern. Even though the coordinates of nodes (or rather transceivers) are discrete (expressed in *ITU*s interpreted as distance units—section 3.2), the *ITU* can be chosen to represent an arbitrarily fine unit of distance. This is important because the coordinate change is always instantaneous: a node is simply teleported from one location to another. There is no problem with this approach as long as the mobility pattern is realistic, i.e., the nodes move in reasonably small steps at a reasonably low (non-relativistic) speed. It is up to the user to maintain the sanity of the mobility model.

6.4.2 Creating radio channels

A single network model can have several radio channels, which can also coexist with (wired) links. This way, it is possible to build models of hybrid networks. Different radio channels are independent: if the user wants to model their cross-interference, the virtual methods describing the conditions for packet reception must account for that. Those methods are provided in user-defined classes extending the built-in `RFChannel` type, in the following way:

```
rfchannel newtypename : oldtypename {
    ...
    attributes and methods
    ...
};
```

If the *oldtypename* part is absent, the new radio channel type is derived directly from `RFChannel`. It is possible to use a user-defined subtype of `RFChannel` as the ascendant of another subtype; in such a case, the ascendant type should appear as *oldtypename* in the declaration.

The `setup` method of `RFChannel` has the following header:

```
void setup (Long nt, int spf = ON);
```

where `nt` is the total number of transceivers to be interfaced to the channel, and `spf` plays the same role as the corresponding link parameter (section 6.2.2).

Similar to links, the `Id` attributes of radio channels (and also their standard names—see section 5.2) reflect the order in which the channels have been created (the first channel is assigned number 0, and so on). In those few cases when a radio channel has to be specified as an argument of a function/method, it is possible to use either a pointer to the `RFChannel` object or the channel's `Id` attribute. The following function:

```
RFChannel *idToRFChannel (Long id);
```

converts a numerical channel `Id` to the `RFChannel` pointer. The global variable `NRFChannels` of type `int` stores the number of all radio channels created so far.

Similar to a link, the actual geometry of a radio channel is determined by the configuration of its transceivers. In particular, those transceivers are assigned positions represented by pairs (or triplets) of coordinates (section 6.5). The distance between a pair of transceivers is determined as the Euclidean distance in two or three dimensions and expressed in DUs (section 3.2).

The primary reason for extending the base type `RFChannel` is to define the so-called *assessment methods*, i.e., virtual functions describing the channel's properties, viz. the conditions for packet reception. Some of the assessment methods accept signal levels, which are typically accompanied by so-called *tags*. The latter can be used to associate generic user-defined properties with the signals, e.g., codes for CDMA-type channels, which will provide additional input driving the decisions taken by the assessment methods.

Thus, the representation of a single signal (typically corresponding to a transmitted packet) arriving at a transceiver is captured by the following structure:

```
typedef struct {
    double Level;
    IPointer Tag;
} SLEntry;
```

where `IPointer` is the smallest integer type capable of accommodating a pointer (see section 3.1). This way, `Tag` can be a simple (integer) value, but it may also point to an arbitrary structure, if the complexity of the channel model calls for that. The tag attribute of the signal associated with a transmitted packet is automatically inherited from the current value of the tag attribute associated with the originating transceiver (see section 6.5.1).

The same `SLEntry` structure can also be used to describe the parameters of a perceiving transceiver, i.e., one carrying out an assessment. In such a case, the `Tag` attribute of that structure refers to the current setting of the transceiver's own tag. The `Level` attribute may then refer to the signal level of the transceiver's own transmission (as in `RFC_add` below), or to its reception *gain* (as in `RFC_act`, `RFC_bot`, `RFC_eot`, `RFC_erb`, and `RFC_erd`). The latter is yet another settable parameter of a transceiver that may affect signal perception.

Below we list the assesment methods and briefly describe their roles. The exact explanation of the way those methods are used to determine the fate of packets is given in section 12.1.4.

```
double RFC_att (const SLEntry *sl, double d, Transceiver *src);
```

This method calculates signal attenuation depending on the distance and possibly other attributes that can be extracted from the signal tag as well as the source and destination transceivers. The first argument is the original (transmitted) signal level (along with its tag), the second argument is the distance in DUs separating the source and destination transceivers. The source transceiver is pointed to by `src`; the destination transceiver can be referenced via the environment variable `TheTransceiver` (`Info02`). The function is expected to return the attenuated level of the original signal `sl->Level`, i.e., its level being actually perceived by the destination transceiver.

```
double RFC_add (int n, int ex, const SLEntry **sl, const SLEntry *xm);
```

This method provides the prescription for calculating the aggregation of multiple signals arriving at the receiver at the same time. The first argument (`n`) specifies the number of entries in `sl`, which is an array of signal levels representing all the individual activities perceivable by the transceiver. The last argument (`xm`) pertains to the present transceiver. If the transceiver is currently transmitting, then the `Level` attribute of `xm` is nonzero and represents the transmitted signal level. In all circumstances, the `Tag` attribute of `xm` indicates the current value of the transceiver's `Tag`. If `ex` is not `NONE` (-1), it gives the index of one entry in `sl` that should be ignored. This operation is also used in calculating the total level of interference affecting one selected signal; in such a case, the signal in question should be excluded from the calculation.

```
Boolean RFC_act (double sl, const SLEntry *s);
```

This method is used to tell whether the transceiver senses any signal at all (carrier sense) based on the total combined signal level `sl` arriving at it at the moment. The second parameters contains the receiver's `Tag` as well as its gain (sensitivity) presented in the `Level` component of `s`.

The purpose of this method is to determine when the channel is perceived *busy* or *idle* (and when to trigger the events `ACTIVITY` and `SILENCE`—section 12.2.1).

```
Boolean RFC_bot (RATE r, const SLEntry *sl, const SLEntry *rs, const IHist
    *ih);
```

The responsibility of this method is to assess whether the beginning of a packet arriving at a transceiver is recognizable as such, i.e., the packet stands a chance of being received. The first argument is the transmission rate (at which the packet was transmitted—in *ITU*s), `sl` gives the received signal level[19] of the packet along with its tag, `rs` describes the receiver's tag along with its gain (sensitivity), and `ih` points to a special data structure called the *interference histogram*, which stores the history of interference suffered by the packet's preamble. The layout and interpretation of interference histograms are described in section 12.1.5.

```
Boolean RFC_eot (RATE r, const SLEntry *sl, const SLEntry *rs, const IHist
    *ih);
```

This method determines whether the end of a packet arriving at a transceiver may trigger a successful reception of the packet. The arguments are exactly as for `RFC_bot`, except that the interference histogram applies to the packet rather than the preamble.

[19] In technical terminology, this is called RSSI, which stands for Received Signal Strength Indication.

```
double RFC_cut (double sn, double rc);
```

This method returns the *cut off* distance (in DUs), i.e., the distance after which a signal ceases to be noticeable (causes no perceptible interference). The two arguments specify the transmitted signal level and the receiver's gain. The method is used to keep track of node neighborhoods, i.e., groups of nodes being affected by other nodes. Note that it does not take tags into consideration and bases its estimate solely on numerical signal levels. Conservative values returned by `RFC_cut` are formally safe, but they may increase the simulation time owing to unnecessarily large neighborhoods.

```
TIME RFC_xmt (RATE r, Long nb);
```

This method determines the amount of time needed to transmit `nb` (payload) bits at rate `r` through the channel. For example, the encoding scheme assumed by the channel may transform logical (payload) bits into physical bits (e.g., encoding each four-bit nibble into a six-bit symbol). Also, if there are any extra framing bits (like start/end symbols) added to physical packets, the method should account for them. Note, however, that the preamble is a separate issue. It is considered a `Transceiver` attribute (see section 6.5.1), which means that different transceivers may use different preamble length, is always expressed directly in physical bits, and is included implicitly and separately in the transmission time of every packet. This is explained in detail in section 12.1.8.

```
Long RFC_erb (RATE r, const SLEntry *sl, const SLEntry *rs, double il, Long
    nb);
```

The method returns the randomized number of error bits within the sequence of `nb` bits received at rate `r`, for the received signal described by `sl`, with receiver parameters `rs`, and at the interference level `il`. Its role is to describe the distribution of bit errors as a function of the signal to interference (SIR) ratio. The method is only needed if the protocol program calls `errors` or `error`, which occur in several variants as methods of `RFChannel` and `Transceiver` (see section 12.2.4). Note that the *bits* counted by `RFC_erb` (and also `RFC_erd`), are "physical" (see section 12.1.1).

```
Long RFC_erd (RATE r, const SLEntry *sl, const SLEntry *rs, double il, Long
    nb);
```

The first four arguments are as for `RFC_erb`. The method returns the randomized number of received bits (under the conditions described by the first four arguments) preceding the occurrence of a user-definable "special" configuration of bits described by the last argument. Typically, that configuration (the timing of its occurrence described as a bit interval) is strongly related to the

distribution of bit errors (thus, `RFC_erd` is closely correlated with `RFC_erb`). For example, the method may return the number of bits preceding the nearest occurrence of a run consisting of `nb` consecutive bit errors. The method is used solely for triggering `BERROR` events, whose interpretation is up to the protocol program (section 12.2.4). In particular, if the program is not interested in perceiving those events, the method need not be provided.

Type `RFChannel` declares the above methods as virtual, so they are expected to be defined in its subtypes. The default definitions do provide a complete functionality (such that `RFChannel` is a complete model), but that functionality is naive. For illustration, here are the built-in versions of `RFChannel`'s virtual methods:

```
virtual double RFC_att (const SLEntry *sl, double d, Transceiver *src) {
    // No attenuation, tag, distance, source, destination ignored
    return sl->Level;
};

virtual double RFC_add (int n, int ex, const SLEntry **sl, const SLEntry *xm) {
    // Straightforward additive combination of signals,
    // including the interference of own transmitter
    double total;
    total = xm->Level;
    while (n--)
        if (n != ex)
            total += sl [n] -> Level;
    return total;
};

virtual Boolean RFC_act (double sl, const SLEntry *rs) {
    // Any nonzero signal means "activity"
    return (sl > 0);
};

virtual Boolean RFC_bot (RATE r, const SLEntry *sl, const SLEntry *rs,
                                                        const IHist *ih) {
    // At least 8 trailing bits of the preamble with zero
    // interference (rs ignored)
    Long nbits;
    return ((nbits = ih->bits (r)) >= 8 && ih->max (r, nbits - 8, 8) == 0.0);
};

virtual Boolean RFC_eot (RATE r, const SLEntry *sl, const SLEntry *rs,
                                                        const IHist *ih) {
    // Zero interference throughout the entire packet
    return ih->max () == 0.0;
};
```

```
virtual double RFC_cut (double xp, double rp) {
    // Global village: cutoff distance is infinite, all nodes
    // are neighbors
    return Distance_inf;
};

virtual TIME RFC_xmt (RATE r, Long nb) {
    // Assumes that logical bits are same as physical
    return (TIME) r * (LONG) nb;
};
```

The default (functionally void) stubs for `RFC_erb` and `RFC_erd` raise errors when called. It is possible to have a fully functional and non-trivial channel model that makes no reference to those methods.

The configuration of the redefinable methods of `RFChannel` has been chosen in such a way as to offer the user high flexibility in specifying the exact behavior of the channel, while being relatively simple to understand and program. Note that the decisions (i.e., the values returned by the `Boolean` methods) can (and often should) be randomized, e.g., to fit specific distributions of bit error rates parameterized by detailed interference profiles.

The interpretation of the signal levels and their role in determining the fate of bits and packets is up to the assessment functions. For example, the default variant of `RFC_add` assumes the linear interpretation of the signals, which is internally preferred, as it simplifies signal combination at the receivers. It is possible to express signal levels in dBm (and their ratios in dB); however, their interpretation must be consistent across all the assessment methods. In particular, `RFC_add` must be rewritten in such a case to use a different formula for signal addition.

The following conversion functions have been provided to help transform signals and their ratios between the linear and logarithmic scales:

```
double dBToLin (double db);
double linTodB (double ln);
```

The first one converts decibels to linear, the other works in the opposite direction. For example, `dBToLin(30.0)` returns 1000.0. Also, `linTodB(dBToLin(a))` returns `a` (or something extremely close).

6.5 Transceivers

Transceivers are represented by type `Transceiver`, which, similar to type `Port`, is not extensible by the user. There are more similarities between transceivers and ports: the

two types of objects play conceptually the same role of interfacing communication channels to stations.

6.5.1 Creating transceivers

One natural way to create a transceiver is to resort to the standard `create` operation (section 5.8). The `setup` method, of `Transceiver`, which determines the configuration of arguments for `create`, has the following header:

```
void setup (RATE r = RATE_0, int pre = 0,
            double xp = 0.0, double rs = 0.0,
            double x = Distance_0, double y = Distance_0);
```

All arguments are optional and have the following meaning:

`r`
: This is the transmission rate in ITUs per bit, analogous to the corresponding `Port` attribute. The default setting of 0 makes it impossible to use the transceiver for transmission (it can still be used for reception).

`pre`
: The preamble length in "physical" bits to be inserted by the transmitter before an actual packet. If the transceiver is to be used for transmission, this length cannot be zero.

`xp`
: Transmission power, i.e., the signal level of the transmitter. This number is used as the original signal level of all packets transmitted by the transceiver, e.g., as the argument to `RFC_att`, as well as the transmitter's interference level (argument `xmt` of `RFC_add`).

`rs`
: Receiver gain (sensitivity). This number is used as the argument in `RFC_act`, `RFC_bot` and `RFC_eot`.

`x` **and** `y`
: These are the (initial) planar coordinates of the transceiver expressed in DUs. By default, all transceivers initially pile up at the same location with coordinates $< 0, 0 >$. Note that locations apply to transceivers rather than to stations. Thus, it is possible to have multiple transceivers of the same station appear at different geographical locations.

Note: If the simulator has been created with the `-3` option (selecting the 3-dimensional spatial model—section 19.2), the `setup` method of `Transceiver` receives one extra argument, i.e., its header is:

```
void setup (RATE r = RATE_0, int pre = 0,
            double xp = 0.0, double rs = 0.0,
            double x = Distance_0,
            double y = Distance_0,
            double z = Distance_0);
```

where `z` stands for the third coordinate.

It is possible to create a transceiver with the default set of attributes (i.e., without specifying any arguments to `create` and then set the relevant attributes by invoking some methods. Similar to a port, a transceiver can be declared statically within a station type, e.g.,

```
station SensorNode {
 ...
  Transceiver RFI;
 ...
  void setup () {
   ...
    RFI.setTRate (myTRate);
    RFI.setPreamble (32);
    RFI.setNNName ("SensorInterface");
    RFI.setLocation (24.1, 99.9);
    RFI.setMinDistance (0.1);
   ...
  };
 ...
};
```

(compare section 6.3.1).

Similar to a port, a transceiver must always be associated with a specific station. If the transceiver has been declared statically as a station attribute, it comes into existence together with the station and is automatically assigned to it. When a transceiver is created by `create`, it is assigned to the *current* station pointed to by the environment variable `TheStation` (see section 6.1.3).

All transceivers belonging to one station are assigned numerical identifiers starting from 0 up to n–1, where n is the total number of transceivers owned by the station. This

numbering reflects the order of transceiver creation and is used to construct the `Id` attributes of transceivers (section 5.2), which resemble the attributes of ports (section 6.3). In particular, the standard name of a transceiver has the following format:

`Tcv` *tid* `at` *stname*

where *tid* is the relative number of the transceiver at its station, and *stname* is the standard name of the station owning the transceiver.

The following `Station` method:

```
Transceiver *idToTransceiver (int id);
```

converts the numerical station-relative transceiver identifier into a pointer to the transceiver object. An alternative way to accomplish the same effect is to call a global function declared exactly as the above method, which assumes that the station in question is pointed to by `TheStation`.

The following two `Transceiver` methods:

```
int getSID ();
int getYID ();
```

return the numerical `Id` of the transceiver's owning station and the station-relative transceiver `Id`, respectively.

6.5.2 Interfacing and configuring transceivers

One important step to be accomplished before a transceiver can become functional is to interface it to a radio channel. This can be done by invoking the following method of `RFChannel`:

```
void connect (Transceiver *tcv);
```

or one of these methods of `Transceiver`:

```
void connect (RFChannel *rfc);
void connect (int rfcId);
```

In the last case, the argument is the numerical `Id` of the radio channel (section 6.4.2).

In contrast to ports and links, there is no need to set up a distance matrix for the radio channel. The coordinates of transceivers determine their Euclidean distances on the virtual plane or in the virtual 3-space. From this perspective, the role of the radio channel is to

collect all transceivers that share the same wireless link, and, based on the indications of `RFC_cut` (section 6.4.2), may become neighbors and perceive each other's signals. The neighborhoods maintained by the radio channel are dynamic: they are updated whenever a node moves or changes its transmission power or receiver gain.

Note that the the relation of neighborhood need not be symmetric. First of all, different transceivers may use different transmission power and receiver gain. Then, the cut-off method (`RFC_cut`—section 6.4.2) can do whatever it pleases to decide when a receiving node qualifies as a transmitter's neighbor. This direction, i.e., transmitter to receiver, defines the concept of a neighborhood. In other words, by a neighborhood, we mean the population of all receivers that are in principle reachable from a given transmitter.

The following methods od `Transceiver` may be useful for configuring transceivers. We list them here, as they can be sensibly called before the protocol is started, but the detailed explanation of some of them will be given later, in section 12, where we shall augment this list with a few more methods.

```
RATE setTRate (RATE r);
```

This method sets (or re-sets) the transmission rate of the transceiver. The transmission rate can be changed at any time, and it immediately becomes effective for all packets transmitted after the change. The method returns the previous setting of the transmission rate.

```
Long setPreamble (Long p);
```

This method sets (or re-sets) the preamble length (in physical bits—section 12.1.1) to be applied to all packets subsequently transmitted from the transceiver. The method returns the previous setting of the preamble length.

```
double setXPower (double xp);
```

This method sets (or re-sets) the transmission power of the transceiver returning the previous value.

```
double setRPower (double xp);
```

This method sets (or re-sets) the receiver gain returning the previous value.

```
IPointer setTag (IPointer tag);
```

This method sets (or re-sets) the tag associated with the port, which will be used to tag all signals of subsequently transmitted packets. Signal tags are attributes of signal level descriptors (structure `SLEntry`—section 6.4.2) and can be interpreted by `RFC_add`, as special user-defined attributes affecting the calculation of the combined signal level (or interference) at a receiver. The method returns the previous value of the tag.

The default value of a transceiver's tag is zero. If the channel model needs no tags, the user can ignore this feature.

`void setLocation (double x, double y);`

This method moves the transceiver to a new location, with the specified Cartesian coordinates expressed in *DU*s. The move is instantaneous (a teleportation). While teleporting nodes over long distances to their initial locations is perfectly OK during the initialization stage, the operation should be used with care during the protocol execution. From the viewpoint of realism of the mobility model, individual movements should be short and combined (over pertinent delays) into a reasonably smooth trajectory covered within a reasonable time.

Note that locations apply to transceivers, not stations; thus, it is possible for multiple transceivers of the same station to be positioned at different locations. This seldom makes sense. If you have stations with multiple transceivers, you can make sure that they are being moved consistently by using the `Station` variant of `setLocation` (whose header is identical to the `Transceiver` variant).

If the simulator has been created with the `-3` option (selecting the 3-D geometry of node deployment—section 19.2), the method is declared as

```
void setLocation (double x, double y, double z);
```

with the last argument needed to pass the value of the third coordinate. This applies to both variants of the method, i.e., `Transceiver` as well as `Station`.

`void setMinDistance (double d);`

This method sets the minimum distance between a pair of transceivers. It is generally reasonable to assume that no two transceivers ever occupy exactly the same location, i.e., the distance between them can be exactly zero. Thus, even if they actually fall into the same spot on the (discrete) plane, SIDE will make sure that their distance is not less than the specified minimum (in *DU*s). The default setting of minimum distance is the *DU* equivalent of 1 *ITU*, i.e., the minimum non-zero distance.

Each of the above "set" methods has a corresponding "get" method that just returns the value of the respective attribute without changing it. Except for `getLocation`, such a method takes no arguments and returns a value of the pertinent type. For `getLocation`, the header is:

```
void getLocation (double &x, double &y);
```

or

```
void getLocation (double &x, double &y, double &z);
```

depending on the selected dimensionality of the node deployment space. The method returns the two or three coordinates of the transceiver (in DUs) via the arguments. There exists a `Station` variant of the method (with the same header) which returns the location of the first transceiver of the station. Note that normally all transceivers of the same station are at the same location, although this is not formally required.

In addition to `getPreamble`, this `Transceiver` method:

```
TIME getPreambleTime ();
```

provides a fast shortcut producing the amount of time (in ITUs) taken by the transmitted preamble. This is equivalent to multiplying the value returned by `getPreamble` by the transceiver's transmission rate.

To determine the total amount of time it would take to transmit a given number of (logical) bits, use this `Transceiver` method:

```
TIME getXTime (Long bits);
```

The value returned by it tells the number of ITUs needed to transmit a packet whose `TLength` attribute is equal to `bits`. Note that this transformation is more complex than in the case of `Port` (where it boils down to a simple multiplication by `TRate`—section 11.1.2) and involves `RFC_xmt` (section 12.1.8).

Many attributes of transceivers can be set globally, i.e., to the same value for all transceivers interfaced to the same radio channel, by calling the corresponding method of `RFChannel` rather than the one of `Transceiver`. Except for `setLocation`, all the "set" methods listed above have their `RFChannel` variants, which may be useful as shortcuts at the initialization stage. The `RFChannel` methods return no values (they are declared as `void`). A few more "set/get" methods of transceivers (and their global `RFChannel` versions) will be introduced in section 12.2.4.

One useful method of `RFChannel`, which has no `Transceiver` counterpart, is

```
double getRange (double &lx, double &ly, double &hx, double &hy);
```

and its 3-D version

```
double getRange (double &lx, double &ly, double &lz,
                 double &hx, double &hy, double &hz);
```

The 2-D variant returns via the four arguments the lower-left and upper-right coordinates of the rectangle encompassing all the transceivers interfaced to the channel, according to

their current locations. The coordinates are in *DU*s and reflect the extreme coordinates of all transceivers. Thus, `lx` is the smallest `x` coordinate, `ly` is the smallest `y` coordinate, `hx` is the highest `x` coordinate, and `hy` is the highest `y` coordinate. The method's value gives the length of the rectangle's diagonal. The 3-D version adds the third coordinate to those values—in the natural way.

Following the initialization stage, SIDE defines the neighborhoods of transceivers based on the indications of `RFC_cut`. Regardless of the indications of the other methods used to assess the level of received signals and their interference, no signal is ever perceived outside its neighborhood. Consequently, the indications of `RFC_cut` should be conservative enough to provide for a realistic model, while being restrictive enough to narrow down the size of neighborhoods, which directly impacts the simulation time. If not sure, you can start with large neighborhoods (e.g., a trivial `RFC_cut` returning the constant `Distance_inf`) and then narrow them down experimentally checking how this reduction affects the observed behavior of the network. In any case, the only penalty for large neighborhoods is a longer execution time and, possibly, increased memory requirements of the simulator.

Note that neither new stations nor new transceivers can be created once the protocol has started. As we point out in section 6.1.2, this is not a serious restriction.

7 Processes

The dynamic part of the user specification, i.e., the protocol program, has the form of a collection of cooperating threads called processes. Each process can be viewed as an *event handler*: its processing cycle consists of being awakened by some event, responding to the event by performing some protocol-related activities, and going back to sleep to await the occurrence of another event.

7.1 Activity interpreters: general concepts

An *activity interpreter*, or *AI* for short, is an object belonging to a (possibly derived) SIDE type rooted under *AI* in figure 5 (section 5.1). Objects of this type are responsible for triggering *events* that can be perceived by processes and for advancing their notion of time. To await an event, a process issues a *wait request* addressed to a specific *AI*. Such a request specifies a class of *events* that the process would like to respond to. The occurrence of an event from this class will *wake up* the process.

A typical example of an *AI* is a port. A process, say P_1, may issue a wait request to a specific port, say p_1, e.g., to be awakened as soon as a packet arrives at p_1. Another process P_2 may start a packet transmission on port p_2 connected to the same link. This activity exhibited by P_2 will be transformed by the link into an event that will wake up process P_1 when, according to the propagation distance between p_1 and p_2, the packet has

arrived at p_1.

Different *AI*s use different means (methods) to learn about the relevant activities exhibited by processes. There is, however, a standard *AI* interface that can be used by a process to issue a wait request. This interface is provided by the following method:

```
void wait (etype event, int state, LONG order = 0);
```

where `event` identifies an event class (the type of this argument is *AI*-specific), and `state` identifies the process state (see section 7.2) to be assumed when the awaited event occurs. The last argument (`order`) is optional. It can be used to assign priorities to awaited events (section 7.4). When multiple events occur at the same modeled time (within the same *ITU*), these priorities determine the order in which the events are presented. The third parameter of `wait` can only be specified if the program was created with the `-p` option of `mks` (section 19.2).

7.2 Defining process types

A process is an object belonging to type `Process`. A process consists of its private *data* area (attributes) and a possibly shared *code* describing the process's behavior. To create a non-trivial process, i.e., one that has some code, the user has to define a subtype of `Process`. Such a definition must obey the standard set of rules (sections 5.3, 5.4) and begin with the keyword `process`. Typically, it has the following layout:

```
process ptype : itypename ( ostype, fptype ) {
    ...
    local attributes and methods
    ...
    states { list of states };
    ...
    perform {
        the process's code
    };
};
```

where *ostype* is the type name of the station owning the process (see section 7.3) and *fptype* is the type name of the process's *father*, i.e., the creating process (see section 7.3).

Each process is always created in the context of (by) some existing process (the father) and belongs to a specific station (see section 7.3). There exist two standard process attributes, called `F` (for father) and `S` (for station), that point to the the process's father and the station owning the process. They are implicitly declared within the process type as

```
ostype    *S;
```

```
fptype    *F;
```

i.e., the arguments `ostype` and `fptype` determine the type of the attributes `F` and `S`. If the parenthesized part of the process type declaration header is absent, the two attributes are not defined. This is legal as long as the process makes no reference to them. If only one argument is specified within parentheses, it is interpreted as the owning station type. In such a case, the `S` pointer is defined while `F` is not.

A `setup` method can be declared for a process type (see section 5.8)—to initialize the local attributes (data) upon process creation. The default `setup` method provided in type `Process` is empty and takes no arguments.

The definition of the process code starting with the keyword `perform` resembles the declaration of a (special) argument-less method. The body of the code method looks like a description of a finite state machine (see section 7.5). The states of this machine must be announced with the `states` declaration and can be any C++ identifiers. These identifiers are viewed as enumeration constants.

As for a regular method, the body of the code method can be defined outside the process type—in the following way:

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

In such a case, the process type declaration must contain an announcement of the code method in the form

```
perform;
```

A process type declared as `virtual` (section 5.4) is not allowed to specify a code method.

A process type T_2 derived from another process type T_1 naturally inherits all attributes and methods of T_1, including the `setup` method(s) and the code method. Of course, these methods can be redefined in T_2, in which case they subsume the corresponding methods of T_1. Although a subsumed `setup` method of the parent type T_1 can be referenced from T_2, there is no natural way to reference the subsumed parent's code method (it would seldom make sense—see section 7.5).

It is legal to have a process type that defines no code method. A process of such a type can be run, provided that a code method is declared somewhere in a supertype. Note that this method may use virtual functions relating its behavior to the code-less subtype.

7.3 Creating and terminating processes

A process is created in a way similar to any other *Object*, with the `create` operation (section 5.8). All activities of the protocol program are always performed from within a process (see section 7.9); therefore, a (user-defined) process is always created by another (known) process. This creating process becomes the *father* of the created process and can be referenced via its `F` attribute (see section 7.2). Similarly, each process belongs to a specific station, which is pointed to by `TheStation` (see section 6.1.3) whenever the process is run. The contents of `TheStation` at the moment of process creation determine the station that will own the new process (see section 5.8). This station can be referenced by the process's `S` attribute (see section 7.2) or by `TheStation`. The latter way of referencing the station owning the process is usually less convenient as the type of `TheStation` is `Station` and in most cases it has to be cast to the proper subtype.

Note: It is not checked whether the actual types of the process's father and its owning station agree with the types specified when the process type was defined (section 7.2). It is assumed that these types agree with those specified in the process type definition and the actual pointers are cast to the types declared for `S` and `F`. It is possible for processes of the same type to be spawned by processes (*fathers*) of different types. In such a case, the `F` pointer, if used, should be cast properly, except when the reference is to a generic attribute (e.g., a method) shared by all processes.

This `Process` method:

```
Station *getOwner ();
```

returns a pointer to the station owning the process. It is not very useful from within the process itself (`S` and `TheStation` point to the respective station), but can be used when you have a process pointer, e.g., referenced from another process.

A process can terminate itself by calling the following method:

```
void terminate ();
```

or by executing

```
terminate;
```

A process can also terminate another process by calling the function

```
terminate (Process *p);
```

or the method

```
p->terminate ();
```

A terminated process ceases to exist.

Note: If a process terminates itself (by calling `terminate ();`), any statements of its code method following the call to `terminate` are ignored.

It may happen that a terminated process has a non-empty set of descendants, which may include children processes and/or other exposable objects (section 17) that have been created by the process during its lifetime. Such objects are linked to the process and, in particular, show up as its descendants in the hierarchy of exposable objects (section 18.5.1). When a process whose descendant list is nonempty is being terminated, the descendants are moved to the process's father, i.e., the process originally responsible for its creation. Then, if the descendant is a process, its father (the `F` attribute) is set to the new owner. Note that the actual type of that owner need not agree with the type of `F` (section 7.2). Therefore, if such situations can legitimately arise at all, the program must take necessary precautions to avoid confusion.

A non-process descendant (e.g., `RVariable` or `EObject`) is simply moved to the new owner for the practically sole purpose of being locatable by `DSD` as an exposable object (section 18.5.1). Normally, a process that terminates itself would take care to clean up its dependents, although sometimes it may be difficult, especially when those descendants are processes. While it is illegal to use standard C++ constructors for a process, destructors are OK. A process that expects to be terminated unexpectedly can use its destructor to make sure that the unnecessary dependents and other dynamic objects are always cleaned up properly.

There exists a way to terminate all processes owned by a particular station. This is accomplished by executing `terminate ()` as a `Station` method, e.g., as in the following sequence:

```
...
 state Reset:
     S->terminate ();
     sleep;
    ...
```

The role of this operation is to provide for a way to reinitialize (reset) a station by completely eliminating its processes and starting from a clean slate. When a process is terminated this way, all its non-process descendants are deallocated, and its children processes are moved to `Kernel` (section 7.9) as its new father. This is usually very temporary: such children are typically owned by the same station, so they will be terminated as well; however, it is technically possible for a process running at station A to create a process at station B (e.g., by explicitly re-setting `TheStation`—section 6.1.3). Such tricks may lead to a confusion, so they should be applied with due care.

As the deallocation of descendants of a process killed by the `Station` variant of `terminate` follows the execution of the process's destructor, there is no danger if the process itself deallocates all or some of them upon termination.

Note that the process executing `S->terminate` will be terminated as well by this operation. As in many cases the operation must be followed by other statements, e.g., ones that start up a new set of processes, it does not automatically return from the process. One should remember that following `terminate` the current process is no more; in particular, any references to its methods and/or attributes will likely result in errors.

There is a way to locate processes by their type names and owning stations, which is seldom used, but provides a powerful last-resort means to find processes and account for them all, e.g., to terminate them as part of a cleanup operation, even if no explicit pointers to them are available. This operation:

```
Long getproclist (Station *s, ptype, Process **PL, Long Len);
```

stores in the array of pointers represented by the third argument the list of pointers to processes described by the first two arguments. The last argument gives the length of the array of pointers. The operation looks like a global function, but, strictly speaking, it isn't because the second argument is the name of a process type. The operation is intercepted by `smpp` (sections 1.5, 19.2) and transformed into a call to some internal and obscure function.

If the first argument is not `NULL`, then it should point to a station. The processes located by the operation will belong to that station. The second argument can be absent, in which case the operation will look like this:

```
Long getproclist (Station *s, Process **PL, Long Len);
```

and will return all processes owned by the station pointed to by the first argument. Finally, if the first argument is also `NULL`, it can be skipped like this:

```
Long getproclist (Process **PL, Long Len);
```

This practically never makes sense as the operation will then return pointers to all processes run by the simulator.

The array of pointers is filled up to the specified length (`Len`). The number of process pointers stored in it is returned as the function value. If that number is equal to `Len`, then there may be more processes to be returned. For illustration, here is the (not very efficient but effective) code to kill all copies of process `Consumer` run by the current station:

```
Process *PT [1];
  ...
while (getproclist (TheStation, Consumer, PT, 1))
    PT [0] -> terminate ();
```

A call to `getproclist` is rather costly as the operation may end up traversing a significant part of the entire object ownership hierarchy (section 18.5.1) before it locates all the relevant processes.

7.4 Process operation

Each process operates in a cycle consisting of the following steps:

1. the process is awakened by one of the awaited events
2. the process responds to the event, i.e., it performs some operations specified by the protocol
3. the process puts itself to sleep

Before a process puts itself to sleep, it usually issues at least one wait request (section 7.1)—to specify the event(s) that will wake it up in the future. A process that goes to sleep without specifying a waking event is terminated. The effect is the same as if the process issued `terminate` as its last statement (section 7.3).

To put itself to sleep, a process can execute the following statement:

```
sleep;
```

or simply exit by falling through the end of the list of statements at the current state (or by the closing brace of the code method).

By issuing a wait request (using the `wait` method of an *AI*—section 7.1), a process identifies the following four elements:

- the activity interpreter responsible for triggering the event
- the event category
- the process state to be assumed upon the occurrence of the event
- the *order* (priority) of the awaited event

The first three items are mandatory. The last element can be left unspecified and, in such a case, the default order of 0 is assumed. Note that the specified order can be negative, i.e., lower than the default.

If before suspending itself (i.e., going to sleep) a process issues more than one wait request, the multiple wait requests are interpreted as an **alternative** of waking conditions. It means that when the process becomes suspended, it will be waiting for a collection of event types, possibly coming from different *AI*s, and the occurrence of the **earliest** of

those events will resuscitate the process. When a process is restarted, its collection of awaited events is **cleared**, which means that in each operation cycle these events must be specified from scratch.

A process may undo its already issued collection of wait requests by calling this function:

```
void unwait ();
```

This operation is seldom useful and must be used with care, as some wait requests may be implicit (and they will be also undone by `unwait`). For example, undoing the implicit wait request issued by `transmit` (section 11.1.2) seldom makes sense.

It is possible that two or more events, from the collection of events awaited by the process, occur at the same time (within the same *ITU*). In such a case, the event with the lowest order is selected and this event actually restarts the process. If multiple events occurring at the same time have the same lowest order (e.g., the default order of 0), the waking event is chosen nondeterministically from among them.[20] In any case, whenever a process is awakened, it is always due to exactly one event.

Note: Many protocols can be programmed without assuming any ordering of events that can possibly occur at the same modeled time. Generally, the nondeterminism inherent in such scenarios is useful, as it accounts for the real-life phenomena like races and thus tends to enhance the model's fidelity. Therefore, for the sake of efficiency, the possibility to specify the third argument of `wait` is switched on by a special compilation option (`-p`) of `mks` (section 19.2). Without this option, `wait` accepts only two arguments and all events (except for the cases mentioned in sections 11.2.2 and 13.2.4) have the same order.

It is possible to force a completely deterministic processing of multiple events scheduled at the same *ITU*. This is accomplished by setting the `-D` option of `mks` (section 19.2). If this option is in effect, all events of the same order occurring at the same *ITU* are additionally ordered by the absolute timing of the corresponding wait requests (earlier awaited events receive lower order). This feature is usually selected together with the real mode of operation (option `-R`, section 19.2) to make sure that a program controlling a real physical system behaves in an absolutely deterministic way.

The state identifier specified as the second argument of a wait request is an enumeration object of type `int`. When the process is awakened (always by one of the awaited events), its code method is simply called (as a regular function) and the state identifier is presented in the global variable

```
int TheState;
```

which can be used by the process to determine what has actually happened. The code method uses the contents of `TheState` to determine at which state it is supposed to execute.

[20] Some exceptions are discussed in sections 11.2.2, 12.2.2 and 13.2.4.

A newly created process is automatically restarted within the *ITU* of its creation with the value of `TheState` equal 0. Value 0 corresponds to the **first symbolic state** declared with the `states` command (section 7.2). This first waking event is assumed to have arrived from the process itself (section 7.7). Its default order is 0; therefore, multiple processes created within the same *ITU* may be run in an unpredictable order. The following `Process` method can be used to make this order predictable:

```
void setSP (LONG order);
```

The method can be called by the creator immediately after executing `create` (which returns a process pointer) or, preferably, from the created process's `setup` method. The argument of `setSP` specifies the order of the first waking event for the new process. The program must have been created with `-p` for `setSP` to be accessible.

Owing to the sequential nature of the SIDE kernel, only one process can be active at a given moment of real time (SIDE does not resort to "external" multi-threading to implement its processes). However, the modeled time **does not flow** while a process is active, which means that multiple processes can be active simultaneously in the internal time. From the viewpoint of simulation, this means that the parallelism in the modeled system is correctly reflected in the model. The programmer exercises full control over time flow and explicit operations are required to force the time to be advanced. From the viewpoint of controlling real physical systems, this means that a SIDE process is viewed as a high-priority non-preemptive interrupt handler whose single grain of action is expected to be simple and fast.

7.5 The process code method

A process code method (section 7.2) is programmed in a way resembling a finite state machine described as a collection of states. An awakened process gets into a specific state and performs a sequence of operations associated with that state. By issuing wait requests, the process specifies dynamically the transition function, which tells it where to go from the current state upon the occurrence of interesting events. The standard layout of the process code method is this:

```
perform {
    state S0:
        ...
    state S1:
        ...
    state Sn-1:
        ...
};
```

Each `state` statement is followed by a symbol identifying one state. This symbol must be declared in the process's `states` list (section 7.2). The dots following a `state` statement represent instructions to be executed when the process wakes up in a given state. The interpretation of the contents of `TheState` is automatic: as soon as the process is awakened, control is switched to the proper `state` indicated by the value of `TheState`. When the list of instructions associated with a given state is exhausted, i.e., the code method attempts to fall through the `state` boundary, the process is automatically put to sleep.

Keyword `transient` can be used instead of `state` to allow the state to be entered directly from the preceding state. In other words, if the list of instructions of the state preceding a `transient` state is exhausted (and it does not end with `sleep`), the `transient` state will be entered and its instructions will be executed.

Example

Below we present a sample process declaration.

```
process AlarmClock {
  TIME delay;
  setup (TIME intvl) {
    delay = intvl;
  };
  states {Start, GoOff};
  perform {
    state Start:
      Timer->wait (delay, GoOff);
    state GoOff:
      WakeUp->put ();
      proceed Start;
  };
};
```

This process can be viewed as an alarm clock that sends out a signal every `delay` time units. The semantics of operations `Timer->wait`, `WakeUp->put`, and `proceed` are described in sections 8.1 and 13.2.3.

7.6 Process environment

By the *process environment* we mean a collection of variables readable by an active process and describing some elements of the process context or carrying some information related to the waking event. Two such variables: `TheStation` (section 6.1.3) and `TheState` (section 7.4) have already been discussed. Two more variables in this category are:

`Time`
: of type `TIME`. This variable tells the current internal time measured in *ITU*s from the beginning of execution.

`TheProcess`
: of type `Process`. This variable points to the object representing the process currently active. From the viewpoint of the process code method, the same object is pointed to by `this`.

Sometimes an event, besides just being triggered, carries some specific information. An example of such an event is a packet being heard on a port or a transceiver. The obvious information that should arrive to the awakened process along with this event is the pointer to the object representing the packet. There exist two general purpose pointers that are used by *AI*s to pass information associated with events or related to some inquiries. They are declared as

```
void *Info01, *Info02;
```

These pointers are practically never used directly. There exist numerous aliases (macros) that cast the above pointers to the proper types corresponding to the potential types of data items that can be returned by events. For example, the following standard macro provides an alias that can be perceived as a packet pointer:

```
#define ThePacket ((Packet*)Info01)
```

This and other aliases will be discussed at the individual *AI*s.

Note: The values returned by `Info01` and `Info02` (in their numerous aliases), are only guaranteed to be valid immediately after the respective event is received (i.e., the process wakes up in the corresponding state). Many operations on activity interpreters (that the process may subsequently want to execute) are likely to overwrite those variables, which are shared by all *AI*s.

7.7 Process as an *AI*

Type `Process` is a subtype of `AI` which suggests that a process can appear as an activity interpreter to another process (or even to itself). It is possible for a process, say P_1, to issue a wait request to another process, say P_2, to be awakened when P_2 enters a specific state. The first argument of `wait` in this case is an integer enumeration value identifying a process state. A few special (negative) values, represented by symbolic constants (e.g., `DEATH` defined as –1), are used to denote some special (non-state) events.

A process P_1 restarted by process P_2 entering the state specified in P_1's wait request addressed to P_2, is awakened **after** P_2 is given control (i.e., when P_2 puts itself to sleep after completing the state), but still within the same *ITU*. The two environment variables `Info01` and `Info02` (section 7.6) presented to P_1 look exactly as they were when P_2 was completing its state. Note that P_2 can modify them to pass a specific message to P_1.

Typically, when a process creates another process, the two processes run in parallel, at least from the viewpoint of the internal time. With the possibility of awaiting the termination of another process, one can easily implement a subroutine-like scenario in which a process creates another process and then waits for the termination of the child. For example, with the following sequence:

```
pr = create MyChild;
pr->wait (DEATH, Done);
sleep;
```

the current process spawns a child process and goes to sleep until the child terminates. When it happens, the parent will resume its execution in state `Done`.

In addition to `DEATH` of a specific process, a process can await the termination of any of its immediate children by issuing a wait request to itself with `CHILD` as the first argument. Here is a sample sequence to make sure that all your children have disappeared before proceeding:

```
...
 state WaitChildren:
     if (children () != 0) {
         wait (CHILD, WaitChildren);
         sleep;
     }
    ...
```

The `children` method of `Process` returns the number of processes appearing in the descendant list of the process. This can be viewed as the formal definition of the children set. There are circumstances (see section 7.3) when a process that has not been explicitly created by a given process may nonetheless show up as its descendant. Note that any process can await the termination of any process's child. For `DEATH`, as well as for `CHILD`, the two environment variables `Info01` and `Info02` (section 7.6) presented to the restarted process look as they were last seen by the process that has triggered the event.

Note: A process terminated by reset, i.e., the `terminate` method of its owning station (section 7.3), triggers neither `DEATH` nor `CHILD` events.

The first waking event for a process, triggered after the process's creation (section 7.4), is

assumed to have been generated by the process itself. This virtual event is called `START` and it cannot be awaited explicitly.[21]

It is legal for a process to wait for itself entering a specific state, although this possibility may seem somewhat exotic. For example, consider the following fragment of a code method:

```
...
state First:
    TheProcess->wait (Second, Third);
    proceed Second;
state Second:
    sleep;
state Third:
...
```

When the process wakes up in state `Second` (the semantics of `proceed` is described in section 8.1), it apparently dies, as it does not issue any wait requests in this state (section 7.4). However, in the previous state, the process declared that it wanted to get to state `Third` as soon as it got to `Second`. Thus, after leaving state `Second` the process will wake up in state `Third`—within the current *ITU*. The effect is as if an implicit `proceed` operation to state `Third` were issued by the process at `Second`, before this state is actually entered. Note that `proceed` is just a special case of a wait request (section 8.1) and, if the process issues other wait requests that are fulfilled within the current *ITU*, the implicit `proceed` may be ineffective, i.e., the process may actually end up in another state. The `order` argument of the state wait request can be used to assign a higher priority to this transition.

The range of the wait request issued in state `First` is just one state ahead. According to what we said in section 7.4, all pending wait requests are cleared whenever a process is awakened.

In the code fragment presented above, the call to `wait` at state `First` does not have to be preceded by `TheProcess->` as the wait request is addressed to the current process. It is recommended, however, to reference all *AI* methods with explicit remote access operators. Note that in the above example, `this` used instead of `TheProcess` would have the same effect.

[21] This event is needed to make sure that whenever a process is run, it always happens in response to some event.

7.8 Signal passing

A process can also be viewed as a repository for *signals*, which provide a simple means of process synchronization. More sophisticated inter-process communication tools are described in section 13.

7.8.1 Regular signals

In the simplest case, a signal is deposited at a process by calling the following process method:

```
int signal (void *s = NULL);
```

where the argument can point to additional information—to be passed along with the signal. If the argument is absent, no signal-specific information is passed: it is assumed that the occurrence of the signal is the only event of interest. Formally, the signal is deposited at the process whose `signal` method is called. Thus, by executing

```
prcs->signal ();
```

the current process deposits a signal at the process pointed to by `prcs`, whereas each of the following two calls:

```
signal (ThePacket);
TheProcess->signal ();
```

deposits a signal at the current process.

A deposited signal can be perceived by its recipient via wait requests addressed to the process *AI*. A process can declare that it wants to await the occurrence of a signal by executing the following method:

```
prcs->wait (SIGNAL, where);
```

where `prcs` points to the process at which the awaited signal is expected to arrive and `where` identifies the state where the waiting process wants to be awakened upon the signal arrival. Note that the signal does not have to be deposited at the waiting process. For example, it can be deposited at the sender or even at a third party which is neither the sender nor the recipient of the signal.

If the signal is pending at the moment when the wait request is issued (i.e., it has been deposited and not yet perceived), the waking event occurs within the current *ITU*. Otherwise, the event will be triggered as soon as a signal is deposited at the corresponding process.

When a process is awakened due to a signal event, the signal is removed from the repository. This only happens when the waking event is *actually* a signal event. Two environment variables are then set: `TheSender` (of type `Process*`) points to the process that has sent (deposited) the signal, and `TheSignal` (of type `void*`) returns the value of the argument given to `signal` by the sender. The two environment variables are aliases for `Info02` and `Info01`, respectively (section 7.6).

Note: It is possible that the process responsible for sending a signal is no longer present (it has terminated itself or has been terminated by some other process) when the signal is received. In such a case, `TheSender` will point to the `Kernel` process, which can be used as an indication that the signal sender has terminated. Generally, any event triggered by a process that terminates before the event is presented is assumed to have been triggered by `Kernel`.

Only one signal can remain pending at a process at a time, i.e., multiple signals **are not** queued. The `signal` method returns an integer value that tells what has happened to the signal. The following values (represented by symbolic constants) can be returned:

`ACCEPTED`
: The signal has been accepted and there is a process already waiting for the signal. The signal has been put into the repository and a waking event for the awaiting process has been scheduled. Note, however, that this event is not necessarily the one that will eventually wake up the process (the process may have other waking events scheduled at the same *ITU*).

`QUEUED`
: The signal has been accepted, but nobody is waiting for the signal at this moment. The signal has been put into the repository.

`REJECTED`
: The repository is occupied by a pending signal and the new signal has been rejected. The `signal` operation has no effect in this case.

It is possible to check whether a signal has been deposited and remains pending at a process. The following process method can be used for this purpose:

```
int isSignal ();
```

The method returns `YES` (1) if a signal is pending at the process, and `NO` (0) if the process's signal repository is empty. In the former case, the environment variables `TheSender` and `TheSignal` are set—as described above. The signal is left in the repository and remains pending. The following process method:

```
int erase ();
```

behaves similarly as `isSignal`, except that the signal, if present, is removed from the repository.

A process can also wait for a signal repository to become empty. This is accomplished with the following wait request:

```
prcs->wait (CLEAR, where);
```

where, as before, `prcs` identifies the owner of the repository. The `CLEAR` event is triggered by `erase`, but only if the repository was nonempty when `erase` was invoked.

The flexibility of selecting the signal repository makes it possible to create signal-based communication scenarios that are best suited for a particular application. For example, in a situation when a single process P_0 creates a number of child processes $P_1, \ldots, P_k$ and wants to pass a signal to each of them, it is natural for P_0 to deposit each signal at the corresponding child. On the other hand, if each of the child processes $P_1, \ldots, P_k$ is expected to produce a message for P_0, the most natural place for a child process to deposit such a signal is in its own repository. In the first case, the `signal` operation issued by P_0 will be addressed to a child and the child will issue a wait request to itself; in the second case, the child will execute its own `signal` method and P_0 will issue a signal wait request to the child.

7.8.2 Priority signals

Sometimes one would like to give a signal event a higher priority—to make sure that the signal is received immediately, even if some other waking events are scheduled at the same *ITU*.[22] One way to achieve this is to assign a very low order (section 7.4) to the wait request for the signal event. This solution requires a cooperation from the signal recipient. It is also possible for the signal sender to indicate that the signal should be received immediately, irrespective of the order of the corresponding wait request at the recipient. Such a communication mechanism can be interpreted as a "dangling branch" from one process to another and is called a *priority event*. A signal priority event is triggered by calling the following method:

```
int signalP (void *s = NULL);
```

where the meaning of `s` is the same as for the regular `signal` operation (section 7.8.1). Similar to `signal`, `signalP` deposits a signal at the process. The following additional rules apply to `signalP`:

[22] Similar problems are discussed in sections 10.9 and 13.2.4.

- At most one `signalP` operation can be issued by a process from the moment it wakes up until it goes to sleep. In fact, a somewhat stronger statement is true. Namely, at most one operation generating a priority event (including the `putP` operation described in section 13.2.4) can be issued by a process after it wakes up and before it suspends itself. In other words: at most one priority event can remain pending at any moment.

- At the moment when `signalP` is executed, the process's signal repository should be empty and there must be exactly one process waiting for the signal event. In other words: the exact deterministic continuation of the process generating the signal must be known at the moment when the `signalP` operation is issued.

The situation when nobody is waiting for a priority signal is not treated as a hard error, however. In such a case, `signalP` returns `REJECTED` and the operation has no effect. Otherwise, the function returns `ACCEPTED`.

As soon as the process that executes `signalP` is suspended, the waiting recipient of the signal is restarted. No other process is run in the meantime, even if there are some other events scheduled at the current *ITU*.

The recipient of a priority signal does not declare in any special way that it is awaiting a priority event: it just executes a regular signal wait request (section 7.8.1) with any (e.g., default) order. The fact that the transaction is to be processed as a priority event is indicated by `signalP`, and the order of the matching wait request is then irrelevant. In sections 10.9 and 13.2.4 we give examples of situations where priority events are useful.

7.9 Organization of a program in SIDE

From the user's perspective, all activities in SIDE are processes. One system process is created by SIDE immediately after the protocol program is run and exists throughout its entire lifetime. This process is pointed to by the global variable `Kernel` of type `Process` and belongs to the `System` station (section 6.1.2). Before the execution is started, `Kernel` creates one process of type `Root` which must be defined by the user. This process is the root of the user process hierarchy. The `Root` process is created with the empty `setup` argument list.

The `Root` process is responsible for building the network (section 6) and starting all initial protocol processes. Then it should put itself to sleep awaiting the end of execution which is signaled as the termination of `Kernel`. For a control program (as opposed to a simulator) the end of execution event may never be triggered.

The following structure of the `Root` process is recommended (at least for a typical simulator):

```
process Root {
```

```
        states {Start, Done};
        perform {
            state Start:
                read input data ...
                create the network ...
                define traffic patterns ...
                create protocol processes ...
                Kernel->wait (DEATH, Done);
            state Done:
                print output results
        };
    };
```

Of course, the particular initialization steps listed at state `Start` can be performed by separate functions or methods.

The termination of `Kernel` that indicates the end of the simulation run is apparent: in fact, `Kernel` never dies and, in particular, it can be referenced (e.g., exposed—see section 17.5.12) in state `Done` of the above `Root` process. The user can define a number of termination conditions for the run (section 15); it is also possible to terminate execution explicitly from a protocol process by executing

```
Kernel->terminate ();
```

i.e., by (apparently) killing the `Kernel` process.

Note: Some elements of the network configuration and traffic generator (the `Client`) are actually built not sooner than just before the protocol is started, specifically, after the user's root process goes to sleep at the end of initial state. Sometimes you may want to perform some operations from the root (before giving control to the protocol) assuming that the network has been fully built. It is possible to force the completion of the network setup (this makes only sense in the initial state of `Root`) by calling this function:

```
void buildNetwork ();
```

Following this, any operations that add new elements to the network, e.g., create new stations, links, traffic patterns, etc., are illegal.

There exists another special event that can only be awaited on the `Kernel` process. This event is `STALL` and it is triggered when the simulator runs out of events to process. All processes waiting for `STALL` on the `Kernel` will then be restarted within the current *ITU*. The `STALL` event is not available in the real mode or while the visualization mode is active.

7.10 The visualization mode

When a simulation experiment is run in this mode, the virtual timing of its events is synchronized to real time, such that the user can "visualize" the model's real-time behavior, possibly scaled down (slow motion) or up (accelerated motion). The visualization mode is available to any simulation program that has been compiled with `-W` (section 19.2). It must be explicitly entered by the program and can be exited at any time. When the visualization mode is *not* active, the simulator executes in a purely virtual manner, with time being entirely abstract and bearing no relationship to the execution time of the program.

The visualization mode is entered by calling this (global) function:

```
void setResync (Long msecs, double scale);
```

where `msecs` is the so-called resync interval (in milliseconds of real time) and `scale` maps that interval to virtual time. For example, if `msecs` is 500 and `scale` is 0.25, the resync interval is 500 milliseconds (0.5s) and that interval will map to 0.25 *ETU*s of simulated time. If *ETU* corresponds to 1 virtual second, the visualization will be carried out in slow motion with the factor of 2 (it will take 0.5s of real time to present 0.25s of virtual time).

They way it works is that every resync interval the simulator compares its virtual clock (variable `Time`—section 7.6) with the real time using the `scale` factor for conversion. If the virtual time tries to advance faster than real time, the simulator will hold (sleep internally) until the two get in sync. Note, however, that the program cannot do much if the real (execution) time advances faster than the simulated time. In such a case, the visualization will be poor or completely misleading.

Visualization can be stopped at any time by calling `setResync` with the first argument equal zero (the value of the second argument is then irrelevant). It can be resumed again, and so on, with no effect on the model, unless the model accepts input from the outside.

Similar to the real-time mode, it is possible to feed a visualized model with external data or make it generate data feeding external programs, e.g., through bound mailboxes (section 13.4). Owing to the fact that such a program pretends to run in real time, it can drive graphic displays and/or react with the environment as a make-believe replica of the real system. Needless to say, for this to be realistic, the real time execution of the simulator must be faster (or at least not slower) than the scaled progress of the model in virtual time.

8 The Timer *AI*

Although formally the `Timer` *AI* occurs in a single copy, it in fact provides an unlimited number of independent alarm clocks[23] that can be set and responded to by protocol processes. This way the processes can explicitly advance the internal notion of time. For simulation (running in the virtual mode) this advancement is logical; in the real mode, this operation involves actual sleeping for the prescribed amount of real time.

8.1 Wait requests

By issuing the following wait request:

```
Timer->wait (delay, where);
```

or

```
Timer->wait (delay, where, order);
```

where *delay* is an object of type `TIME`, the process declares that it wants to be awakened in state *where*, *delay ITU*s after the current moment.

The delay can also be specified in *ETU*s by using the following method instead of `wait`:

```
void delay (double etus, int state);
void delay (double etus, int state, LONG order);
```

`Timer->delay (e, st)` is strictly equivalent to `Timer->wait (etuToItu (e), st)` (section 3.6).

Implicit wait requests to the `Timer` *AI* are issued by some compound operations (e.g., see section 11.1.2). In particular, the following operation:

```
proceed newstate;
```

is used to transit directly (without awaiting any explicit event) to the state indicated by the argument. This is in fact a compound operation which is equivalent (with a minor exception explained below) to

```
Timer->wait (TIME_0, newstate, 0);
sleep;
```

[23] Those clocks may even appear unsynchronized, i.e., running at slightly different rates, to account for the limited accuracy of independent clocks in real life (section 8.3).

Thus, the transition is performed in response to an event that wakes up the requesting process within the current *ITU*.

Note that the default order[24] of the wait request issued by `proceed` is 0. Thus, if other (e.g., explicit) wait requests have been issued by the process, that can be fulfilled within the current *ITU*, they can take precedence over `proceed` and the operation may be ineffective. Moreover, other processes whose waking events have been scheduled for the current *ITU* can be run in the meantime. It is possible to assign an order to `proceed` by using one of the following two variants:

```
proceed newstate, order;
proceed (newstate, order);
```

where the second argument is an expression evaluating to a `LONG` integer number. This number will be used as the third argument of the implicit `Timer` wait request issued by the operation.

Another operation somewhat similar to `proceed` is

```
skipto newstate;
```

which is equivalent to

```
Timer->wait (TIME_1, newstate, 0);
sleep;
```

and performs a transition to state *newstate* with a one-*ITU* delay. This operation is useful for skipping certain events that persist until time is advanced (see section 11.2.3 for an example).

Similar to `proceed`, it is possible to assign an order to the wait request issued by `skipto`. The following two variants of the operation are available:

```
skipto newstate, order;
skipto (newstate, order);
```

where the meaning of the second argument is as for `proceed`.

The two-argument variants of `proceed` and `skipto` are only available if the SIDE program has been created with the `-p` option of `mks` (section 19.2).

If you really want to transit to a given state immediately, with absolute certainty that nothing will happen in the meantime, use this operation:

```
sameas newstate;
sameas (newstate);
```

[24] See sections 7.1 and 7.4.

where the second argument is the symbolic state identifier. This operation actually executes a direct branch to the beginning of the state body. Before that happens, `TheState` is set to the new state number. The environment variables `Info01` and `Info02` (section 7.6) remain unchanged.

Note: When a process is restarted by a `Timer` event, the two environment variables `Info01` and `Info02` are set to `NULL`, i.e., timer events carry no environment information. When a process wakes up at the target state of `proceed`, the environment variables contain their previous values—as they were at the moment when `proceed` was executed.

8.2 Operations

Issuing a `Timer` request (section 8.1) can be viewed as setting up an alarm clock or a timer. Unless the process is awakened by an earlier event, the alarm clock will make sure to restart the process after the specified delay.

A process can inquire about an alarm clock set up by another process and also reset that alarm clock to a different delay. For this purpose, alarm clocks are identified by their target states. The following method:

```
Timer->getDelay (process, state);
```

returns the delay of the timer set up by the indicated process that, when (and if) it goes off, will move the process to the indicated state. If no such alarm clock is currently defined by the process, the method returns `TIME_inf` (section 3.5). Note that the value returned upon success gives the current (i.e., remaining) delay of the timer, rather than the original value given when the alarm clock was set.

A timer can be reset with the following operation:

```
Timer->setDelay (process, state, value);
```

where the third argument is an object of type `TIME` specifying the new delay value. The method will succeed (and return `OK`) only if an alarm clock restarting the process in the indicated state is already defined by the process. Otherwise, the method returns `ERROR` and has no further effect. This means that one cannot introduce new timers this way, i.e., an alarm clock can be reset from the outside of a process but it cannot be set up.

The `state` argument of `getDelay` and `setDelay` must specify a symbolic state name of the process indicated by the first argument. The first argument must be a process pointer of the proper type, i.e., it cannot be a generic `Process` pointer, because the state name can only be interpreted within the context of a specific process type.

Note that if the original alarm clock was set with the variant of `wait` specifying the order of the timer event (section 7.1), this order will be retained when the alarm clock is reset by `setDelay`.

It is possible, although generally redundant and useless, to issue from a single process multiple `Timer wait` requests specifying the same target state. Of course, the earliest of those requests is the only one that makes sense, while the remaining ones are useless and will be ignored. Notably, `getDelay` and `setDelay` work correctly in such circumstances. Namely, `getDelay` returns the delay of the earliest of the multiple timers, and `setDelay` sets them all to the new specified interval (which has the net effect of reducing all the multiple timers to a single one).

8.3 Clock tolerance

There exists only one global `Timer` *AI*, which would suggest that all stations in the network use the same notion of time and their clocks run in a perfectly synchronized fashion. This may make sense in a control program driving a real physical system (where the different stations represent some conceptual blocks of a real network), but is not very realistic in a simulation model (where the different stations represent independent components of some modeled hardware). In a realistic simulation model of a physical system, we often need to simulate the limited accuracy of local clocks, e.g., to study race conditions and other phenomena resulting from imperfect timing. Thus, SIDE makes it possible to specify the *tolerance* for clocks used by different stations. This idea boils down to randomizing slightly all delays specified as arguments of `Timer` wait requests issued by processes belonging to a given station. This randomization also affects implicit wait requests to the `Timer`, e.g., issued by `transmit` and `sendJam` (sections 11.1.2 and 12.1.8).

Clock tolerances can be defined globally during the network initialization phase (section 6) before creating stations. Each call to the function

```
setTolerance (double deviation = 0.0, int quality = 2);
```

(re)defines the global clock tolerance parameters, and the new definition will affect all stations created henceforth, until it is overridden by a new definition, i.e., another call to `setTolerance`. Specifying `deviation`=0 results in zero tolerance, i.e., absolutely accurate clocks. This is the default assumed before `setTolerance` is called for the first time.

It is also possible to define the clock tolerance individually for a given station by calling the above function as a `Station` method. There exists a reverse function (and its method variant) defined like this:

```
double getTolerance (int *quality = NULL);
```

which returns the current tolerance setting (global or for an individual station). If the argument is not NULL, the quality parameter is stored under the specified pointer.

With a nonzero clock tolerance in effect, the actual delay used by the `Timer` may differ from the requested delay d_r by up to `deviation`$\times d_r$ rounded to full ITUs. The actual deviation,

which can be either positive or negative, is determined according to the distribution

$$\beta(\texttt{quality}, \texttt{quality})$$

(see section 4.1).

9 The Monitor *AI*

Similar to `Timer`, the `Monitor` *AI* occurs in a single instance pointed to by the global constant `Monitor`. It provides a crude, but effective and cheap, inter-process synchronization tool. The idea is extremely simple: wait requests identify events by numbers. When signaled, the event will wake up all processes waiting for it, globally, regardless of their ownership and other properties.

The primary role of this simple tool is to offer an efficient mechanism to synchronize processes operating "behind the scenes" of the actual protocol program, e.g., library modules, complex traffic generators or absorbers. Its true object-oriented counterpart, recommended for higher-level solutions, is provided by barrier mailboxes (section 13.3).

9.1 Wait requests

There is a single type of wait request that identifies the awaited event. It looks like this:

```
Monitor->wait (event, where);
```

or

```
Timer->wait (event, where, order);
```

depending on whether the program has been compiled without or with `-p` (section 7.4). The type of the *event* argument is `void*`. This is because it is natural to use addresses of the "critical" variables or data structures as event identifies. Note that those identifiers are global, so the uniqueness and consistency of events intentionally restricted to the context of, say, a single station can only be maintained by using different numbers. To this end, SIDE defines this macro:

```
#define MONITOR_LOCAL_EVENT(o) ((void*)(((char*)TheStation)+(o)))
```

to be used for creating event identifiers intended to be local with respect to the current station (section 6.1.3). For example, by declaring the following apparent constants:

```
#define EVENT_XMIT      MONITOR_LOCAL_EVENT(0)
#define EVENT_RCV       MONITOR_LOCAL_EVENT(1)
#define EVENT_DONE      MONITOR_LOCAL_EVENT(2)
```

you can use them as event identifiers, which will actually represent different numbers within the context of different stations—at least as long as the argument of the macro is not larger than the size of the station object.

A process issuing a wait request to `Monitor` is suspended until the event is delivered (signaled) by some other process. A signaled event is never queued: the receiving process must be already waiting for the event in order to perceive it.

9.2 Signaling monitor events

The operation of triggering a monitor event looks like sending a signal to a process, except that the signal is addressed to the monitor. This is accomplished by calling:

```
Monitor->signal (event);
```

where the type of `event` is `void*` (it matches the *event* argument of `wait`—section 9.1). All the processes waiting for the specified event number will be restarted within the current *ITU*. The method returns an `int` number of processes that have been waiting for the signal. In particular, the value 0 means that the operation was void.

When a process is resumed due to the occurrence of a monitor event, its environment variable `TheSignal` (an alias for `Info02`—section 7.6) returns the event number, while `TheSender` (`Info01`) points to the process that has sent the signal (executed the `signal` operation).

Similar to priority signals for processes (section 7.8.2), this operation:

```
Monitor->signalP (event);
```

sends the signal as a priority event. The semantics is exactly as described in section 7.8.2. The method returns `ACCEPTED`, if there was a (single) process awaiting the signal, and `REJECTED` otherwise.

10 The `Client` *AI* and `Traffic` *AI*s

The `Client` *AI*, which can be viewed as a union of all `Traffic` *AI*s, is the agent responsible for providing the network with traffic. It is used exclusively in simulators, possibly including some simulated components of physical systems controlled by SIDE programs.

10.1 General concepts

The traffic in a modeled network consists of *messages* and *packets*. A message represents a logical unit of information that arrives to a station from outside and must be sent to

some destination. From the viewpoint of the protocol, a message is characterized by the sender (i.e., the station at which the message arrives for transmission), the receiver (i.e., the station to which the message is to be sent), and the length (i.e., the number of bits).

The process of message arrival is defined by the user as a collection of *traffic patterns*. Each traffic pattern is described by the distribution of senders and receivers, and the distribution of message interarrival time and length. A single simulation experiment may involve multiple and diverse traffic patterns.

Each traffic pattern is managed by a separate activity interpreter called a `Traffic`. This *AI* is responsible for generating messages according to the distribution parameters specified by the user, queuing them at the sending stations, responding to *inquiries* about those messages, and triggering waking events related to the message arrival process. The `Client` *AI* is in fact a union of all `Traffic` *AI*s. For illustration, assume that a protocol process *P* at some station *S* wants to check whether there is a message queued at *S* that has been generated according to traffic pattern *T*. This kind of inquiry will be addressed to the specific traffic pattern (`Traffic` *AI*). By issuing the same kind of inquiry to the `Client` *AI*, *P* will check whether there is **any** message (belonging to any traffic pattern) awaiting transmission at *S*. Similarly, *P* may decide to go to sleep awaiting the arrival of a message of some specific pattern, or it may await any message arrival. In the former case, the wait request will be addressed to the respective `Traffic` object, which will be also responsible for delivering the awaited event. In the latter case, the wait request will be addressed to the `Client`, and the waking event will also arrive from that *AI*.

Each `Traffic` *AI* has its private message queue at every user-defined station. A message arriving to a station is queued at the end of the station's queue handled by the `Traffic` *AI* that has generated the message. Thus, the order of messages in the queues reflects their arrival time.

If the standard tools provided by SIDE for defining the behavior of a `Traffic` *AI* are insufficient to describe a refined traffic pattern, the user can program this behavior as a collection of dedicated processes.

Before a message can be transmitted over a communication channel (link), it must be turned into one or more *packets*. A packet represents a physical unit of information (a frame) that, besides information bits inherited from a message, usually carries some additional information required by the protocol (header, trailer, etc.).[25]

A typical execution cycle of a protocol process that takes care of transmitting packets at a station consists of the following steps:

1. The process checks if there is a message to be transmitted and if so, acquires a packet

[25]This interpretation assumes that messages and packets are used to model traffic in a communication network. It is possible to use messages and packets to model other things, e.g., boxes traveling through conveyor belts. In such scenarios, the headers and trailers (as well as some other message/packet attributes) may be irrelevant and they need not be used.

from that message and proceeds to 3.

2. If there is no message awaiting transmission, the process issues a wait request to the `Client` or some specific `Traffic` *AI*—to await a message arrival, and puts itself to sleep. When the waking event is triggered, the process wakes up at 1.

3. The process obeys some rules required by the protocol and transmits the packet.

4. When the transmission is complete, the process continues at 1.

The operation of acquiring a packet for transmission consists in turning a message (or just a portion of it) into a packet. It may happen that the protocol imposes some rules as to the maximum (and/or minimum) packet length. In such a case, a single long message may have to be split into several packets before it can be transmitted to the destination. This also applies to many nonstandard interpretations of messages and packets, e.g., a message representing a truckload of boxes (packets) to be transported via a conveyor belt.

SIDE collects some standard performance measures, separately for each `Traffic` and globally for the `Client` (i.e., for all traffic patterns combined). The user is able to provide private performance measuring functions that either augment or replace the standard ones.

10.2 Message and packet types

SIDE provides two standard base types `Message` and `Packet` that can be used to define protocol-specific message and packet types. Note that these types are not subtypes of *Object* (section 5.1). In many cases, `Message` and `Packet` need not be extended by the user: they are usable directly.

Message types are defined according to the rules described in sections 5.3 and 5.4. The declaration of a message type starts with the keyword `message`.

Example

The following declaration:

```
message RMessage {
    int Route [N], RouteLength;
};
```

defines a message type called `RMessage` which is derived directly from the base type `Message`.

Messages are very seldom generated "by hand" in the protocol program, although, in principle, it is legal. In such an unlikely case, the message type may declare a `setup`

method and the regular `create` operation can be used to generate an object of the message type. Note that the version of `create` that assigns a nickname to the created object cannot be used to build a message (or a packet) as these objects have no *nicknames* (section 5.2). The default `setup` method defined in `Message` takes no arguments and its body is empty.

The declaration of a new packet type obeys the standard set of rules (sections 5.3 and 5.4) and starts with the keyword `packet`.

Example

The following declaration defines a packet type that may correspond to the message type `RMessage` declared above:

```
packet RPacket {
    int Route [N], RouteLength;
    void setup (RMessage *m) {
        for (RouteLength = 0; RouteLength < m->RouteLength;
          RouteLength++)
            Route [RouteLength] = m->Route [RouteLength];
    };
};
```

The `setup` method copies the `Route` attribute of the message from which the packet is built into the corresponding packet attribute. The interpretation of the packet `setup` method that specifies a `Message` subtype pointer as its argument type is special (see section 10.8.1).

Typically, packets are created automatically from messages by the `Client` when a protocol process acquires a new packet. Sometimes it is desirable to create a *non-standard* packet directly. In such a case, the regular `create` operation (without a nickname) can be used to explicitly create a packet object. For such occasions, a packet type may declare a `setup` method (or a collection of `setup` methods) to initialize its attributes upon creation.

For standard packets, i.e., those built automatically by the `Client`, one `setup` method has a special meaning (see section 10.8.1). Namely, a user-defined packet type being a non-trivial extension of `Packet` (i.e., specifying some additional attributes) **must** declare the following setup method:

```
void setup (mtype *m);
```

where `mtype` is a message type, i.e., a subtype of `Message`. This method will be called whenever a packet is acquired from a message—to set the non-standard attributes of the packet. The argument `m` will then point to the message from which the packet is extracted.

The standard type `Message` declares a number of `public` attributes that may be of interest to the user, as they are sometimes useful for programming private traffic generators. Below we list the `public` attributes of `Message`.

```
Message    *next, *prev;
Long       Length;
IPointer   Receiver;
int        TP;
TIME       QTime;
```

Attributes `next` and `prev` are used to store the message in a queue at the sending station. They have been made publicly visible to facilitate programming of non-standard tools for putting messages into queues, extracting them, and turning them into packets. The `next` attribute points to the next message in the queue and `prev` points to the previous message in the queue. For the last message in the queue, `next` contains `NULL`. The `prev` attribute of the first message in the queue points to a nonexistent dummy message whose `next` attribute coincides with the queue pointer (see section 10.7).

The `Length` attribute gives the message length. For the standard interpretation of messages and packets, this length is in bits.[26] The `Length` of a message generated by the standard `Client` is guaranteed to be a strictly positive multiple of 8.

Attribute `Receiver` contains the `Id` (section 6.1.2) of the station to which the message is addressed. For a broadcast message (sections 10.5, 10.6.2) `Receiver` points to a data structure representing a *group* of stations. One special value of the `Receiver` attribute is `NONE`: it means that the message is not explicitly addressed to any particular station.

`TP` is the `Id` attribute of the traffic pattern (the `Traffic` *AI*) that has created the message (see section 10.6.2). The attribute is set automatically for messages generated in the standard way. It is recommended to use negative `TP` values for messages that are generated explicitly (by `create`)—to avoid confusing them with messages generated by the `Client`.

`QTime` tells the time when the message was generated and queued at the sender. This attribute is used for calculating *message delay* (see section 14.2.1).

The public part of the `Packet` type declaration contains the following attributes:

```
Long       ILength, TLength,
           Sender;
IPointer   Receiver;
int        TP;
TIME       QTime, TTime;
FLAGS      Flags;
```

[26] For example, the packet length determines the amount of time needed to transmit the packet, i.e., insert it into a port with a given transmission rate (see section 11.1.2).

Attributes Receiver, QTime, and TP are directly inherited from the message that the packet was acquired from. ILength contains the length of the *information part* of the packet, i.e., the part that comes from the message and carries "useful" information. This part is also called the packet's *payload*. TLength is always greater than or equal to ILength and gives the *total length* of the packet, including the possible header and trailer required by the protocol. Both these lengths are in bits (the same units as the message length).

The Sender attribute contains the Id of the station sending the packet. Note that for a message, such an attribute would be redundant, as a message is always queued at a specific station, and only that station can be the sender of the message. On the other hand, a packet propagates among stations (often making several hops on its path to the destination), which makes the issue of identifying its sender less trivial.

TTime stands for *top time* and indicates the moment when the packet became ready for transmission (section 10.3). This time is used for measuring *packet delay* which excludes the message queuing time (section 14.2.1).

Attribute Flags represents a collection of binary values describing various elements of the packet *status*. Five bits (27–31) have a predefined meaning and are used by SIDE: they are discussed elsewhere. Other flags are left for the user. Symbolic constants PF_usr0, . . . , PF_usr26 representing integer values from 0 to 26 are recommended for accessing these flags (see section 4.3).

Non-standard packets, i.e., packets created outside the traffic patterns and the Client, should have their TP attributes negative or greater than the maximum valid traffic pattern Id (see section 10.6.2). These two Packet methods:

```
int isStandard ();
int isNonstandard ();
```

tell a standard packet from a non-standard one. A packet is assumed to be standard (isStandard returns 1 and isNonstandard returns 0), if it has been obtained from a message generated by one of the traffic patterns (i.e., if its TP attribute contains a valid traffic pattern Id). Otherwise, the packet is considered non-standard and isNonstandard returns 1, while isStandard returns 0.

Note: If the protocol program has been created with the -g (or -G) option of mks (debugging—see section 19.2), each packet carries one additional attribute of type Long called Signature. For packets acquired from the Client (via getPacket—see section 10.8.1) this attribute is set to consecutive integer values, starting from 0, which are different for different packets. Intentionally, packet Signatures are to be used for tracing packets, e.g., with observers (section 16.3). The Signature attribute of a non-standard packet is NONE (if needed, it should be set explicitly by the protocol program).

10.3 Packet buffers

Typically, when a protocol process wants to acquire a packet for transmission, it polls the `Client` or one of the `Traffic` *AI*s. If a suitable message is queued at the station, the *AI* will build a packet and return it to the process. The packet is stored into a buffer provided by the polling process (section 10.8.1).

Packet buffers are associated with stations. A packet buffer is just an object of a packet type (i.e., a subtype of `Packet`—section 10.2) declared statically within a station. For example, in the following declaration of station type `MySType`:

```
packet MyPType {
    ...
};
...
station MySType {
    ...
    MyPType Buffer;
    ...
};
```

`Buffer` is declared as a packet buffer capable of holding packets of type `MyPType`.

Packet buffers declared statically within the station class are viewed as its "official" buffers and are exposed by the station's buffer exposure (section 17.5.13). In principle, it is legal, although not recommended, to use as a packet buffer a packet created dynamically (by the `create` operation). When a packet buffer is created this way, the setup argument list should be empty.

Note: It is illegal to declare a packet structure (not a packet pointer) statically outside a station.

A packet buffer can be either *empty* or `full` which is determined by the contents of the `PF_full` flag (bit number 29) in the buffer's `Flags` attribute (section 10.2). This flag is irrelevant from the viewpoint of the packet contents: its meaning is restricted to the interpretation of the packet object as a buffer. This minor difference between a packet buffer and the packet itself did not seem to warrant introducing a separate data type—just for representing packet buffers and nothing else. Two simple `Packet` methods

```
int isFull () {
    return (flagSet (Flags, PF_full));
};
int isEmpty () {
    return (flagCleared (Flags, PF_full));
};
```

provide predicates determining whether a packet buffer is full or empty without explicitly examining the PF_full flag. There exist other methods for examining other flags of packets which, for all practical purposes, completely disguise the packet flags under a more friendly interface.

In most cases, a packet buffer is filled by calling getPacket (see section 10.8.1); then, the buffer's PF_full flag is automatically set. The standard way to empty a packet buffer, say pointed to by buf, is to call

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

The purpose of release is to mark the buffer as empty and to update certain performance measures (section 14.2.1). It should be called as soon as the contents of the buffer are no longer needed, e.g., after the packet has been completely transmitted, dispatched, acknowledged, discarded, etc., as required by the protocol.

There exist two Client versions of release which perform exactly the same action as the Packet method. The above call is equivalent to

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

or

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

i.e., the second version takes a structure as the argument rather than a pointer.

Finally, similar two methods are available from traffic patterns. If TPat points to a traffic pattern, then each of the following two calls:

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

is equivalent to buf->release (). As an added value, they provide assertions to verify that the buffer being released holds a packet belonging to the traffic pattern pointed to by TPat.

It is possible to fill the contents of a packet buffer explicitly, without acquiring a packet from the Client. The following Packet method can be used for this purpose:

```
void fill (Station *s, Station *r, Long tl, Long il = NONE,
       Long tp = NONE)
```

The first two arguments point to the stations to be used at the packet's sender and the receiver, respectively. The third argument will be stored in the TLength attribute: it

represents the total length of the packet. The next, optional, argument `il` gives the length of the packet's information part. If unspecified (or equal `NONE`), it is assumed that the information length (attribute `ILength`) is the same as the total length. It is checked whether the information length, if specified, is not bigger than the total length. Finally, the last argument will be stored in the `TP` attribute of the packet. It cannot be equal to the `Id` of any defined traffic pattern: it is illegal for a non-standard packet to pose for one that has been generated by the `Client`.

There exists an alternative version of `fill` which accepts station `Id`s instead of pointers as the first two arguments. Any of the two station `Id`s (or both) can be `NONE`. If the receiver's `Id` is `NONE`, it means that the packet is not explicitly addressed to any station.

When saving a packet for later use (e.g., after its reception—sections 11.2.3 and 11.2), the program should create a copy of the packet structure, rather than saving a pointer to the original structure, which will be deallocated when the packet formally disappears from the channel (see sections 11.2 and 12.2.1). Declaring static packet structures (to accommodate packet copies) may be somewhat tricky because such a structure tends to be interpreted as a packet buffer (and receives a special treatment—see above). An easy way to copy (clone) a packet structure is to call the following `Packet` method:

```
Packet *clone ();
```

which creates an exact replica of the packet and returns its pointer. Of course, this structure can (and should) be deallocated (by `delete`) when it is no longer needed.

10.4 Station groups

Sometimes a subset of stations in the network must be distinguished and identified as a single object. For example, to create a broadcast message, i.e., a message addressed to more than one recipient, one must be able to specify the set of all receivers as a single entity. A *station group* (an object of type `SGroup`) is just a set of station identifiers (`Id` attributes) representing a (not necessarily proper) subset of all stations in the network. An `SGroup` is created in the following way:

```
sg = create SGroup ( setup arguments );
```

(the nickname version of `create` is not applicable), where the setup arguments can be specified according to one of the following patterns:

`()` (i.e., no arguments)

The communication group created this way contains all stations in the network.

```
(int ns, int *sl)
```

If the first argument (ns) is greater than zero, it gives the length of the array passed as the second argument. This array contains the Ids of the stations to be included in the group. If the first argument is negative, its negation gives the length of sl, which is assumed to contain the list of exceptions, i.e., the stations that are not to be included in the group. In such a case, the group will consist of all stations in the network, except those mentioned in sl.

Note: The array of station Ids pointed to by sl can be deallocated after the station group has been created. The station group does not rely on its permanence.

```
(int ns, int n1, ...)
```

The first argument specifies the number of the remaining arguments, which are station Ids. The stations whose Ids are explicitly listed as arguments are included in the group.

```
(int ns, Station *s1, ...)
```

As above, but pointers to station objects are used instead of the numerical Ids.

The internal layout of SGroup is not interesting to the user: there is no need to reference SGroup attributes directly. In fact, explicitly created station groups are used quite seldom.

The following two methods are defined within SGroup:

```
int occurs (Long sid);
int occurs (Station *sp);
```

which return YES (1) if the station indicated by the argument (either as an Id or as an object pointer) is a member of the group.

Stations within a station group are ordered (this ordering may be relevant when the station group is used to define a communication group—section 10.5). If the station group has been defined by excluding exceptions or by using the empty setup argument list, the ordering of the included stations is that of the increasing order of their Ids. In all other cases, the ordering corresponds to the order in which the included stations have been specified.

Note: A station that has been specified twice counts as two separate elements. It makes no difference for determining its membership in the group, but is significant when the group is used to define a communication group (see section 10.5).

10.5 Communication groups

Communication groups (objects of type CGroup) are used (explicitly or implicitly) in definitions of refined traffic patterns, i.e., objects of type Traffic or its subtypes (see

section 10.6). A communication group consists of two station groups (section 10.4) and two sets of numerical weights associated with these groups. One group is called the *senders group* and contains the stations that can be potentially used as the senders of a message. Together with its associated set of weights, the senders group constitutes the *senders set* of the communication group. The other group is the *receivers group* and contains the stations that can be used as the receivers. The receivers group together with its associated set of weights is called the *receivers set* of the communication group.

A communication group can be either a *selection group*, in which the station group of receivers specifies individual stations and a message can be addressed to one of those stations at a time, or a *broadcast group*, in which the receivers group identifies a set of receivers for a broadcast message. A message whose generation has been triggered by such a communication group (see below) is a broadcast message addressed simultaneously to all the stations specified in the receivers set. No weights are associated with the receivers of a broadcast communication group.

To understand how the concept of communication groups works, let us consider a traffic pattern T based on a single communication group G (many traffic patterns are in fact defined this way). Let G be a selection group and

$$S = \langle (s_1, w_1), \ldots, (s_k, w_k) \rangle$$

be the set of senders of G. Similarly, by

$$R = \langle (r_1, v_1), \ldots, (r_p, v_p) \rangle$$

let us denote the set of receivers of G. Assume that a message is generated according to T. Among a number of attributes of this message that must be determined are two station identifiers indicating the sender and the receiver. The sender is chosen at random in such a way that the probability of a station

$$s_i : (s_i, w_i) \in S$$

being selected is equal to

$$P_S(s_i) = \frac{w_i}{\sum_{l=1}^{k} w_k}.$$

Thus the weight of a station in the senders set specifies its relative frequency of being used as the sender of a message generated according to T.

Once the sender has been determined (suppose it is station s), the receiver of the message is chosen from the receiver set R in such a way that the probability of station r_i being selected is equal to

$$P_R(r_i) = \frac{h_i}{\sum_{l=1}^{p} h_l},$$

where

$$h_i = \begin{cases} v_i & \text{if } r_i \neq s \\ 0 & \text{if } r_i = s \end{cases} .$$

In simple words, the receiver is determined at random in such a way that the chances of a particular station from the receiver set for being selected are proportional to its weight. However, the sender is excluded from the game, i.e., it is guaranteed that the message is addressed to another station. Note that the above formula for P_R makes no sense when the sum in the denominator is zero. In such a case, SIDE is not able to determine the receiver and the simulation is aborted with a pertinent error message.

Should G be a broadcast group, the problem of selecting the receiver would be trivial, namely, the entire receiver group would be used as a single object (see section 10.2) whose pointer would be stored in the `Receiver` attribute of the message.

The following operation explicitly creates a communication group:

```
cg = create CGroup ( setup arguments );
```

Similar to `SGroup`, the nickname variant of `create` (section 5.8) cannot be used for a `CGroup` object. The following configurations of setup arguments are allowed:

```
(SGroup *s, float *sw, SGroup *r, float *rw)
(SGroup *s, float *sw, SGroup *r)
(SGroup *s, SGroup *r, float *rw)
(SGroup *s, SGroup *r)
```

The first (most general) configuration specifies the senders group (`s`), the senders weights (array `sw`), the receivers group (`r`), and the receivers weights (array `rw`). The weights are assigned to individual members of the corresponding station groups according to their ordering within the groups (see section 10.4). Note that if a station appears twice (or more times) within a group, it counts as two (or more) stations and two (or more) entries are required for it in the weight array. The effect is as if the station occurred once with the weight equal to the sum of weights associated with all occurrences.

With the remaining configurations of the setup arguments, one set (or both sets) of weights are not specified. In such a case, each station in the corresponding station group is assigned the same weight of $1/n$ where n is the number of stations in the group.

The following two setup configurations are used to create a broadcast communication group:

```
(SGroup *s, float *sw, SGroup *r, int b)
(SGroup *s, SGroup *r, int b)
```

The value of the last argument can only be `GT_broadcast` (–1). Note that receivers weights are not specified for a broadcast communication group. With the second configuration, the default weights (of $1/n$) are assigned to the senders.

Note: The station groups passed as arguments to `create` must **not** be deallocated for as long as the communication group is being used.

10.6 Defining traffic patterns

As with other *Objects*, it is possible to define subtypes of `Traffic`; these subtypes are then used to create their specific instances called traffic patterns.

10.6.1 Defining traffic types

A definition of a traffic pattern type obeys the rules described in sections 5.3 and 5.4, and has the following format:

```
traffic ttype : itypename ( mtype, ptype ) {
    ...
    attributes and methods
    ...
};
```

where `mtype` and `ptype` are the names of the message type and the packet type (section 10.2) associated with the traffic pattern. All messages generated by the traffic pattern will belong to type *mtype* and all packets acquired from this traffic pattern will be of type *ptype*. If these types are not specified, i.e., the parentheses together with their contents are absent, it is assumed that they are `Message` and `Packet`, respectively. If only one identifier appears between the parentheses, it determines the message type; the associated packet type is then assumed to be `Packet`.

The base traffic type `Traffic` can be used directly to create traffic patterns. Messages and packets generated by a traffic pattern of type `Traffic` belong to the basic types `Message` and `Packet`. A new traffic type definition is only required if at least one of the following statements is true:

- the behavior of the standard traffic generator has to be changed by replacing or extending some of its methods
- the types of messages and/or packets generated by the traffic pattern are proper subtypes of `Message` and/or `Packet`, i.e., at least one of these types is not basic

The list of *attributes and methods* of a user-defined traffic type may contain redeclarations of some virtual methods of `Traffic` and possible declarations of non-standard variables used by these new methods. Essentially, there are two types of `Traffic` methods than can be overridden by the user's declarations: the methods used in generating messages and transforming them into packets (section 10.6.3), and the methods used for calculating performance measures (section 14.2.3).

10.6.2 Creating traffic patterns

Traffic patterns should be built by the user's root process (section 7.9) after the population of stations has been fully defined. A single traffic pattern is created by the `create` operation, in the way described in section 5.8.

The standard setup methods built into `Traffic` accept the following configurations of arguments:

```
(CGroup **cgl, int ncg, int flags, ...)
(CGroup *cg, int flags, ...)
(int flags, ...)
```

The first configuration is the most general one. The first argument (`cgl`) points to an array of pointers to communication groups (section 10.5). This array is assumed to contain `ncg` elements. The communication groups describe the distribution of senders and receivers for messages generated according to the created traffic pattern.

The second configuration specifies only one communication group and provides a shorthand for the situation when the array of communication groups contains exactly one element.

Finally, with the third configuration, the distribution of senders and receivers is not specified at the moment of creation (it can be specified later—see below) and the definition of the traffic pattern is incomplete.

The `flags` argument is a collection of binary options that select the type of the message arrival process and indicate whether standard performance measures (see section 14.2) should be calculated for this traffic pattern. The actual argument passed as `flags` should consist of a sum (logical or arithmetic) of symbolic constants selected from the following list:

`MIT_exp`
: message interarrival time is exponentially distributed

`MIT_unf`
: message interarrival time is uniformly distributed

`MIT_fix`
: message interarrival time is fixed

`MLE_exp`
: message length is exponentially distributed

`MLE_unf`
: message length is uniformly distributed

`MLE_fix`
: message length is fixed

`BIT_exp`
: traffic pattern is *bursty* and the burst interarrival time is exponentially distributed

`BIT_unf`
: traffic pattern is bursty and the burst interarrival time is uniformly distributed

`BIT_fix`
: traffic pattern is bursty and the burst interarrival time is fixed

`BSI_exp`
: burst size is exponentially distributed (relevant only if one of `BIT_exp`, `BIT_unf`, or `BIT_fix` has been specified)

`BSI_unf`
: burst size is uniformly distributed (relevant only if one of `BIT_exp`, `BIT_unf`, or `BIT_fix` has been specified)

`BSI_fix`
: burst size is fixed (relevant only if one of `BIT_exp`, `BIT_unf`, or `BIT_fix` has been specified)

`SCL_on`
: standard client processing for this traffic pattern is switched **on**, i.e., the traffic pattern will be used automatically to generate messages and queue them at the sending stations (this is the default)

`SCL_off`
: standard client processing is switched **off**, i.e., the traffic pattern will not automatically generate messages (used if the traffic pattern is to remain permanently inactive or if the message generation process is to be kept under exclusive user control)

`SPF_on`
: standard performance measures (section 14.2) will be calculated for the traffic pattern (this is the default); the user can still override or extend the standard methods that take care of this end

`SPF_off`
: no standard performance measures will be collected for this traffic pattern

It is illegal to specify more than one distribution type for a single distribution parameter, i.e., `MIT_exp+MIT_unf`. If no option is selected for a given distribution parameter, this parameter is not defined. For example, if none of `BIT_exp`, `BIT_unf`, or `BIT_fix` is specified, the traffic pattern is not bursty. It makes little sense to omit the specification of the message arrival time or message length distribution, unless `SCL_off` is selected. SIDE will complain about the incomplete definition of a traffic patterns to be automatically controlled by the simulator.

The number and interpretation of the remaining setup arguments depend on the contents of `flag`. All these arguments are expected to be `double` numbers. They describe the numerical parameters of the arrival process in the following way and order:

- If `MIT_exp` or `MIT_fix` is selected, the next (single) number specifies the (mean) message interarrival time in *ETU*s (section 3.2). If `MIT_unf` is specified, the next two numbers determine the minimum and the maximum message interarrival time (also in *ETU*s). Note that if these two numbers are equal, the message interarrival time is fixed.
- If `MLE_exp` or `MLE_fix` is chosen, the next number specifies the (mean) message length (in bits). If `MLE_unf` is selected, the next two numbers determine the minimum and the maximum message length (in bits). Note that if these two numbers are equal, the message length is fixed.
- If `BIT_exp` or `BIT_fix` is selected, the next number specifies the (mean) burst interarrival time in *ETU*s (section 3.2). If `BIT_unf` is specified, the next two numbers determine the minimum and the maximum burst interarrival time (also in *ETU*s).
- If `BSI_exp` or `BSI_fix` is chosen, the next number specifies the (mean) burst size as the number of messages constituting the burst. If `BSI_unf` is selected, the next two numbers determine the minimum and the maximum burst size.

Thus, the maximum number of the numerical distribution parameters is 8. It is important to remember that all of them must be `double`, even those whose values are apparently integral. Owing to the fact that the setup method of `Traffic` accepts an unknown number of arguments, those arguments cannot be type-checked during compilation and automatically converted to `double`.

A completely defined traffic pattern describes a message arrival process. For a non-bursty traffic, this process can be envisioned as the following cycle:

1. A time interval t_{mi} is generated at random, according to the message interarrival time distribution. The process sleeps for t_{mi} *ITU*s and then moves to step 2.

2. A message is generated. Its length is determined at random according to the message length distribution; its sender and receiver are selected using the list of communication groups associated with the traffic pattern (as explained below). The message is queued at the sender, and the generation process continues at step 1.

With a bursty traffic pattern, it is assumed that messages arrive in groups called bursts. Each burst carries a specific number of messages that are to be generated according to similar rules as for a regular, non-bursty traffic pattern. The process of burst arrival is carried out according to this scheme:

1. An integer counter denoted by *PMC* (for *pending message count*) is initialized to 0.

2. A time interval T_{bi} is generated at random according to the burst interarrival time distribution. The process sleeps for t_{bi} *ITU*s and then continues at step 3.

3. A random integer number s_b is generated according to the burst size distribution. This number determines the number of messages to be generated within the burst.

4. *PMC* is incremented by s_b. If the previous value of *PMC* was zero, a new process is started which looks similar to the message generation process for a non-bursty traffic pattern (see above) and runs **in parallel** with the burst arrival process. Each time a new message is generated, the message arrival process decrements *PMC* by one. When *PMC* goes down to zero, the process terminates. Having completed this step, **without waiting for the termination of the message arrival process**, the burst arrival process continues at step 2.

Typically, the message interarrival time within a burst is much shorter than the burst interarrival time. In particular, it can be 0 in which case all messages of the burst arrive at once—within the same *ITU*. Note that when two bursts overlap, i.e., a new burst arrives before the previous one is gone, the new burst adds to the previous burst.

The procedure of determining the sender and the receiver for a newly-arrived message, in the case when the traffic pattern is based on one communication group, was described

in section 10.5. Now we shall discuss the case of multiple communication groups. Again suppose that

$$S = \langle (s_1, w_1), \ldots, (s_k, w_k) \rangle \quad \text{and}$$

$$R = \langle (r_1, v_1), \ldots, (r_p, v_p) \rangle$$

are the sets of senders and receivers of a communication group *G = (S, R)*. If *G* is a broadcast group, we may assume that all $v_i, i = 1, \ldots, p$ are zeros (they are irrelevant then). The value of

$$W = \sum_{i=1}^{k} w_i$$

is called the sender weight of group *G*. Suppose that the definition of a traffic pattern *T* involves communication groups $G_1, \ldots, G_m$ with their sender weights $W_1, \ldots, W_m$, respectively. Assume that a message is generated according to *T*. The sender of this message is determined in such a way that the probability of a station

$$s_i^j : (s_i^j, w_i^j) \in S_j \quad \text{and} \quad S_j \in G_j$$

being selected is equal to

$$P_S(s_i^j) = \frac{w_i^j}{\sum_{l=1}^{m} W_l}.$$

Thus, the weight of a station in the senders set specifies its relative frequency of being chosen as an actual sender with respect to all potential senders specified in all communication groups belonging to *T*. Note that the expression for P_S is meaningless if the sum of all W_l's is zero. Thus, at least one sender weight in at least one selection group must be strictly positive.

Once the sender has been established, the receiver of the message is chosen from the receivers set *R* of the communication group that was used to determine the sender. If this communication group happens to be a broadcast group, the entire receivers group is used as the *receiver*. In consequence, the message will be addressed to all the stations listed in the receivers group, including the sender, if it occurs in *R*. For a non-broadcast message, the receiver is selected from *R*, as described in section 10.5.

The senders set of a traffic pattern to be handled by the standard `Client` must not be empty: the `Client` must always be able to queue a generated message at a specific sender station. However, the receivers set can be empty in which case the message will not be addressed explicitly to any specific station (its `Receiver` attribute will be `NONE`).

Example

Despite the apparent complexity, the definition of a traffic pattern is quite simple in most cases. For example, to describe a uniform traffic pattern (in which all stations participate

equally) with a *Poisson* message arrival process and fixed-length messages, the following sequence can be used:

```
traffic MyPattType (MyMsgType, MyPktType) {
};
MyPattType *tp;
SGroup *sg;
CGroup *cg;
...
sg = create SGroup;
cg = create CGroup (sg, sg);
tp = create MyPattType (sg, MIT_exp+MLE_exp, iart, lngth);
```

`MyMsgType` and `MyPktType` are assumed to have been defined earlier; `iart` and `lngth` are variables of type `double` specifying the mean message interarrival time and the mean message length, respectively.

To define a simple traffic pattern, as the one above, you do not even have to bother with communication groups. As a matter of fact, any traffic pattern that can be defined using explicit communication groups can be defined without them. Namely, by calling the following methods of `Traffic` you can describe the sets of senders and receivers directly, one-by-one, without putting them into groups first:

```
addSender (Station *s = ALL, double w = 1.0, int gr = 0);
addSender (Long sid, double w = 1.0, int gr = 0);
addReceiver (Station *s = ALL, double w = 1.0, int gr = 0);
addReceiver (Long sid, double w = 1.0, int gr = 0);
```

By calling `addSender` (or `addReceiver`) we add one sender (or receiver) to the traffic pattern, assign an optional weight to it, and (also optionally) assign it to a specific communication group. One exception is a call with the first argument equal `ALL` (an alias for `NULL`) or without any arguments. In such a case all stations are added to the group. The only legal value for `w` is then 1.0; the weights of all stations are the same and equal to 1/`NStations`.

It is possible to use the above methods to supplement the description of a traffic pattern that was created with explicit communication group(s). For the purpose of `addSender` and `addReceiever`, the explicit communication groups are numbered from 0, in the order of their occurrence on the list that was specified as the first setup argument for the traffic pattern.

For `addReceiver`, if `BROADCAST` (–1) is put in place of `w`, it means that the communication group is a broadcast one. This must be consistent with the previous and future additions of receivers to the group.

With addSender and addReceiver, the above example can be simplified into the following equivalent form:

```
traffic MyPattType (MyMsgType, MyPktType) {
};
MyPattType *tp;
...
tp = create MyPattType (MIT_exp+MLE_exp, iart, lngth);
tp->addSender (ALL);
tp->addReceiver (ALL);
```

Note that before addSender and addReceiever are called, the traffic pattern, although formally created, is incompletely specified. If it were left in such a state, SIDE would detect a problem upon initializing the message generation process for this pattern.

Traffic patterns are *Objects*. Id attributes of traffic patterns reflect their creation order. The first-created traffic pattern is assigned Id 0, the second 1, etc. The function

```
Traffic *idToTraffic (Long id);
```

converts the traffic Id to the object pointer. The global variable NTraffics of type int contains the number of traffic patterns defined so far.

10.6.3 Modifying the standard behavior of traffic patterns

A traffic pattern created with SCL_on (see section 10.6.2) is automatically activated and its behavior is driven by the standard traffic generation process of the Client described in the previous section. Traffic patterns created with SCL_off can be controlled by user-supplied processes.

A traffic pattern driven by the standard Client can be activated and deactivated dynamically. To deactivate an active traffic pattern, the following Traffic method can be used:

```
void suspend ();
```

Calling suspend has no effect on a traffic pattern already being inactive. An active traffic pattern becomes inactive, in the sense that its generation process is suspended. From now on, the Client will not generate any messages described by the deactivated traffic pattern.

The following Traffic method cancels the effect of suspend for the traffic pattern:

```
void resume ();
```

Similar to `suspend`, it has no effect on an active traffic pattern. Calling `resume` for an inactive traffic pattern has the effect of resuming the `Client` process associated with the traffic pattern.

The above operations also exist as `Client` methods (with the same names). Calling the corresponding method of the `Client` corresponds to calling this method for all traffic patterns. Thus

```
Client->suspend ();
```

deactivates all (active) traffic patterns and

```
Client->resume ();
```

activates all traffic patterns being inactive.

The most natural application of the last two methods is in situations when the modeled system should undergo a non-trivial startup phase during which no traffic should be offered to the network. In such a case, all traffic patterns can be suspended before the protocol processes are started and then resumed when the startup phase is over.

A traffic pattern driven by a user-supplied process (or a collection of user-supplied processes) is not automatically suspended by `suspend` and resumed by `resume`, but the user process can learn about the status changes of the traffic pattern and respond consistently to these changes. The following two methods of type `Traffic`:

```
int isSuspended ();
int isResumed ();
```

can be used to poll a traffic pattern about its suspended/resumed status. These methods also exist as `Client` attributes and, in their `Client` versions, return the combined status of all the traffic patterns. The `Client` variant of `isSuspended` returns `YES` if all traffic patterns have been suspended and `NO` otherwise. The other method is a straightforward negation of `isSuspended`.

The following two events are triggered on a traffic pattern whenever its suspended/resumed status is changed:

`SUSPEND`
: The event occurs whenever the traffic pattern changes its status from "resumed" to "suspended." If a wait request for `SUSPEND` is issued to a traffic pattern that is already suspended, the event occurs immediately, i.e., within the current *ITU*.

RESUME
: The event is generated whenever the traffic pattern changes its status from "suspended" to "resumed." If a wait request for RESUME is issued to a traffic pattern that is already resumed, the event occurs immediately.

A similar pair of events exists for the Client. The Client versions of SUSPEND and RESUME are triggered by the Client versions of suspend and resume operations. Such an operation must be *effective* to generate the corresponding event, i.e., the status of at least one traffic pattern must be actually changed by the operation. Note that by calling the Client's suspend/resume, we also trigger the SUSPEND/RESUME events for the traffic patterns that are affected by the operation.

Note: Normally, while a traffic pattern is suspended, the modeled time that passes during that period counts, e.g., in calculating the effective throughput (sections 17.5.5, 17.5.12). Generally, it is recommended to (re)initialize the standard performance measures for the traffic pattern (section 14.2.4) whenever its status changes from inactive to active. This suggestion also applies to the entire Client. (In most cases, all traffic patterns are deactivated and activated simultaneously.)

In the course of their action, the Client processes governing the behavior of a traffic pattern call a number of methods declared within type Traffic. Most of these methods are virtual and user-accessible; thus, they can be redefined in a user-declared subtype of Traffic. Some variables defined in Traffic are also made public—to facilitate programming non-standard extensions of the Client. The following variables store the options specified in setup arguments when the traffic pattern was created:

```
char DstMIT, DstMLE, DstBIT, DstBSI, FlgSCL, FlgSUS, FlgSPF;
```

Each of the first four variables can have one of the following values:

UNDEFINED
: if the distribution of the corresponding parameter (MIT—*message interarrival time*, MLE—*message length*, BIT—*burst interarrival time*, BSI—*burst size*) was not defined at the moment when the traffic pattern was created (see section 10.6.2)

EXPONENTIAL
: if the corresponding parameter is exponentially distributed

UNIFORM
: if the corresponding parameter has uniform distribution

`FIXED`
: if the corresponding parameter has a fixed value

The remaining two variables can take values `ON` or `OFF` (1 and 0, respectively). If `FlgSCL` is `OFF`, it means that the standard processing for the traffic pattern has been switched off (`SCL_off` was specified at the creation of the traffic pattern); otherwise, the standard processing is in effect. The value of `FlgSUS` tells whether the traffic pattern has been suspended (`ON`) or is active (`OFF`).

If `FlgSPF` is `OFF` (`SPF_off` was selected when the traffic pattern was created), no standard performance measures are collected for this traffic pattern. The issues related to gathering performance statistics for traffic patterns are discussed in section 14.2.

The following four pairs of attributes contain the numerical values of the distribution parameters describing the arrival process:

```
double ParMnMIT, ParMxMIT,
       ParMnMLE, ParMxMLE,
       ParMnBIT, ParMxBIT,
       ParMnBSI, ParMxBSI;
```

If the corresponding parameter (`MIT`—*message interarrival time*, `MLE`—*message length*, `BIT`—*burst interarrival time*, `BSI`—*burst size*) is uniformly distributed, the two attributes contain the minimum and the maximum values. Otherwise (fixed or exponential distribution), the first attribute of the pair contains the fixed or mean value, and the the contents of the second attribute are irrelevant.

Note: The `TIME` values, i.e., `ParMnMIT`, `ParMxMIT`, `ParMnBIT`, `ParMxBIT` are kept in ITUs. They are converted to ITUs from the user setup specification (which is in ETUs).

The following virtual methods of `Traffic` take part in the message generation process:

```
virtual TIME genMIT ();
```

This method is called to generate the time interval until the next message arrival. The standard version generates a random number according to the message interarrival time distribution.

```
virtual Long genMLE ();
```

This method is called to determine the length of a message to be generated. The standard version generates a random number according to the message length distribution.

```
virtual TIME genBIT ();
```

The method is called to determine the time interval until the next burst arrival. The standard version generates a random number according to the burst interarrival time distribution.

```
virtual Long genBSI ();
```

This method is called to generate the size of a burst (the number of messages within a burst). The standard version generates a random number according to the burst size distribution.

```
virtual int genSND ();
```

This method is called to determine the sender of a message to be generated. It returns the station `Id` of the sender or `NONE`, if no sender can be generated. The standard version determines the sender according to the configuration of communication groups associated with the traffic pattern—in the way discussed at the end of section 10.6.2.

```
virtual IPointer genRCV ();
```

The method is called to find the receiver for a message to be generated. It returns the station `Id` of the receiver or a pointer to the station group (for a broadcast receiver), or `NONE` if no receiver can be generated. The standard version generates the receiver from the communication group used to generate the last sender (section 10.6.2). Thus, it should be called after `genSND`.

```
virtual CGroup *genCGR (Long sid);
CGroup *genCGR (Station *s) {
    return (genCGR (s->Id));
};
```

The standard version of this method returns a pointer to the communication group containing the indicated station in the senders set. The station can be specified either via its `Id` or as a pointer to the station object. It may happen that the indicated station (call it s) occurs in the senders sets of two or more communication groups. In such a case, one of these groups is chosen at random with the probability proportional to the sender weight of s in that group versus its weights in the other groups. Specifically, if $G_1, \ldots, G_k$ are the multiple communication groups that include s as a sender, and $w_s^1, \ldots, w_s^k$ are the respective sender weights of s in them, then G_i is selected with the probability

$$P^i = \frac{w_s^i}{\sum_{j=1}^k w_s^j} \ .$$

The chosen communication group becomes the "current group" and the next call to `genRCV` will select the receiver from that group.

One application of `genCGR` is when the sender is already known (e.g., it has been selected in a non-standard way), and a matching receiver must be generated. One may want to generate this receiver in such a way that it is chosen from a certain communication group containing the sender station, in accordance with the distribution of weights.

If the specified station does not occur as a sender in any communication group associated with the traffic pattern (i.e., it cannot be a sender for this traffic pattern) `genCGR` returns `NULL`. This way the method can be used to determine whether a station is a legitimate sender for a given traffic pattern.

```
virtual Message *genMSG (Long snd, IPointer rcv, Long lgth);
Message *genMSG (Long snd, SGroup *rcv, Long lgth);
Message *genMSG (Station *snd, Station *rcv, Long lgth);
Message *genMSG (Station *snd, SGroup *rcv, Long lgth);
```

The method is called to generate a new message and queue it at the sender. The sender station is indicated by the first argument, which can be either a station `Id` or a station object pointer. The second argument can identify a station (in the same way as the first argument) or a station group for a broadcast message. The last argument is the message length in bits.

The new message is queued at the sender at the end of the queue corresponding to the given traffic pattern (see section 10.7). Its pointer is also returned as the function value.

10.6.4 Intercepting packet and message deallocation

Sometimes packets and/or messages may contain pointers to dynamically allocated structures that have to be freed when the packet disappears. Unfortunately, using the standard destruction mechanism for this kind of clean-up is not possible in SIDE, because, for the sake of memory efficiency, packets are represented within link activities as variable-length chunks (allocated at the end of the activity object) that formally belong to the standard type `Packet`. Virtual destructors do not help because the function that internally clones a transmitted packet to become part of the corresponding link activity cannot know the packet's actual type: it merely knows the length of the packet's object.

It is possible, however, to intercept packet and message deallocation events, including the standard types, i.e., `Packet` and `Message`, in a way that exploits the connection between those objects and the client. To this end, the following two virtual methods can be declared within a traffic type:

```
void pfmPDE (ptype *p);
void pfmMDE (mtype *m);
```

where ptype and mtype are the types of packets and messages associated with the traffic pattern (section 10.6.1). If declared, pfmPDE will be called whenever a packet structure is about to be deallocated with the argument pointing to the packet. This covers the cases when a packet is explicitly deleted by the protocol program, as well the internal deallocations of Link or RFChannel activities (sections 11.1.1 and 12.2.1). Note that the deallocation of a non-standard packet (i.e., one that has no associated traffic object—section 10.2) cannot be intercepted this way. Link-level cleanup (section 11.5) is recommended for such packets, if needed.

Similarly, pfmMDE will be called for each message deallocation, which in most cases happens internally—when a queued message has been entirely converted into packets (section 10.8.1).

The names of the two methods start with pfm because they formally belong to the same class as the performance collecting methods (discussed in section 14.2.3), which are called to mark certain relevant stages in packet processing. In this context, the deallocation of a packet/message is clearly a relevant processing stage, although its perception is rather useless from the viewpoint of calculating performance measures.

10.7 Message queues

Each station has two user-visible pointers that describe queues of messages awaiting transmission at the station. These pointers are declared within type Station in the following way:

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

MQHead points to an array of pointers to messages, each pointer representing the head of the message queue associated with one traffic pattern. The Id attributes of traffic patterns (reflecting their creation order—see section 10.6.2) index the message queues in the array pointed to by MQHead. Thus,

```
S->MQHead [i]
```

references the head of the message queue (owned by station S) corresponding to the traffic pattern number i. Note that S is a standard process attribute identifying the station that owns the process (section 7.2).

Similarly, MQTail identifies an array of message pointers. Each entry in this array points to the last message in the corresponding message queue and can be used to append a message at the end of the queue in a fast way.

MQHead and MQTail are initialized to NULL when the station is created. The two pointer arrays are generated and assigned to MQHead and MQTail when the first message is queued

at the station by `genMSG` (section 10.6.3). Thus, a station that never gets any message to transmit will never have a message queue, even an empty one.

An empty message queue is characterized by both its pointers (i.e., `MQHead [i]` and `MQTail [i]`) being `NULL`. The way messages are kept in queues is described in section 10.2. The `prev` pointer of the first message of an nonempty queue number `i` (associated with the `i`th traffic pattern) contains `&MQHead[i]`, i.e., it points to a fictitious message containing `MQHead[i]` as its `next` attribute.

It is possible to impose a limit on the length of message queues—individually at specific stations or globally for the entire network. This possibility may be useful for investigating the network behavior under extreme traffic conditions. By setting a limit on the number of queued messages, you will be able to avoid overflowing the memory of your protocol program. A call to the following global function:

```
void setQSLimit (Long lgth);
```

declares that the total number of messages queued at all stations together cannot exceed `lgth`. When `setQSLimit` is called as a `Traffic` method, it restricts the number of queued messages belonging to the given traffic pattern. It is also possible to call the function as a `Station` method in which case it limits the combined length of message queues at the given station. The parameter can be skipped; it is then assumed to be `MAX_long` and stands for "no limit" (which is the default).

Whenever a message is to be generated and queued at a station by `genMSG` (section 10.6.3), SIDE checks whether queuing it would not exceed one of the three possible limits: the global limit on the total number of messages awaiting transmission, the limit associated with the traffic pattern to which the message belongs, or the limit associated with the station at which the message is to be queued. If any of the three limits is already reached, no message is generated and the action of `genMSG` is void. `TheMessage` returns `NULL` in such a case.

Messages belonging to a traffic pattern that was created with the `SPF_off` indicator (section 10.6.2) do not count to any of the three possible limits. For such a traffic pattern, `genMSG` always produces a message and queues it at the corresponding queue of the selected sender.

Note: For efficiency reasons, the queue size limit checking is disabled by default. The program must be created with the `-q` option of `mks` (section 19.2) to enable this feature.

The following two `Station` methods tell the number of messages and bits queued at the station:

```
Long getMQSize (int tp = NONE);
BITCOUNT getMQBits (int tp = NONE);
```

When the argument is NONE (or absent), the methods return the total number of messages/bits accumulated over all traffic patterns (queues) at the station. Otherwise, the argument must be a legal traffic pattern Id and the returned values pertain to the single indicated traffic pattern.

10.8 Inquiries

A process interested in acquiring a packet for transmission can *inquire* for it. Such an inquiry can be addressed to

- a specific traffic pattern—if the process is explicitly interested in packets generated according to this pattern
- the Client—if the process does not care about the traffic pattern
- a message—if the process wants to explicitly indicate the message from which the packet is to be created

We will refer to all these types of inquiries as to *client inquiries*, although only a certain subset of them is implemented as a collection of methods in the Client.

10.8.1 Acquiring packets for transmission

The following method of the Client can be used to check whether a message awaiting transmission is queued at the station and, if so, to create a packet out of this message:

```
int getPacket (Packet *p, Long min, Long max, Long frm);
```

The first argument is a pointer to the packet buffer where the acquired packet is to be stored. The remaining three arguments indicate the minimum packet length, the maximum packet length, and the length of the packet frame information (header and trailer), respectively. All these lengths are in bits.

The function examines all message queues at the current station. If the value returned by the function is NO, it means that all the queues are empty and no packet can be acquired. Otherwise, the earliest-arrived message is chosen, and a packet is created out of this message. If two or more messages with the same earliest arrival time are queued at the station (possibly in different queues), one of them is chosen at random.[27]

If max is greater than the message length, or equal 0, which stands for "no limit," the entire message is turned into the packet. Otherwise, only max bits are extracted from the

[27] If the program has been created with the -D option of mks (sections 7.4 and 19.2), the traffic pattern with the lowest Id wins in such a case.

message and its remaining portion is left in the queue for further use. If `min` is greater than the packet length obtained so far, the packet is artificially expanded (*inflated*) to the size of `min` bits. The added bits count to the total length of the packet (e.g., they influence the packet's transmission time), but they are not considered to be information bits, e.g., they are ignored in calculating the *effective throughput* (sections 17.5.5, 17.5.12). Note that `min` can be greater than `max` in which case the packet is always inflated.

Finally, `frm` bits are added to the total length of the packet. This part is used to represent the length of various headers and trailers required by the protocol, but not useful from the viewpoint of the packet information content.

Let `m` denote the message length in bits. The values of the two length attributes `TLength` and `ILength` (section 10.2) of a packet acquired by `getPacket` are determined according to the following algorithm:

```
ILength = (m->Length > max) ? max : m->Length;
TLength = ILength + frm;
if (ILength < min) TLength += min - ILength;
m->Length -= ILength;
```

Before the message length is decremented, the packet's setup method is called with the message pointer passed as the argument (see section 10.2). The standard version of this method is empty. The user may (and should) define such a setup method in proper subtypes of `Packet`—to set the non-standard attributes. Note that generally, the setting of the non-standard attributes of the packet may depend on the contents of the message from which the packet has been acquired. In particular, this message may belong to a proper subtype of `Message` and may have non-standard attributes of its own.

If after subtracting the packet length, `m->Length` ends up equal 0, the message is discarded from the queue and deallocated as an object. Note that it is legal for the original message length to be zero. If this is the case, the information length of the acquired packet will be zero and, following the acquisition, the message will be removed from the queue. The standard traffic generator of the `Client` never generates zero-length messages; however, they can be created by custom traffic generators programmed by the user (section 10.6.3).

It is illegal to try to acquire a packet into a buffer that is not *empty* (section 10.3). After `getPacket` puts a packet into the buffer, the buffer status is changed to *full*.

Two environment variables (section 7.6) are set after a successful return (with value `YES`) of `getPacket`. If the message from which the packet has been acquired remains in the message queue (i.e., the effective packet length is smaller than the original message length), the environment variable `TheMessage` (`Info01`) of type `Message*` points to that message. Otherwise, `TheMessage` contains NULL. In either case the environment variable `TheTraffic` (`Info02`) of type `int` returns the `Id` of the traffic pattern to which the acquired packet belongs.

Below we list other variants of `getPacket`. All the rules described above, unless we explicitly say otherwise, apply to the other variants as well.

The following `Client` method works identically as the one discussed above:

```
int getPacket (Packet &p, Long min, Long max, Long frm);
```

the only difference being that the packet buffer is passed by reference rather than a pointer.

Sometimes, the simple selection of the earliest message awaiting transmission is not what the protocol program wants. In general, the predicate describing which message should be used to create a packet may be described by a compound and dynamic condition. The following method of the `Client` takes care of this end:

```
int getPacket (Packet *p, MTTYPE f, Long min, Long max, Long frm);
```

where `f` is a pointer to a function which should be declared as:

```
int f (Message *m);
```

This function should return nonzero if the message pointed to by `m` satisfies the selection criteria, and zero otherwise.

The message from which the packet is to be extracted is determined in a manner similar to the previous version of `getPacket`, with the exception that instead of "the earliest-arrived message" the qualified version looks for "the earliest-arrived message that satisfies `f`." Again, if there are two or more earliest-arrived messages satisfying `f` (i.e., all these messages have arrived at the same time) one of them is chosen at random.[28]

There is a variant of qualified `getPacket` in which the buffer is passed by reference rather than a pointer.

The methods described above are defined within the `Client` and they treat all traffic patterns globally. One possible way of restricting the range of the `Client`'s `getPacket` to a single traffic pattern is to use one of the two qualified variants and to pass the following function as `f`:

```
int f (Message *m) { return (m->TP == MyTPId); };
```

where `MyTPId` is the `Id` of the traffic pattern in question. There is, however, a better (and more efficient) way to achieve the same result. Namely, all the `Client` versions of `getPacket` have their counterparts as methods of `Traffic`. Their configurations of arguments and behavior are identical to those of the corresponding `Client` methods with the exception that the search for a message is restricted to the single traffic pattern.

[28] If the program has been created with the `-D` option of `mks` (sections 7.4 and 19.2), the traffic pattern with the lowest `Id` wins in such a case.

The two unqualified versions of `getPacket` also occur as methods of `Message` (with the same configurations of arguments). They are useful when the message from which a packet is to be extracted is already known. With the following call:

```
Msg->getPacket (buf, min, max, frm);
```

where `Msg` points to a message (i.e., an object belonging to a subtype of `Message`), a packet is extracted from `Msg` (according to the standard rules) and put into `buf`. If after this operation the message pointed to by `Msg` turns out to be empty, its `prev` attribute is examined to determine whether the message belongs to a queue. If `prev` is `NULL`, it is assumed that the message does not belong to a queue and no further processing is taken. Otherwise, the empty message is dequeued and deallocated as an object. The user should make sure that the `prev` attribute of a message created in a non-standard way, which shouldn't be automatically dequeued by the `Message` version of `getPacket`, contains `NULL`.

There is no way for a `Message` `getPacket` to fail (returning `NO`). For compatibility with the other versions, `YES` is always returned as the method's value.

10.8.2 Testing for packet availability

Using one of the `getPacket` variants discussed in the previous section, it is possible to acquire a packet for transmission or learn that no packet (of the required sort) is available. Sometimes one would like to learn whether a packet can be acquired without actually acquiring it. The following two `Client` methods can be used for this purpose:

```
int isPacket ();
int isPacket (MTTYPE f);
```

The first version of `isPacket` returns `YES` if there is a message (of any traffic pattern) queued at the current station. Otherwise, the method returns `NO`. The second version returns `YES` if a message satisfying the predicate `f` (see section 10.8.1) is queued at the station, and `NO` otherwise.

Similar two versions of `isPacket` are defined within `Traffic` and they behave exactly as the `Client` method, except that only the indicated traffic pattern is examined.

If `isPacket` returns `YES`, two environment variables (section 7.6) are set. `TheMessage` (`Info01`) of type `Message*` points to the message located by the method,[29] and `TheTraffic` (`Info02`) of type `int` contains the `Id` of the traffic pattern to which the message belongs.

No `isPacket` method is defined for `Message`.

[29] Note that this is not necessarily the same message that will be used by a subsequent call to `getPacket`, unless the program has been generated with the `-D` option of `mks` (section 19.2.

10.9 Wait requests

When an attempt to acquire a packet for transmission fails (e.g., `getPacket` returns `NO`—section 10.8.1), the process issuing the inquiry may choose to suspend itself until a message gets queued at the station. By issuing the following wait request:

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

the process declares that it wants to be awakened when a message of any traffic pattern is queued at the station. Thus, the part of the process that takes care of acquiring packets for transmission may look as follows:

```
...
state TryNewPacket:
    if (!Client->getPacket (buf, minl, maxl, frml)) {
        Client->wait (ARRIVAL, TryNewPacket);
        sleep;
    }
...
```

Of course, the `order` parameter can be specified, as usual, to assign a priority to the `Client` wait request.

There are two ways of indicating that we are interested in messages that belong to a specific traffic pattern. One way is to put the `Id` of the traffic pattern in place of `ARRIVAL` in the above call to the `Client`'s `wait`. The other, more recommended solution is to use the `Traffic`'s `wait` method instead, i.e., to call

```
TPat->wait (ARRIVAL, TryAgain);
```

where `TPat` is the pointer to the traffic pattern in question. Such a call indicates that the process wants to be restarted when a message belonging to the indicated traffic pattern is queued at the station.

The keyword `ARRIVAL` can be replaced in the above examples with `INTERCEPT`. The semantics of the two events is identical, except that `INTERCEPT` gets a higher priority (a lower order) than `ARRIVAL`, irrespective of the actual values of the order parameters specified with the requests. Imagine that a process wants to intercept all messages arriving at the station, e.g., to preprocess them by setting some of their non-standard attributes. At first sight it would seem natural to program this process in the following way:

```
...
state WaitForMessage:
```

```
        Client->wait (ARRIVAL, NextMessage);
    state NextMessage:
        ...
        preprocess the message
    ...
```

However, this solution has one serious drawback. Assume that a message arrives at a station within the current *ITU* and two processes owned by the station are waiting for `ARRIVAL`. The waking events for both processes are scheduled to occur within the same current *ITU* and there is no way to tell which process will *actually* run first (see section 7.4). One way to make sure that the message preprocessor is run before any process that may use the message is to assign a low order to the wait request issued by the preprocessor and make sure that the other processes specify higher values. Another possibility is to replace `ARRIVAL` in the wait request issued by the preprocessor with `INTERCEPT`. `INTERCEPT` is an event which, similar to `ARRIVAL`, is triggered when a message arrives at the station, but the process awaiting `INTERCEPT` will be given control before any other process waiting for `ARRIVAL` is awakened. Only one process at a station may be waiting for `INTERCEPT` at a time, unless each of the multiple `INTERCEPT` requests is addressed to a different traffic pattern and none of them is addressed to the `Client`.

The implementation of `INTERCEPT` is similar to the implementation of priority signals (section 7.8.2) and the *priority put* operation for mailboxes (section 13.2.4).

Whenever a process is awakened by a message arrival event (`ARRIVAL` or `INTERCEPT`), the environment variables `TheMessage` (of type `Message*`) and `TheTraffic` (of type `int`) (sections 7.6, 10.8.2) return the pointer to the new message and the `Id` of the message traffic pattern, respectively.

11 Links and the `Port` *AI*

A process may wish to reference a port for one of the following three reasons:

- to insert into the port (and thus into the underlying link) an *activity*, e.g., a packet
- to *inquire* the port about its current or past status, e.g., to check whether an activity is heard on the port
- to issue a *wait request* to the port, e.g., to await an activity to arrive at the port in the future

11.1 Activities

Two types of activities can be inserted into ports: *packets* and *jamming signals*. Packets (section 10.2) represent some portions of information exchanged between stations. The

operation of inserting a packet into a port is called a *packet transmission*. Jamming signals (or jams for short) are special activities that are clearly distinguishable from packets.

11.1.1 Processing of activities in links

Activities must be explicitly started and explicitly terminated (section 11.1.2). When an activity is started, an internal object representing the activity is built and added to the link. One attribute of this object (*starting time*) tells the time (in *ITU*s) when the activity was started. Another attribute (*finished time*) indicates the time when the activity was terminated. The difference between *finished time* and *starting time* is called the *duration* of the activity.

The *finished time* attribute of an activity that has been started but not terminated yet is undefined, and so is the activity's duration. These elements become defined when the activity is terminated.

An activity inserted into one port of a link will arrive at another port connected to the link according to the rules described in section 6.2.1. In particular, assume that two ports p_1 and p_2 are connected to the same link l and the propagation distance from p_1 to p_2 is d *ITU*s.[30] If an activity is started on p_1 at time t_s, the beginning of this activity will arrive at p_2 at $t_s + d$. Similarly, if the activity is terminated at t_f, its end will be perceived at p_2 at $t_f + d$.

Two sets of activities are associated with each link. One set contains the *alive* activities, i.e., activities that at the given moment are "physically" present in the link. The other set is the link *archive* and contains activities that have disappeared from the link. If an activity that was started on port p is terminated at time t, it will be removed from the alive set at time t_{max}, where t_{max} is the maximum distance from p to another port connected to the same link. Then the activity will be put into the link archive where it will be kept for a certain amount of time called the *archival period*, which is determined upon link creation (see section 6.2.2). If the archival period for a link is zero, no link archive is maintained: all activities that formally leave the link are immediately discarded and forgotten. This is how things appear in real life.

11.1.2 Starting and terminating activities

The following port method is used to start a packet transmission:

```
void startTransfer (Packet *buf);
```

where `buf` points to the buffer containing the packet to be transmitted (section 10.3). A copy of this buffer is made and inserted into a data structure representing a new link ac-

[30] Note that the distances are specified by the user in *DU*s and internally converted to *ITU*s (section 6.3.3).

tivity. The pointer to this packet copy is returned via the environment variable `ThePacket` (`Info01`) of type `Packet*`. Another environment variable, `TheTraffic` (`Info02`) of type `int`, returns the `Id` attribute of the traffic pattern that was used to generate the packet.

There are two ways to terminate a packet transmission. The following port method:

```
int stop ();
```

terminates the transmission of a packet inserted by a previous call to `startTransfer`. With `stop`, the terminated packet is *complete*, i.e., formally equipped with a pertinent trailer that allows the receiving party to recognize the complete packet. Instead of `stop` you can use this port method:

```
int abort ();
```

which *aborts* the transmission. This means that the packet has been interrupted in the middle, and it will not be perceived as a complete (receivable) packet.

Each of the two methods returns an integer value that tells the type of the terminated activity. This value can be either `TRANSFER` or `JAM`. It is illegal to attempt to interrupt a non-existent activity, i.e., to execute `stop` or `abort` on an idle port.

If the terminated activity was a packet transmission, two environment variables `ThePacket` and `TheTraffic` (see above) are set. The first one points to the packet in question, the second tells the `Id` of the traffic pattern to which the packet belongs.

Formally, the amount of time needed to transmit a packet `buf` on a port `prt` is equal to `buf->TLength * prt->getTRate()`, i.e., to the product of the total length of the packet (section 10.2) and the port's transmission rate (section 6.3.1). Thus, a sample process code for transmitting a packet might look as follows:

```
...
state Transmit:
    MyPort->startTransfer (buffer);
    Timer->wait (buffer->TLength*MyPort->getTRate (), Done);
state Done:
    MyPort->stop ();
    ...
```

Note that generally the time spent on transmitting a packet need not have much to do with the contents of the packet's `TLength` attribute. This time is determined solely by the difference between the moments when the transmission of the packet was stopped and when it was started. In particular, if a process starts transmitting a packet and then forgets to terminate it (by `stop` or `abort`), the packet will be transmitted forever.

In most cases, the user would like to transmit the packet for the amount of time determined by the product of the packet length and the port transmission rate, possibly aborting the transmission earlier if a special condition occurs. The port method

```
void transmit (Packet *buf, int done);
```

starts transmitting the packet contained in `buf` and automatically sets up the `Timer` to wake up the process in state `done` when the transmission is complete. The transmission still has to be terminated explicitly. Thus, the code fragment listed above is equivalent to

```
...
state Transmit:
    MyPort->transmit (buffer, Done);
state Done:
    MyPort->stop ();
    ...
```

The following port method is recommended to convert a number of bits to the number of *ITU*s required to transmit these bits on a given port:

```
TIME bitsToTime (int n = 1);
```

Calling

```
t = p->bitsToTime (n);
```

has the same effect as executing:

```
t = p->getTRate () * n;
```

The transmission rate of a port can be changed at any time by calling `setTRate` (see section 6.3.1). If `setTRate` is invoked after `startTransfer` or `transmit` but before `stop`, it has no impact on the packet being transmitted. It is also possible to change the transmission rates of all ports connected to a given link by executing the `setTRate` method associated with type `Link`. The `Link`'s method accepts the same argument as the `Port`'s method, but it returns no value (its type is `void`).

The following port method:

```
void startJam ();
```

is used to start emitting a jamming signal on the port. Similar to packet transmissions, jams must be explicitly terminated by `stop` or `abort`; in this case, both methods have identical semantics.

It is possible to start emitting a jamming signal and, at the same time, set the `Timer` for a specific amount of time. A call to this port method:

```
void sendJam (TIME d, int done);
```

is equivalent to the sequence:

```
MyPort->startJam ();
Timer->wait (d, done);
```

Note: It is illegal to insert more than one activity into a single port at a time. However, it is perfectly legitimate to have two ports connected to the same link and separated by the distance of 0 *ITU*s. Thus, two or more activities can be inserted simultaneously into the same place of a link (but from different ports).

11.2 Wait requests

The first argument of a wait request addressed to a port is a symbolic constant that identifies the event category. The event categories for the `Port` *AI* are listed below.

`SILENCE`
: The event occurs at the beginning of the nearest silence period perceived on the port. If no activity is heard on the port when the request is issued, the event occurs within the current *ITU*.

`ACTIVITY`
: The event occurs at the beginning of the nearest activity (a packet or a jamming signal) heard on the port. The activity must be preceded by a silence period to trigger the event, i.e., two or more activities that overlap (from the viewpoint of the sensing port) are treated as a single continuous activity. If an activity is heard on the port when the request is issued, the event occurs within the current *ITU*.

`BOT`
: (*Beginning of Transmission*). The event occurs as soon as the port perceives the nearest beginning of a packet transmission. The event is not affected by interference or collisions, i.e., the possible presence of other activities arriving at the port at the same time. It is up to the protocol program to detect such conditions and interpret them.

EOT

(*End of Transmission*). The event occurs when the nearest end of a packet terminated by `stop` (section 11.1.2) is perceived by the port. Aborted transfer attempts, i.e., packets terminated by `abort` do not trigger this event. Similar to `BOT`, the event is not affected by the possible presence of other activities arriving at the port at the same time.

BMP

(*Beginning of My Packet*). The event occurs when the port perceives the beginning of a packet for which the current station (the one whose process issues the wait request) is the receiver (or one of the receivers, for a broadcast packet). Similar to `BOT` and `EOT`, the event is not affected by the possible presence of other activities arriving at the port at the same time.

EMP

(*End of My Packet*). The event occurs when the port perceives the end of a packet for which the current station is the receiver (or one of the receivers). The event is only triggered if the packet was terminated by `stop`, but (similar to `BOT`, `EOT` and `BMP`) it is not affected by the possible presence of other activities arriving at the port at the same time.

BOJ

(*Beginning of a Jamming signal*). The event occurs when the port hears the nearest beginning of a jamming signal preceded by anything not being a jamming signal. If a number of jams overlap (according to the port's perception), only the first of those jams triggers the `BOJ` event, i.e., overlapping jams are heard as one continuous jamming signal. If a jam is being heard on the port when the request is issued, the event occurs within the current *ITU*.

EOJ

(*End of a Jamming signal*). The event occurs when the port perceives the nearest end of a jamming signal (followed by anything not being a jamming signal). Two (or more) overlapping jams are heard as one, so only the end of the last of those signal will trigger the event. If no jam is heard when the request is issued, the event is **not** triggered immediately.

ANYEVENT

This event occurs whenever the port begins to sense a new activity or stops to sense some activity. Overlapping activities are separated, e.g., two overlapping activities generate four separate events, unless some of those events occur within the same *ITU* (see below).

`COLLISION`
: The event occurs when the earliest collision is sensed on the port (see section 11.2.1). If a collision is present on the port when the request is issued, the event occurs immediately.

When a process is awakened by one of the above-listed port events, with exception of `ANYEVENT` (see below), the environment variable `ThePort` (`Info02`) of type `Port*` contains the pointer to the port on which the waking event has occurred. If the waking event is related to a packet being heard at the port (i.e., `BOT`, `EOT`, `BMP`, `EMP`, and also `ANYEVENT`—see below), the environment variable `ThePacket` (`Info01`) of type `Packet*` returns the pointer to the object representing the packet. This object is a copy of the packet buffer given to `startTransfer` or `transmit` (section 11.1.2) when the transmission of the packet was started.

Note: The object pointed to by `ThePacket` is deallocated when the activity carrying the packet is removed from the link archive. If the archival period is zero (section 11.1.1), the activity (and the packet) is deallocated as soon as the last bit of the packet disappears from the link. To be exact, "as soon" includes a grace period of 1 *ITU*. This way, regardless of the priorities of the requisite events, the most remote recipient can always properly recognize and safely receive all perceived packets.

With `ANYEVENT`, if two (or more) events occur at the same time (within the same *ITU*), only one of them (chosen at random) will be presented. The restarted process is able to learn about all events that occur at the current moment by calling `events` (see section 11.3.1).

When a process awaiting `ANYEVENT` is awakened, the environment variable `TheEvent` (`Info02`) of type `int` contains a value identifying the event type. The possible values returned in `TheEvent` are represented by the following symbolic constants:

`BOT`	beginning of a packet
`EOT`	end of a complete packet terminated by `stop`
`ABTTRANSFER`	end of an aborted packet, i.e., terminated by `abort`
`BOJ`	beginning of a jamming signal
`EOJ`	end of a jamming signal

If the event type is `BOT`, `EOT` or `ABTTRANSFER`, `ThePacket` points to the corresponding packet object.

The purpose of `ANYEVENT` is to cover all circumstances when anything (anything at all) changes in the port's perception of the link. Thus, the event provides a hook whereby the user can implement custom functionality that is not available through a combination of

other (more specific events). One potential problem with ANYEVENT is that the persistent nature of port events may make its processing tricky. Consider the following generic code fragment:

```
...
state MonitorEvents:
    MyPort->wait (ANYEVENT, CheckItOut);
state CheckItOut:
    determine and handle the case
    proceed MonitorEvents;
...
```

This isn't going to work because whatever condition has triggered the event in state MonitorEvents, it still remains pending when, having completed handling the event in state CheckItOut, the process proceeds back to the initial state. The problem is that port events require a time advance to disappear. One possible way out is to replace proceed with skipto (section 8.1), but then it becomes difficult to intercept absolutely all conditions, like multiple events falling within the same *ITU*.

It is possible to declare a special semantics of ANYEVENT, whereby the event never remains pending and requires an actual condition change that *follows* the wait request to be triggered. This means that a once perceived condition will never trigger another wakeup, or, put differently, that the triggers are condition changes rather than the conditions themselves. Here is the port method to do the trick:

```
Boolean setAevMode (Boolean mode);
```

where the argument can be YES (equivalent to AEV_MODE_TRANSITION), or NO (AEV_MODE_PERSISTENT). In the first case, the non-persistent mode of triggering ANYEVENT is assumed. The second value declares the "standard" (and default) handling of ANYEVENT, whereby events persist until time is advanced. The methods returns the option's previous setting.

The recommended way of receiving ANYEVENT in the non-persistent mode is as follows:

```
...
state MonitorEvents:
    check for events pending and handle them
    MyPort->wait (ANYEVENT, MonitorEvents);
    sleep;
...
```

To avoid missing any events, before going to sleep, the process should account for all the events that are available (present) at the moment. This can be accomplished through port inquiries (section 10.8). Note that, with multiple events falling within the same *ITU*, the process still has to avoid interpreting the same event more than once. However, it does not have to worry that the same event will wake it up over and over again—in an infinite loop. In many practical cases, the interpretation of events is *idempotent*, i.e., processing them multiple times is not an issue. Otherwise, the process can mark them as processed, e.g., using packet flags (section 10.2).

It is possible to set the processing mode for `ANYEVENT` globally for all ports connected to the same link by executing the `Link` variant of `setAevMode` (that variant returns no value).

11.2.1 Collisions

By a collision, we mean a situation when two or more transfer attempts in the same link overlap, and the information carried by each of these transfers is (at least partially) destroyed. We say that a port perceives a collision if at least one of the following two statements is true:

- a jamming signal is heard on the port
- a packet being heard on the port is garbled by some other activity

The purpose of jamming signals is to represent special link activities that are always different from correctly transmitted packets. The explicit occurrence of a jamming signal is semantically equivalent to a collision. This interpretation is assumed for historical compatibility with CSMA/CD protocols, which use jamming signals to enforce the so-called collision consensus.

The meaning of the word "garbled" depends on the link type. For types `BLink` (also `Link`), `ULink`, and `PLink` (section 6.2), a packet is garbled if another activity is heard on the port together with the packet or its fragment. The collision begins at the first moment when the port starts sensing two or more packets at the same time, or a jamming signal possibly mixed with other activities. The collision ends as soon as the port gets into a state where it senses at most one packet transmission and no jamming signal.

With the link types mentioned above, it is possible for two short packets to pass through each other and arrive at their destinations undamaged. All natural links, e.g., radio channels, cables, fiber optics, behave in this manner, which is a straightforward consequence of the wave mechanics governing electromagnetic phenomena.

In a link belonging to type `CLink` (section 6.2), collisions spread, in the sense that when two packets meet in any place of the link, the interference propagates (as a kind of activity

of its own) and is perceived in due time by all ports connected to the link. Note that type `CLink` represents bidirectional links. For a unidirectional link, the concept of propagating collisions makes no sense.

11.2.2 Event priorities

Sometimes it is necessary to assign different priorities to different events that can occur simultaneously on the same port. Let us consider the following fragment of a process code:

```
...
state SomeState:
    MyPort->wait (BOT, NewPacket);
state NewPacket:
    MyPort->wait (EMP, MyPacket);
    MyPort->wait (EOT, OtherPacket);
    MyPort->wait (SILENCE, Garbage);
...
```

Most likely, the intended interpretation of the wait requests issued in state `NewPacket` is as follows:

> Upon detection of the end of a packet addressed to this station we want to get to state `MyPacket`; otherwise, if we get the end of a packet addressed to some other station, we want to continue at `OtherPacket`; finally, if the packet does not terminate properly (it has been aborted), we want to resume in state `Garbage`.

The problem is that the three awaited events may occur (and usually occur) at the same time. Therefore, special measures must be taken to enforce a specific order in which the events are to be triggered.

If the program has been created with `-p` (section 19.2), it is possible to use the third argument of the wait request to specify the ordering of the awaited events, e.g.,

```
...
state SomeState:
    MyPort->wait (BOT, NewPacket);
state NewPacket:
    MyPort->wait (EMP, MyPacket, 0);
    MyPort->wait (EOT, OtherPacket, 1);
    MyPort->wait (SILENCE, Garbage, 2);
...
```

The problem mentioned above is rather common. Therefore, if explicit event priorities are not available (the program was created without `-p`), SIDE implicitly prioritizes port events to reflect their intuitive order of importance illustrated in the example. This ordering is not automatically imposed if the program has been compiled with `-p`: then it is assumed that the user wants to exercise full control over ordering events.

Suppose that the SIDE program was created without `-p`, i.e., the explicit event ordering is unavailable. Imagine that a packet arrives at a port and there is a single process awaiting three different events that may be caused by the same packet arrival, namely: `BMP`, `BOT`, and `ACTIVITY`. The `BMP` event has the highest priority, i.e., it is the one that will be triggered, if the packet happens to be addressed to the station running the process. On the other hand, `ACTIVITY` is the lowest-priority event and is only triggered when none of the two higher-priority events occurs. This rule only applies when a single packet is heard on the port and there is no other (interfering) activity that may cause similar events.

Similarly, the events `EMP`, `EOT`, and `SILENCE` are assigned priorities such that `EMP` is the highest-priority event, whereas `SILENCE` has the lowest priority. Again, these priorities only apply if the events are caused by a single packet passing by the port. In particular, in the example discussed above, the order in which the three events will be presented to the awaiting process agrees with the intended interpretation.

11.2.3 Receiving packets

By sensing `BMP` and `EMP` events a process can recognize packets that are addressed to its station. A packet triggering one of these events must be either a non-broadcast packet whose `Receiver` attribute contains the `Id` of the current station, or a broadcast packet with `Receiver` pointing to a station group including the current station.

Another way to determine whether a given packet is addressed to a specific station is to call this method of `Packet`:

```
int isMy (Station *s = NULL);
```

which returns nonzero if and only if the indicated station is the packet's receiver (or one of the receivers in the broadcast case). If the argument is NULL (or absent), the inquiry refers to the station pointed to by `TheStation`, i.e., to the station that runs the inquiring process.

Typically, the receiving process waits until the end of the packet arrives at the port (event `EMP`), before assuming that the packet has been received completely. Then the process can access the packet via the environment variable `ThePacket` (section 11.2) and examine its contents. The most often performed operation after the complete reception of a standard packet (section 10.2) is

```
Client->receive (ThePacket, ThePort);
```

The purpose of this operation is to update the performance measures associated with the traffic pattern to which the packet belongs and with the link on which the packet has been received. The semantics of `receive` are discussed in sections 14.2 and 14.3. From the viewpoint of this section it is enough to say that `receive` should always be executed for a standard packet (for which performance measures are to be collected) as soon as, according to the protocol rules, the packet has been completely and successfully received at its final destination.

The following variants of `receive` are declared within `Client`:

```
void receive (Packet *pk, Port *pr);
void receive (Packet *pk, Link *lk = NULL);
void receive (Packet &pk, Port &pr);
void receive (Packet &pk, Link &lk);
```

The second version accepts a link pointer rather than a port pointer and the last two versions are equivalent to the first two, except that instead of pointers they take objects as arguments. If the second argument is not specified, it is assumed to be `NULL` which means the the link performance measures will not be affected by the packet's reception.

Exactly the same collection of `receive` methods is declared within type `Traffic`. The `Traffic` methods operate exactly as the corresponding `Client` methods, with an additional check whether the received packet belongs to the indicated traffic pattern. Thus, they also act as simple assertions.

All versions of `receive` check whether the station whose process is executing the operation is authorized to receive the packet, i.e., if the packet is addressed to the station. A packet whose `Receiver` attribute is `NONE` can be received by any station.

Below we list a sample code of a process responsible for receiving packets.

```
perform {
    state WaitForPacket:
        RcvPort->wait (EMP, NewPacket);
    state NewPacket:
        Client->receive (ThePacket, ThePort);
        skipto WaitForPacket;
};
```

Instead of `ThePort`, `RcvPort` could be used as the second argument of `receive`—with the same results. After returning from `receive`, the process uses `skipto` (section 8.1) to resume waiting for a new packet. Note that if `proceed` were used instead of `skipto`, the process would loop infinitely. This is because the process would transit to `WaitForPacket` without advancing the simulated time and the previous `EMP` event would still be present on the port; it would restart the process immediately—within the same *ITU* and for the same packet.

The `Receiver` attribute of a packet can be interpreted in two ways: as the `Id` of a single station or as a station group pointer (section 10.2). The latter interpretation takes place for a broadcast packet. To tell whether a packet is a broadcast one, the receiver can examine its flag number 3 (`PF_broadcast`—section 10.2) which is set for broadcast packets. The following two packet methods:

```
int isBroadcast ();
int isNonbroadcast ();
```

return the broadcast status of the packet without referencing this flag explicitly.

Another flag (number 30 or `PF_last`) is used to mark the last packet of a message. It is possible (see section 10.8.1) that a single message is split into a number of packets which are transmitted and received independently. If the `PF_last` flag of a received packet is set, it means that the packet is the last (or the only) packet acquired from its message. This flag is used internally by SIDE in calculating message delay (section 14.2.1). The following two packet methods:

```
int isLast ();
int isNonlast ();
```

determine whether the packet is the last packet of a message without referencing the `PF_last` flag directly.

11.3 Port inquiries

By issuing a port wait request a process declares that it would like to learn when something happens on the port in the future. It is possible to ask a port about its present status, i.e., to determine the configuration of activities currently perceived by the port, or about its past status—within the time interval determined by the link's *archival period* (see sections 6.2.2, 11.1.1).

11.3.1 Inquiries about the present

The following port method:

```
Boolean busy ();
```

returns nonzero if the port is currently perceiving any activity (a packet transmission, a jamming signal, or a number of overlapping activities), and zero otherwise. There also exists a complementary method,

```
Boolean idle ();
```

which is a simple negation of `busy`.

In the case when a number of activities are heard at the port at the same time, it is possible to determine their types by calling one of the following port methods:

```
int activities (int &t, int &j);
int activities (int &t);
int activities ();
```

With the first method, the two reference arguments return the number of packet transmissions (`t`) and the number of jamming signals (`j`) sensed by the port within the current *ITU*. The method's value tells the total number of all those activities, i.e., the sum of `t` and `j`. The second method does not explicitly return the number of jams; however, the method's value still gives the total number of all activities. The last method has no arguments and it merely returns the total number of all activities perceived at the port.

In the case when exactly one packet transmission is sensed by the port and no other activity is heard at the same time, each of the above methods sets the environment variable `ThePacket` (section 11.2) to point to the packet object. `ThePort` is always set to point to the port to which the inquiry was addressed.

When a process is awakened by `ANYEVENT` (section 11.2), it may happen that two or more simultaneous events have triggered the waking event. It is possible to learn how many events of a given type occur on the port at the current moment by calling this method of `Port`:

```
int events (int etype);
```

where the argument identifies the event type in the following way:

`BOT`	beginnings of packet transmissions
`EOT`	endings of packet transmissions[31]
`BOJ`	beginnings of jamming signals
`EOJ`	endings of jamming signals

The method returns the number of events of the type indicated by the argument.

By calling `activities` (see above) a process can learn the number of packets that are simultaneously heard on a port. Sometimes the process may wish to examine all these packets. The following port method can be used in such situations:

```
Packet  *anotherPacket ();
```

[31] Only the transmissions terminated by `stop`—section 11.1.2—are counted.

The method can be called after receiving one of the events `BOT`, `EOT`, `BMP`, `EMP`, or after calling `events` with the second argument equal `BOT` or `EOT`, or after calling `activities`.

When the method is called after the process has been awakened by one of the four packet events, its subsequent calls return pointers to all packets triggering the event at the same time. These pointers are returned both via the function value and in `ThePacket`. If `NULL` is returned, it means that there are no more packets, i.e., all of them have been examined. If there are n packets that cause a given event simultaneously (within the same ITU), n calls to `anotherPacket` are required to look at all of them.

When called after `events`, the method behaves exactly as if the process were awakened by `BOT` or `EOT`, depending on the argument of `events`. After a call to `activities`, the method returns (in its subsequent calls) pointers to all packets currently perceived on the port.

11.3.2 Inquiries about the past

In a real network, it is impossible to ask a port about its past status. In SIDE this possibility has been provided to simplify implementations of some protocols. Of course, it is always possible to re-program a protocol in such a way that no port inquiries about the past are made: processes can maintain their private data bases of the interesting past events.

Below, we present the port methods that perform inquiries about the past. All of them return values of type `TIME` and take no arguments, so we just list their names.

`lastBOA`
: The method returns the time when the last beginning of (any) activity (the end of the last silence period) was perceived on the port. `TIME_0` is returned if no activity has been heard on the port within the archival period. The method can also be referenced as `lastEOS` (for "last End Of Silence").

`lastEOA`
: The method returns the time of the beginning of the current silence period on the port (the last End Of Activity). `TIME_inf` (meaning "undefined") is returned if an activity is currently heard on the port. If no activity has been observed on the port within the archival period, the method returns `TIME_0`. The two port methods `busy` and `idle` (section 11.3.1) are in fact macros that expand into `undef(lastEOA())` and `def(lastEOA())`, respectively. The `lastEOA` method can also be called as `lastBOS` (for "last Beginning Of Silence").

`lastBOT`
: The method returns the time when the last beginning of packet was heard on

the port. TIME_0 is returned if no beginning of packet has been noticed within the link archival period. The event is unaffected by other activities that may be overlapping with the packet.

lastEOT
: The method returns the time when the last end of a complete packet transmission (i.e., terminated by stop—see section 11.1.2) was perceived on the port. TIME_0 is returned if no end of packet has been noticed on the port within the archival period. Similar to lastBOT, other activities occurring at the same time have no impact on the event, as long as the packet was terminated by stop.

lastBOJ
: The method returns the time when the last beginning of a jamming signal was heard on the port. Overlapping jams as recognized as a single continuous signal, so only the beginning of the earliest of them is detected. TIME_0 is returned if no jamming signal has been perceived on the port within the archival period.

lastEOJ
: The method returns the time when the last end of a jamming signal was heard on the port. TIME_inf is returned if a jamming signal is currently present. If no jam has been observed on the port within the archival period, the method returns TIME_0.

lastCOL
: The method returns the time of the beginning of the last collision sensed by the port. By the beginning of a collision we mean the first moment when the collision was recognized (see section 11.2.1). TIME_0 is returned if no collision has been perceived on the port within the archival period.

The history of the past activities in a link is kept in the link archive (sections 6.2.2, 11.1.1) for the amount of time specified upon the link's creation. If no inquiries about the past are ever issued to a link (i.e., to a port connected to the link), the link's archival period should be set to 0, which will reduce the simulation time. Otherwise, the archival time should be big enough to make sure that the protocol works correctly, but not too big—to avoid overloading the simulator with a huge number of archived activities.

11.4 Faulty links

It is possible to implant into a link a faulty behavior, whereby packets transmitted over it may be be damaged with a certain probability. A damaged packet is marked in a special

way and this fact can affect the interpretation of certain events and the outcome of certain inquiries. The following link method declares a fault rate for the link:

```
setFaultRate (double rate, int ftype);
```

The first argument specifies the probability of error for a single packet bit, i.e., the bit error rate (BER). Its value must be between 0 and 1; usually it is much less than 1. The second argument selects the *processing type* for invalid packets—as explained below.

By default, i.e., before `setFaultRate` is called, the link is error-free and packets inserted into it are never damaged.[32] A faulty link can be reverted to its error-free status by calling `setFaultRate` with the second argument equal `NONE` (the value of the first argument is then irrelevant).

The valid/invalid status of a packet is stored in the packet's `Flags` attribute (section 10.2). Two flags are used for this purpose, and the packet can be damaged in the following two ways:

- The packet's header is unrecognizable and the packet's destination cannot be (formally) determined. This status is indicated by two flags `PF_hdamaged` (flag 28) and `PF_damaged` (flag 27) being set simultaneously.
- The packet's header is correct, but the the packet is otherwise damaged (e.g., its trailer checksum is invalid).

Note that a packet that is *header-damaged* is also *damaged*, but not *vice-versa*.

The damage status of a packet can always be determined directly, by calling any of the following packet methods:

```
int isDamaged ();
int isValid ();
int isHDamaged ();
int isHValid ();
```

Each of the above methods returns either `YES` or `NO`—in the obvious way. Note that `isHDamaged` implies `isDamaged`; similarly, `isValid` implies `is isHValid`.

Depending on the value of the second argument of `setFaultRate` (`ftype`), the damaged status od a packet can be interpreted automatically, by suppressing certain events that are normally triggered by a valid packet. In addition to `NONE` (which makes the link error-free), the following three values of `ftype` are legal:

[32] Of course, the protocol program is still able to detect collisions and packets that have been aborted by the sender.

FT_LEVEL0
: No automatic processing of damaged packets is in effect. Damaged packets trigger the same events and respond to the same inquiries as valid packets. The only way to tell a damaged packet from a valid one is to examine the packet's Flags attribute, e.g., with one of the above methods.

FT_LEVEL1
: A header-damaged packet does not trigger the BMP event. A damaged packet does not trigger the EMP event (section 11.2). This also applies to anotherPacket (section 11.3.1) called after BMP (EMP), which ignores header-damaged (damaged) packets. The packet method isMy (section 11.2.3) returns NO for a header-damaged packet (but works as usual if the packet is damaged without being header-damaged).

FT_LEVEL2
: The interpretation of damaged packets is as for FT_LEVEL1, but BOT and EOT events are also affected. Thus, a header-damaged packet does not trigger the BOT event and a damaged packet does not trigger the EOT event. This also applies to anotherPacket and to the link inquiries about past BOT/EOT events (section 11.3.2).

Regardless of the processing type for damaged packets, events SILENCE, ACTIVITY, COLLISION, BOJ, EOJ, and ANYEVENT (section 11.2) are not affected by the damage status of a packet. Note that jamming signals are never damaged.

The standard link method used by SIDE to generate damaged packets is declared as "virtual"; therefore, it can be overridden by the user in a Link subtype declaration. The header of this method looks as follows:

```
virtual void packetDamage (Packet *p);
```

The method is called for each packet given as an argument to transmit (section 11.1.2) to decide whether it should be damaged or not. First, based on the length of the packet header (p->TLength-p->ILength), packetDamage determines the probability that the packet will be header-damaged. This probability is equal to

$$1 - (1 - \texttt{rate})^h,$$

where h is the header length in bits. A uniformly distributed random number between 0 and 1 is then generated, and if its value is less than the damage probability, the packet is marked as both header-damaged and damaged (the flags PF_hdamaged and PF_damaged are both set). Otherwise, the probability of the packet being damaged (without being header-damaged) is determined based on the packet's ILength attribute and another random

number is generated—in the same way as before. If this number turns out to be less than the damage probability for the packet's payload, the packet is marked as damaged (`PF_damaged` is set, but `PF_hdamaged` is cleared). If the packet is not to be damaged at all, both damage flags are cleared.

Note: The possibility of declaring links as "faulty" is switched off by default—to avoid simulation overhead. To enable faulty links, the program must be created with the `-z` option of `mks` (section 19.2).

11.5 Cleaning after packets

Sometimes you would like a packet to carry a pointer to a dynamically allocatable data structure that must be deallocated when the packet disappears from the network. This issue was discussed in section 10.6.4, where tools for cleaning up after standard (i.e., `Client`) packets (and messages) were presented. It is possible to perform a similar kind of operation at the link level—in a way that makes it applicable to all packets (including non-standard ones) that complete their life time within the link (or its archive—section 11.1.1). With this `Link` method:

```
void setPacketCleaner (void (*)(Packet*));
```

you can plug[33] into the link a function to be called for every packet disappearing from it. By examining the packet's `TP` attribute (section 10.2) and casting the packet to the proper type, you can then selectively perform the required cleaning duties, e.g.,

```
void cleanPackets (Packet *p) {
    if (p->TP == -1)
        delete ((HelloPacket*)p)->NeighborList;
}
...
EtherLink->setPacketCleaner (cleanPackets);
...
```

Note that the cleaner function is global. Needless to say, one has to be aware that the data structure deallocated by the cleaner is volatile. Thus, if the packet's recipient needs to preserve it beyond the life time of the packet's activity in the link, it must make itself a copy. Also, if traffic-level cleaning is used together with link-level cleaning, you should make sure that the same packets are not cleaned twice.

[33] As link types cannot be extended by the user, virtual functions cannot be applied here. Extending a link type to accommodate this single feature into a virtual method would be (arguably, we admit) an overkill.

By calling `setPacketCleaner` with a `NULL` argument, you remove the packet cleaner function from the link.

Note that for a packet belonging to a traffic pattern (i.e., one generated and absorbed by the `Client`—sections 10.2, 10.6.1), packet deallocation can be intercepted with a virtual method defined within the traffic type to which the packet belongs (section 10.6.4). Thus, a link-level interceptor is only recommended for those situations when either all or most packets must be processed that way, or special packets (i.e., ones that do not originate in the `Client` and have no associated traffic objects) must be subjected to the procedure. Beware that, unlike the cleaning methods associated with traffic patterns, link-level cleaners are only invoked when a packet disappears from the link or its archive. They are not called when a packet is deallocated the standard way, i.e., with explicit `delete`.

12 Radio channels and the `Transceiver` *AI*

The role of transceivers is similar to that of ports; thus, transceivers share many methods with ports and trigger similar events. The kinds od operations available on transceivers cover:

- starting and terminating packet transmissions
- inquiring about the present status of the transceiver, i.e., its perception of the radio channel
- wait requests, e.g., to await an event related to a packet arrival

One difference with respect to ports is that transceivers implement no methods for inquiring about the past (section 11.3.2). This results from the fact that, unlike links, radio channels maintain no archives of past activities, i.e., ones that have "physically" disappeared from them.

12.1 Interpreting activities in radio channels

Essentially, there is only one type of activity that can be inserted into a transceiver, i.e., a packet transmission.[34] A packet is always preceded by a *preamble*, which consists of a specific number of "physical" bits. These bits do not count to the total length of the packet (attribute `TLength`—section 10.2), but they do take bandwidth, i.e., time to transmit. A preamble can be as short as one bit, but its length cannot be zero. The preamble length can be defined when the transceiver is created (section 6.5.1), or re-defined at any time with `setPreamble` (section 6.5.2). Different transceivers can use preambles of different length.

[34] In contrast to ports (section 11.1), there are no jamming signals.

12.1.1 The stages of packet transmission and perception

A packet is transmitted, and also perceived[35] at those transceivers that it will ever reach, in three stages, which mark the timing of the possible events that the packet may ever trigger on a perceiving transceiver:

1. *Stage P*: the beginning of preamble. This stage marks the moment when the first bit of the preamble is transmitted at the source or arrives at a perceiving transceiver.

2. *Stage T*: the end of preamble, which usually coincides with the beginning of packet. This is the moment when the last bit of the preamble has been transmitted (or perceived) and the first bit of the actual packet appears.

3. *Stage E*: the end of packet representing the moment when the last bit of the packet has been transmitted (or disappeared from a perceiving transceiver).

The timing of the three stages at the transmitter is determined by the transceiver's transmission rate (`TRate`) and the lengths of the corresponding packet components. If stage P occurs at time t, then stage T will take place at t + `tcv->TRate * tcv->Preamble`, where `tcv` is the transmitting transceiver and `Preamble` is its currently defined preamble length. Then, stage E will occur `RFC_xmt (tcv->TRate, pkt->TLength)` ITUs after stage T, where `pkt` points to the packet being transmitted. Recall that `TRate` is a transceiver attribute representing the number of ITUs required to transmit a single bit (section 6.3.1), and `RFC_xmt` is one of the "assessment methods" of the radio channel to which the transceiver is interfaced (section 6.4.2).

Note that the default version of `RFC_xmt` (section 6.4.2) calculates the packet transmission time as `tcv->TRate * pkt->TLength`, assuming a straightforward interpretation of the packet's bits as the "actual" bits to be sent over the channel at the specified rate. This need not always be the case. In many encoding schemes, the "logical" packet bits (represented by `pkt->TLength`) are transformed into symbols expressed as sequences of "physical bits." For example, those symbols may balance the number of physical zeros and ones and/or provide for forward error correction (FEC). It is the responsibility of `RFC_xmt` to account for such features and determine the correct timing for sending the packet contents over the channel. The transmission rate (`tcv->TRate`) usually applies to the physical bits. The preamble, however, is treated separately, in the sense that its length is always assumed to be in physical bits, i.e., its is directly multiplied by the rate to yield the separation between stages P and T. Thus, it should be viewed as a timing parameter rather than a sequence of some bits, which need not be related to the bits constituting the packet content.

[35] We use the word "perception" instead of "reception" to denote the fact that the packet is noticed at all by the transceiver. This means that the packet is audible and its signal may be recognized and, for example, may interfere with other packets, but need not mean that the packet can be received. The word "reception" will be used to refer to the latter circumstance.

Formally, the time separation of the three stages at any perceiving transceiver reached by the transmitted packet is identical to their separation at transmission. The delay of their appearance at the receiver, with respect to the transmission time, is equal to the distance in *ITU*s separating the receiver from the transmitter. This distance, in turn, is determined by the Euclidean coordinates of the two transceivers (section 6.5). For illustration, suppose that the packet is transmitted by transceiver v with coordinates $< x_v, y_v >$.[36] Let w, with coordinates $< x_w, y_w >$, be a neighbor of v perceiving the packet. If stage P of the packet occurs at time t at v (i.e., the first bit of the preamble is transmitted by v at time t), then it will occur at w at time $t + \sqrt{(x_w - x_v)^2 + (y_w - y_v)^2}$, assuming that the transceiver coordinates are expressed in *ITU*s.[37] The same time transformation concerns the remaining two stages of the packet, i.e., their occurrence at w is offset by $\sqrt{(x_w - x_v)^2 + (y_w - y_v)^2}$ with respect to v.

12.1.2 Criteria of event assessment

A packet being transmitted on a given transceiver may contribute to the overall signal level perceived by the same transceiver (its receiver), but it never triggers the standard packet events (`BOP`, `BOT`, `EOT`, and so on—see section 12.2.1). This means that a packet transmitted from a given transceiver cannot be (automatically and inadvertently) received by the same transceiver.

The extent to which a transmitted packet is going to affect the activities perceived by other transceivers (and, in particular, if it is going to be received as a valid packet) is described by the collection of user-redefinable (virtual) methods of the underlying radio channel (`RFChannel`). Those methods (see section 6.4.2 for the complete list of their headers) can base their decisions on the following criteria:

The transmission power. This is a `double` attribute of the transmitting transceiver, which can be set with `setXPower` (see section 6.5.2).

The receiver gain. This is a `double` attribute of the perceiving transceiver, which can be set with `setRPower` (see section 6.5.2).

The tag. This is an `IPointer`-type (section 3.1) attribute of the transmitting transceiver, which can be set with `setTag` (see section 6.5.2). Its role is that of a general-purpose discerning feature for packet transmissions, whose interpretation is entirely up to the user. For example, different tags can represent different codes in CDMA channels.

The distance. This is the Cartesian distance in *DU*s separating the transmitting transceiver from the receiving one (section 6.5.1).

The interference histogram. This is a representation of the complete *interference history* of the packet. It describes how much other signals perceived by the transceiver

[36] Of course, this illustration can be trivially extended to three dimensions.

[37] They are internally; however, the user specifies them in *DU*s (see section 6.5.2).

at the same time as the packet have interfered with the packet at various moments of its perception.

Two of the assessment methods, `RFC_erb` and `RFC_erd`, facilitate converting user-defined distributions of error-bit rates (typically dependent on the signal to interference ratio) into conditions and events determining the success or failure of a packet reception. These methods are optional: they provide convenient shortcuts, e.g., available to other assessment methods like `RFC_bot` and `RFC_eot`, and help implement tricky events whose timing depends on the distribution of bit errors in received packets. Note that the *bits* referred to above are physical bits (see section 12.1.1).

Owing to the fact that the assessment methods are (easily) exchangeable by the user, the radio channel model provided by SIDE is truly open-ended. The standard versions of those methods (section 6.4.2) implement a very naive and practically useless channel model. In contrast to wired channels, the large population of various propagation models and transmission/reception techniques makes it impossible to satisfy everybody with a few complete built-in models. Consequently, SIDE does not attempt to close this end. Instead, it offers a simple and orthogonal collection of tools for describing different channel models with a minimum effort.

12.1.3 Neighborhoods

For each transceiver, SIDE maintains its *neighborhood*, i.e., the population of transceivers (interfaced to the same `RFChannel`) that may perceive packets transmitted from the given transceiver. The configuration of neighborhoods is dynamic and may change in the following circumstances:

1. A transceiver changes its location. In such a case, the neighborhood of the moving transceiver may change; also the neighborhoods of other transceivers may have to be modified by either including or excluding the moved transceiver.

2. A transceiver changes its transmission power (attribute `XPower`). If the power increases, the neighborhood of the transceiver may include more transceivers; if it decreases, the neighborhood may have to be trimmed down.

3. A transceiver changes it reception gain (attribute `RPower`). If the gain increases, the transceiver may have to be included in some new neighborhoods; if it decreases, the transceiver may have to be removed from some neighborhoods.

The actual decisions as to which transceivers should be included in the neighborhood of a given transceiver are made by invoking `RFC_cut` (section 6.4.2). The method returns the maximum distance in *DU*s on which a signal transmitted at a given power (the first argument) will be perceived by a receiver operating at a given gain (the second argument).

Note that the method can be conservative, i.e., the model should remain formally correct if the neighborhoods are larger than they absolutely have to be. In particular, one may be able to get away with a "global village," i.e., a single neighborhood covering all transceivers (RFC_cut returning Distance_inf). The primary advantage of trimmed neighborhoods is the reduced simulation time and, possibly, reduced memory requirements of the simulator. This advantage is generally quite relevant, especially for networks consisting of hundreds and thousands of transceivers.

Note that one more circumstance (ignored in the above list) potentially affecting the neighborhood status of a pair of transceivers is a tag change (in one or both of them). This, however, has been intentionally eliminated from the scope of RFC_cut (see section 6.4.2), which method deals with the pure signal level (plus "pure" receiver gain) ignoring the tags of the assessed parties. This is because otherwise any change in a tag would require a complete reconstruction of the neighborhoods, which would be a rather drastic and time consuming action. Consequently, if tags are used by the channel model, RFC_cut should assume that their configuration is always most favorable from the viewpoint of signal propagation, i.e., the returned neighborhood distance should be the maximum over all possible values of tags.

12.1.4 Event assessment

At any moment, a given transceiver may perceive a number of packets arriving from different neighbors and being at different stages. The role of the assessment procedure at a receiving transceiver is to determine whether any of those packets should be received or, more specifically, whether it should trigger the events that will conceptually amount to its reception. The decisions are arrived at by the collective interaction of the user-exchangeable *assessment methods* listed in section 6.4.2.

First, RFC_att is called to determine the signal level of the packet at the perceiving port. Its first argument describes the transmission power at which the packet was originally transmitted at the sending transceivers, as well as the tag of that transmission, the second argument gives the distance between the sender and the perceiving transceiver, and the last argument points to the sending transceiver. The receiving (destination) transceiver can be accessed through the environment variable TheTransceiver (Info02).

If all transmissions are homogeneous, then the tags are not needed. In a distance-based propagation model, the method may use a (randomized) function of the original power level and the distance to produce the attenuated signal level. For one example of a scenario involving tags, consider multiple channels with nonzero crosstalk. With the tags representing the different channels, RF_att may apply an additional factor to the signal depending on the difference between the channel numbers.

Note that, although the sender transceiver is directly available to RFC_att (via the third argument), the method should avoid consulting its attributes unless absolutely necessary.

In particular, instead of using the `Tag` value from the first `SLEntry*` argument, the method may be tempted to invoke `getTag` for the sender's transceiver. However, these two values can be legitimately different, as the tag of the perceived signal was associated with it in the past (when the packet was transmitted), and the current tag setting of the sender's transceiver need not be the same. On the other hand, the attributes of the receiving transceiver (its pointer is available via the environment variable `TheTransceiver`) are up to date and they apply directly to the assessed situation.

The signal level returned by `RFC_att` is associated with the packet at its perceiving transceiver. Suppose that some transceiver v perceives n packets denoted $p_0, \ldots, p_{n-1}$ with $r_0, \ldots, r_{n-1}$ standing for their received signal levels. The interference level suffered by one of those packets, say p_k is determined by a combination of all signals of the remaining packets. Method `RFC_add` is responsible for calculating this combination. Generally, the method takes a collection of signal levels and returns their combined signal level.

One should remember that the signals presented to `RFC_add` (in the array passed in the third argument) have been already attenuated, i.e., they are outputs of `RFC_att`. Consequently, although the tags of those signals are available to `RFC_add`, the method should not apply to them the same kind of "factoring" as prescribed for `RFC_att`, although, of course, it may implement any idiosyncratic way of adding those signals required by the model.

If the second argument of `RFC_add` is nonnegative, it should be viewed as the index of one entry in the signal array that must be ignored. This is exactly what happens when the method is called to calculate the interference suffered by one packet. The collection of signals passed to the method in the third argument always covers the complete population of signals perceived by the transceiver. Thus, the indicated exception refers to the one signal for which the interference produced by the other signals is to be determined. Sometimes `RFC_add` is invoked to calculate the global signal level caused by all perceived packets with no exception. In such a case, the second argument is `NONE`, i.e., –1.

In all cases, the `Tag` element of the last argument of `RFC_add` refers to the current value of `Tag` at the perceiving transceiver. The `Level` component of that argument is zero, unless the transceiver is currently transmitting a packet of its own, in which case it contains the transceiver's transmission power.

The actual decision regarding a packet's reception is made by `RFC_bot` and `RFC_eot`. Their meaning is similar, but they are called at different stages of the packet's perception. `RFC_bot` is invoked at stage T, i.e., at the end of preamble and before the first bit of the actual packet. It receives the following arguments:

- the transmission rate `r` at which the packet was transmitted
- the perceived signal level of the packet along with the tag `sl`
- the reception gain of the perceiving transceiver combined with the transceiver's tag

rs

- the interference histogram of the preamble ih

The method returns a Boolean value: YES if the perceived quality of the preamble makes it possible to start the packet's reception, NO if the packet cannot be received. In physical terms, this decision determines whether the receiver has been able to recognize that a packet is arriving and, based on the quality of preamble, clock itself to the packet. If the decision is NO, the packet will not be received. In particular, its next stage (E) will not be subjected to another assessment, and the packet will trigger no reception events (section 12.2.1). If the method returns YES, the packet will undergo another assessment at stage E—regarding the successful reception of its last bit. This second assessment will be handled by RFC_eot. In any case (also following a negative assessment by RFC_bot) the packet will continue contributing its signal to the population of activities perceived by the transceiver until stage E, i.e., until the packet is no more heard at the transceiver.

Note that RFC_eot has exactly the same header as RFC_bot. RFC_eot is called in stage E to decide whether the packet has been successfully received (or rather is receivable, as the actual reception must be performed explicitly by the protocol program). The idea is exactly the same as for the preamble, except that this time the interference histogram covers the proper packet, i.e., the part following the preamble.

12.1.5 The interference histogram

The interference histogram passed as the last argument to RFC_bot concerns solely the preamble component of the packet. On the other hand, the histogram passed to RFC_eot applies to the packet proper, excluding the preamble. In both cases, the histogram describes the complete interference history of the respective component as a list of different interference levels suffered by it along with the intervals during which it remained the same.

The interference histogram is a class comprising several useful methods and two public attributes:

```
int NEntries;
IHEntry *History;
```

where History is an array of size NEntries. Each entry in the History array is a simple record that looks like this:

```
typedef struct {
    TIME Interval;
    double Value;
} IHEntry;
```

The sum of all `Intervals` in `History` is equal to total duration (in *ITU*s) of the activity being assessed. The corresponding `Value` attribute of the `History` entry gives the interference level (as calculated by `RFC_add`) suffered by the activity within that interval. The complete `History` covers as many intervals as many different interference levels have happened during the perception of the assessed activity. If the preamble or packet has suffered absolutely no interference, its `History` consists of a single entry whose `Interval` spans its entire duration and whose `Value` is 0.0.

Below we list the public methods of `class IHist`, which may be useful for examining the contents of the interference history.

```
void entry (int i, TIME &iv, double &v);
void entry (int i, double &iv, double &v);
void entry (int i, RATE r, Long &iv, double &v);
```

These are three different versions of essentially the same method that returns the contents of a single entry (number `i`) from the interference histogram. In all three cases, the last argument returns the `Value` attribute from the `i`-th entry. The first variant returns the `Interval` in *ITU* (via the second argument), while the second variant converts the `Interval` to *ETU*s and returns it as a `double` number. The third version takes four arguments. It converts the `Interval` to the number of bits based on the transmission rate specified as the second argument. The value returned in `iv` is equal to `Interval/r`.

In the last case, if `Interval` is not divisible by `r`, the actual number of bits returned in `iv` is either $\lfloor$`Interval/r`$\rfloor$ or $\lfloor$`Interval/r`$\rfloor + 1$, at random, with the probability of the latter equal to the fractional part of the exact result. In other words, the average result over an infinite number of calls to `entry` is equal to the exact ratio `Interval/r`.

```
TIME duration ();
```

This method returns the total duration of the activity represented by the histogram, i.e., the sum of all `Intervals`, in *ITU*s.

```
Long bits (RATE r);
```

This method returns the total duration of the activity represented by the histogram, i.e., the sum of all `Intervals`, in bits, based on the specified transmission rate `r`. The result is obtained by dividing the value returned by `duration` by `r` and, if the fractional part is nonzero, applying the randomized adjustment described at `entry`.

```
double avg ();
double avg (TIME ts, TIME du = TIME_inf);
double avg (RATE r, Long sb, Long nb = -1);
```

These methods return the average interference level within the specified interval of the histogram:

1. Over the entire histogram. The average is taken over time not over entries, i.e., longer intervals contribute proportionately to the average.

2. Over the specified time: `ts` is the starting time in *ITU*s counting from zero (the beginning of the history), and `du` is the duration (also in *ITU*s). If the interval exceeds the duration of the histogram, the excessive component is ignored, i.e., it does not count to the time over which the average is taken. If the interval falls entirely behind the histogram's duration, the returned average interference is zero. If the second argument is not specified (or is `TIME_inf`), it stands for the end of the histogram's duration. If the first argument is `TIME_inf`, the second argument is interpreted as the interval from the end of the histogram's duration, i.e., the last `du` *ITU*s of the activity are examined.

3. Starting at bit `sb` and extending for `nb` bits. The first bit of the histogram is numbered 0. Intervals extending behind the histogram are treated as in the previous case. Argument `r` specifies the transmission rate needed to transform time intervals into bits. If the last argument is not specified (or is negative), all the bits of the histogram starting at bit number `sb` are examined. If `sb` is negative, `nb` bits are scanned from the end of the activity represented by the histogram.

```
double max ();
double max (TIME ts, TIME du = TIME_inf);
double max (RATE r, Long sb, Long nb = -1);
```

The arguments are exactly as for the `avg` methods above. These methods return the maximum interference over the specified time fragment of the histogram.

12.1.6 Event reassessment

In some advanced reception models, the conditions determining the calculation of signal levels, interference, the bit error rate and, consequently, event assessment for `BOT` and `EOT` may change half way through a packet reception. We do not consider here mobility, whose impact is only re-evaluated at activity (packet) boundaries. In all realistic mobility scenarios, the amount of distance change during the reception of a single packet is completely negligible.

For an example of what we have in mind in this section, suppose that the receiver uses a directional antenna that can be reset instantaneously at will. The right way of implementing this operation in the channel model is to use a dynamic gain factor in `RFC_att` (sections 6.4.2, 12.1.4) depending on the antenna angle and the location of the transmitter

and the receiver. When the antenna setting changes, the transceiver must re-evaluate all signals reaching it at the moment based on the new setting. This is accomplished with this `Transceiver` method:

```
void reassess ();
```

which is to be called in all circumstances where the perception of signal levels by the receiver may have changed "behind the scenes." Note that standard situations, like, for example, changing the receiver gain with `setRPower` (section 12.1.2), are handled automatically by the simulator. By "behind the scenes," we mean "in a way not naturally perceptible by the simulator," which practically always means "affecting the results produced by `RFC_att`."

Note that directional transmission is different in this respect. This is because (at least in all sane schemes), the transmitter sets the antenna before sending out a packet, and the entire packet is sent out with the same antenna setting. Also note that an obsessive experimenter may use this feature to implement "extreme fidelity" mobility models, whereby signals at the receiver are reassessed after a minuscule distance change occurring half-way through a packet reception. Needless to say, `reassess` is a rather costly method and its frivolous usage is discouraged.

12.1.7 The context of assessment methods

The assessment methods, whose role is described in section 12.1.4, constitute the bulk part of the user-specified channel model. One should be aware that they execute in a context that is somewhat different from that of a protocol process. In this respect, they are unique examples of user-supplied code that does not belong to any process, but instead constitutes a kind of plug-in to the simulator kernel.

However, two standard environment variables, `TheTransceiver` and `ThePacket` (section 12.2.1) are made available to some assessment methods—to provide them with local information that may facilitate the implementation of tricky models. Specifcally, for these methods: `RFC_att`, `RFC_add`, `RFC_act`, `RFC_bot`, `RFC_eot`, `TheTransceiver` points to the transceiver carrying out the assessment. Additionally, for `RFC_att`, `RFC_bot`, `RFC_eot`, `ThePacket` points to the assessed packet. For `RFC_xmt`, `TheTransceiver` identifies the sending transceiver. These variables can be viewed as extra arguments to the respective assessment methods.

12.1.8 Starting and terminating packet transmission

Similar to how this is done for ports (see section 11.1.2), a packet transmission on a transceiver can be started with this method:

```
void startTransfer (Packet *buf);
```

where `buf` points to the buffer containing the packet to be transmitted (section 10.3). A copy of this buffer is made and inserted into a data structure representing a new activity in the underlying radio channel—to be propagated to the neighbors of the transmitting transceiver. The pointer to this packet copy is returned via the environment variable `ThePacket` (`Info01`) of type `Packet*`. Another environment variable, `TheTransceiver` (`Info02`) of type `Transceiver*`, returns a pointer to the transceiver executing the operation. Note that this is different from the `Port` variant of `startTransfer`; this departure is useful for the *assessment methods* of `RFChannel`, notably `RFC_xmt`, which may want to reference the transceiver.

A packet transmission in progress can be terminated in two ways. The following transceiver method:

```
int stop ();
```

terminates the transmission in a way that renders the packet *complete*, i.e., formally equipped with a pertinent trailer that allows the receiving party to recognize the complete packet. Instead of `stop` you can use this method:

```
int abort ();
```

which *aborts* the transmission. This means that the packet has been interrupted in the middle, and it will never be perceived as a complete (receivable) packet. For compatibility with the similar methods available for `Port`s (section 11.1.2) both the above methods return `TRANSFER`. Note that both `stop` as well as `abort` will trigger an error if no packet is being transmitted.

Both methods set two environment variables `ThePacket` (to point to the packet whose transmission is terminated) and `TheTraffic` (to the `Id` of the traffic pattern to which the packet belongs).

The following transceiver method:

```
void transmit (Packet *buf, int done);
```

starts transmitting the packet contained in `buf` (by calling `startTransfer`) and automatically sets up the `Timer` to wake up the process in state `done` when the transmission is complete.

As explained in section 12.1.1, each packet transmitted over a radio channel is preceded by a preamble, whose length cannot be zero. The total amount of time needed to transmit a packet `buf` via a transceiver `tcv` is equal to

```
RFC_xmt (tcv->TRate, buf->TLength) + (tcv->Preamble * tcv->TRate)
```

and this is the amount of time after which the process will wake up in state `done`.

A started transmission has to be terminated explicitly. Thus, the following sample code illustrates the full incantation:

```
...
state Transmit:
    MyTransceiver->transmit (buffer, Done);
state Done:
    MyTransceiver->stop ();
    ...
```

Note that it is identical to the transmission sequence for a port (section 11.1.2).

The following transceiver method is recommended to convert a number of bits to the number of *ITU*s required to transmit these bits on a given port:

```
TIME bitsToTime (int n = 1);
```

Calling

```
t = p->bitsToTime (n);
```

has the effect of executing:

```
t = RFC_xmt (p->TRate, n);
```

Note, however, that various variants of `RFC_xmt` may consider it illegal to be called with certain values of `n`. For example, the method may only work if the bit count is a multiple of 8. Also, it may include a standard "fixed" nonzero component in the returned value, e.g., accounting for the length of a start symbol. Thus, unlike the `Port` variant of this method (section 11.1.2), `Transceiver bitsToTime` comes with caveats and is not recommended for directly converting logical bits to time.

The transmission rate of a transceiver can be changed at any time by calling `setTRate` (see section 6.5.2). If `setTRate` is invoked after `startTransfer` or `transmit` but before `stop`, it has no impact on the packet being transmitted.

Note: It is illegal to transmit more than one packet on the same transceiver at the same time.

The transmission of a packet can be aborted prematurely (by `abort`) during the preamble, i.e., before the first bit of the proper packet has been transmitted. Such a packet is a special

case of sorts: it has a degenerate stage T (the preamble does end but the proper packet never begins), and has no stage E. No assessment will ever be carried out for such a packet: RFC_bot will not be called after the preamble has been received, and the packet will not trigger any packet events (see section 12.2.1), although it will trigger EOP (the end-of-preamble event). Note that formally stop can also be called before the end of preamble. This makes no sense from the viewpoint of the model, but it does not cause an error. Instead, stop in such a case is interpreted as abort.

A packet aborted past the preamble will never trigger an end-of-packet event (and no RFC_eot assessment will be done for it), although it may trigger beginning-of-packet events (and RFC_bot will be called for such a packet in stage T).

12.2 Packet perception and reception

The receiver part of a transceiver can be switched off and on. A switched off receiver is incapable of sensing any activities, and, in particular, it will not trigger any events related to packet reception. This feature is essential from the viewpoint of modeling the behavior of real-life transceivers, whose receivers may often remain in the off state for some periods, e.g., when transiting from transmit to receive mode. For example, if the receiver is switched on after the arrival of a packet preamble, but before the arrival of the first bit of the actual packet, it may still be able to recognize a sufficiently long fragment of the preamble to let it receive the packet. The preamble interference histogram passed to RFC_bot (sections 12.1.4 and 12.1.5) accounts only for the portion of the preamble that was in fact perceived by the transceiver while it was switched on. Thus, the protocol program is able to realistically model the deaf periods of the transceiver by appropriately switching its receiver on and off. This can be done with the following methods of Transceiver:

```
void rcvOn ();
void rcvOff ();
```

whose meaning is obvious. Note that if the receiver is switched on after the arrival of the first bit of the proper packet, the packet will not be received (its end-of-packet event will not even be assessed by RFC_eot). By default, e.g., immediately after the transceiver's creation, the receiver is in the "on" state.

Note that a similar handling of the transmitter would be redundant. If the transmitter is to remain off for some period of time, the protocol program should simply refrain from transmitting packets during that period. A sane program would not try to transmit while the transmitter is switched off, so maintaining this status internally would make little sense.

The following two transceiver methods:

```
Boolean busy ();
```

```
Boolean idle ();
```

tell the presence or absence of a (any) signal on the transceiver (the methods are complementary). If the receiver is off, its status is always "idle." Otherwise, it is determined by `RFC_act` (section 6.4.2), one of the user-exchangeable assessment methods (section 12.1.4), whose sole responsibility is to tell whether the channel is sensed busy or idle. In addition to determining the values returned by the above two methods, `RFC_act` is also used in timing two transceiver events: `SILENCE` and `ACTIVITY` (see section 12.2.1).

`RFC_act` takes two arguments. The first one gives the combined level of all signals currently perceived on the transceiver, as determined by `RFC_add` with the second argument equal `NONE` (section 12.1.4). The second argument of `RFC_act` is the reception gain of the transceiver (attribute `RPower` combined with its `Tag`). Intentionally, the method implements a threshold on the perceived signal level based on the current setting of the receiver gain.

12.2.1 Wait requests

The first argument of a wait request addressed to a transceiver is a symbolic constant that identifies the event type. Below we list those constants and explain the conditions for the occurrence of the respective events.

`SILENCE`
: The event occurs at the beginning of the nearest silence period perceived by the transceiver. This will happen as soon as 1) the receiver is switched off, or 2) the total signal level perceived by the transceiver makes `idle` return `YES` (or `busy` return `NO`). If the transceiver is `idle` when the request is issued, the event occurs within the current *ITU*.

`ACTIVITY`
: The event occurs at the nearest moment when 1) the receiver is on, and 2) the total signal level perceived by the transceiver makes `busy` return `YES` (or `idle` return `NO`). If the transceiver is `busy` when the request is issued, the event occurs within the current *ITU*.

`BOP`
: The event occurs at the nearest moment when the beginning of a preamble is heard on the transceiver. If the beginning of a preamble occurs exactly at the moment (within the same *ITU*) when the request is issued, the event occurs within the same *ITU*.

  Note that this event is not assessed (section 12.1.4) and is not a realistic event to be perceived by any physical hardware. For example, the beginning of any

preamble, no matter how much drowned in other activities will trigger the event. Its role is to enable exotic assessment schemes and other tricks that the user may want to play with the simulator.

If the receiver is switched on after the first bit of the preamble has passed through the transceiver, BOP will no longer be triggered: the event only occurs at the "true" beginning of a preamble.

EOP

The event occurs at the nearest moment when the end of a preamble is heard on the transceiver, i.e., immediately after the last bit of a preamble. If the end of a preamble occurs exactly at the moment (within the same *ITU*) when the request is issued, the event occurs within the same *ITU*.

Similar to BOP, this event is not assessed, and its role is to enable tricks rather than provide for modeling realistic phenomena. It will be triggered by the end of an aborted preamble, even if it is not followed by a packet (section 12.1.8). If the preamble is followed by a packet, then EOP will coincide with BOT (see below), except that the latter event is assessed.

BOT

(*Beginning of Transmission*). The event occurs as soon as the transceiver perceives the beginning of packet following a preamble (stage *T*), and RFC_bot returns YES for that packet (section 12.1.4). The interpretation of this event is that the preamble has been recognized by the transceiver, which can now begin to try to receive a packet. If the event condition is present at the moment when the wait request is issued, the event is triggered within the same *ITU*.

EOT

(*End of Transmission*). The event occurs as soon as the following three conditions are met simultaneously: 1) the transceiver perceives the end of packet (stage *E*), 2) RFC_bot returned YES for that packet in stage *T*, 3) RFC_eot returns YES for the packet in the current stage (section 12.1.4). The interpretation of this event is that a packet has been successfully received. If the event condition is present at the moment when the wait request is issued, the event is triggered within the same *ITU*.

BMP

(*Beginning of My Packet*). The event occurs under the same conditions as BOT with one added necessary condition: the packet must be addressed to the station running the process that has issued the wait request. Formally, it means that isMy (section 11.2.3) returns YES for the packet (this also covers

broadcast scenarios in which the current station is one of possibly multiple recipients).

`EMP`
: (*End of My Packet*). The event occurs under in the same conditions as `EOT` with one added necessary condition: the packet must be addressed to the station running the process that has issued the wait request. Formally, it means that `isMy` (section 11.2.3) returns `YES` for the packet (this also covers broadcast scenarios in which the current station is one of possibly multiple recipients). This is the most natural way of receiving packets at their proper destinations.

`ANYEVENT`
: This event occurs whenever the transceiver perceives anything of potential merit (any change in the configuration of perceived activities), e.g., when any of the preceding events would be triggered, if it were awaited. Beginnings and ends of all packets (stages T and E), not necessarily positively assessed by `RFC_bot`/`RFC_eot`, also trigger `ANYEVENT`. If any activity boundary or stage occurs at the moment when the wait request for `ANYEVENT` is issued, the event is triggered within the current *ITU*.

The last four event types deal with signal thresholds in two kinds of scenarios. First, the protocol program may want to perceive changes in the received signal level, e.g., to implement various "listen before talk" collision avoidance schemes. It is possible to define a specific signal threshold and then learn when the received signal level gets above or below the threshold. This is accomplished with the following transceiver method:

```
double setSigThreshold (double sl);
```

which sets the threshold level for the transceiver and returns the previous setting. By default, i.e., immediately after a transceiver is created, its threshold setting is 0.0. The two threshold events are:

`SIGLOW`
: This event occurs whenever the received signal level perceived by the transceiver (as returned by `RFC_add` with the second argument equal `NONE`—section 12.1.4) becomes less than or equal to the current threshold setting. If the condition holds when the wait request is issued, the event occurs within the current *ITU*.

`SIGHIGH`
: This event occurs whenever the received signal level perceived by the

transceiver becomes greater than the current threshold setting. If the condition holds when the wait request is issued, the event occurs within the current *ITU*.

It is also possible to directly monitor the level of interference suffered by a selected activity, typically a packet being in some partial stage of reception. This transceiver method:

```
Boolean follow (Packet *p = NULL);
```

allows the program to declare the activity to be monitored (followed). If a non-NULL argument is present, it should point to a packet being carried by one of the activities currently perceived by the transceiver. Such a packet pointer can be obtained, e.g., via `ThePacket` by receiving a packet-related event (`BOT` or `BMP`), or through one of the inquiries addressed to the transceiver, as explained in section 12.2.5. If the argument is absent (or NULL), it implies the activity last examined by a transceiver inquiry (section 12.2.5). The method returns `OK`, if the argument (or the last transceiver inquiry) identifies an activity currently perceived by the transceiver (in which case the activity whose interference level is to be monitored has been successfully specified), or `ERROR` otherwise.

Given the pointer to one of the packets currently being perceived by a transceiver, this `Transceiver` method:

```
Boolean isFollowed (Packet *p);
```

returns `YES`, if and only if the packet pointed to by `p` is the one being "followed," i.e., it was previously marked with `follow`.

The threshold interference level is set by `setSigThreshold`—as for `SIGLOW` and `SIGHIGH`. The following two events are available:

`INTLOW`
: This event occurs whenever the interference level of the "followed" activity becomes less than or equal to the threshold. If the condition holds when the wait request is issued, the event occurs within the current *ITU*.

`INTHIGH`
: This event occurs whenever the interference level of the "followed" activity becomes greater than the threshold. If the condition holds when the wait request is issued, the event occurs within the current *ITU*.

Note that the transceiver must be "on" for all packet perception/reception events to be ever triggered. Only `SILENCE` and `SIGLOW` are triggered when the receiver is (or goes) off. The perceived signal level of a switched-off receiver is zero. Neither `INTLOW` nor `INTHIGH`

is triggered when the receiver is switched off. The "followed" activity must be perceptible (at least in principle) to trigger any of these two events.

When a process is awakened by any of the above events (except for `ANYEVENT`—see section 12.2.1), the environment variable `TheTransceiver` (`Info02`) of type `Transceiver*` contains the pointer to the port on which the waking event has occurred. If the waking event is related to a specific (proper) packet (this concerns `BOT`, `BMP`, `EOT`, `EMP` and also `ACTIVITY`—see below), the environment variable `ThePacket` (`Info01`) of type `Packet*` returns the packet pointer. This object pointed to by `ThePacket` is the copy of the packet buffer created by `startTransfer` or `transmit` (section 12.1.8) when the packet transmission was started.

Before triggering the `ACTIVITY` event, SIDE determines whether the transceiver is perceiving a "receivable" packet, and if this is the case, its pointer is returned via `ThePacket`. Otherwise, `ThePacket` contains `NULL`. A packet is deemed receivable if 1) it was positively assessed by `RFC_bot`, 2) its portion received so far is positively assessed by `RFC_eot`. Thus, if the protocol program wants to take advantage of this feature, `RFC_eot` must be prepared to meaningfully assess incomplete packets.

Note: The object pointed to by `ThePacket` is deallocated when the activity carrying the packet is removed from the radio channel. This usually happens one *ITU* after the last bit of the packet disappears from the most distant transceiver of the sender's neighborhood.

The philosophy of the event handling mechanism of SIDE enforces the view that one event conveys information about one thing. While generally this makes perfect sense, there may be occasional situations when the protocol program wants to discern several conditions that may be present within the same *ITU*. Similar to the port case (section 11.2), `ANYEVENT` directly returns information about a single event, and if two (or more) of them occur within the same *ITU*, the one actually presented is chosen at random. The awakened process can learn about all of them by calling `events` (section 12.2.5).

When a process awaiting `ANYEVENT` is awakened, the environment variable `TheEvent` (`Info02`) of type `int` contains a value identifying the event type. The possible values returned in `TheEvent` are represented by the following constants (coinciding with the identifiers of respective events):

`BOP`	beginning of preamble
`EOP`	end of preamble
`BOT`	beginning of a packet (stage T)
`EOT`	end of a packet (stage E)

If the event type is `BOT` or `EOT`, `ThePacket` points to the related packet object.

Similar to the port variant of the event, it is possible to declare a special mode of ANYEVENT, whereby the event is never pending (see section 11.2). This is accomplished by invoking the setAevMode method of Transceiver, whose properties are identical to its Port counterpart.

12.2.2 Event priorities

Section 11.2.2 (referring to ports) also applies to transceivers. If the explicit event ordering mechanism is switched off, i.e., the third argument of wait is unavailable (section 7.4), SIDE imposes implicit ordering on multiple events awaited by the same process that may refer to the same actual condition. Thus, for a transceiver, BMP is triggered before BOT, and EMP is triggered before EOT, which in turn precedes SILENCE. Technically, ACTIVITY, if occurring together with BMP or BOT, has the lowest priority; however, for a transceiver, ACTIVITY is unlikely to occur exactly on the packet boundary (stage T) as the packet is preceded by a nonzero-length preamble, which (in all sane circumstances) should trigger ACTIVITY earlier. Even if the receiver is switched on exactly on the packet boundary, in which case ACTIVITY will formally coincide with BOT, the total lack of preamble will fail the assessment of this event without ever consulting RFC_bot.

Events BOP and EOP are not covered by the implicit priority scheme.

12.2.3 Receiving packets

Refer to section 11.2.3 for the explanation of how packets should be extracted from transceivers and formally received. Everything said in that section applies also to transceivers: just mentally replace all occurrences of "port" with "transceiver." In particular, the receive methods of the Client and traffic patterns exist in the following variants:

```
void receive (Packet *pk, Transceiver *tr);
void receive (Packet *pk, RFChannel *rfc = NULL);
void receive (Packet &pk, Transceiver &pr);
void receive (Packet &pk, RFChannel &rfc);
```

which perform the required bureaucracy of packet reception, including updating various performance-related counters associated with the radio channel.

A sample code of a process responsible for receiving packets from a transceiver may look like this:

```
perform {
    state WaitForPacket:
        RcvXcv->wait (EMP, NewPacket);
```

```
        state NewPacket:
            Client->receive (ThePacket, TheTransceiver);
            skipto WaitForPacket;
    };
```

where `RcvXcv` is a transceiver pointer.

12.2.4 Hooks for handling bit errors

The simulation model of a communication channel can be viewed as consisting of two main components: the part modeling signal attenuation, and the part transforming the received signal level (possibly affected by interference and background noise) into bit errors. The results from the first component are pipelined into the second one, whose ultimate purpose is to tell which packets have been correctly received.

Example

Suppose that you know how to calculate the bit error rate based on the received signal strength (`sl`), receiver gain (`rs`), and the interference level (`il`). This part of your channel model can be captured by a function, say

```
double ber (double sl, double rs, double il);
```

which returns the probability of a single-bit error.[38] One way to arrive at such a function is to calculate the signal-to-noise ratio, e.g.,

$$SNR = \frac{sl \times rs}{il \times rs + bg}$$

where bg is a fixed (mean) level of background noise. Then, SNR can be transformed into the probability that a single bit is received in error, e.g., by using an interpolated table obtained experimentally, or resorting to some closed formula from a formal model. With a function like `ber` in place, `RFC_eot` may look like this:

```
Boolean RFC_eot (RATE r, const SLEntry *sl, const SLEntry *sn,
                                                const IHist *h) {
    int i;
    Long nb;
    double intf;
    for (i = 0; i < h->NEntries; i++) {
```

[38] For simplicity, we ignore the impact of the transmission rate: in many models the rate is fixed and can thus be hardwired into `ber`.

```
            h->entry (i, r, nb, intf);
            if (rnd (SEED_toss) >
              pow (1.0 - ber (sl->Level, sn->Level, intf), nb))
                return NO;
        }
        return YES;
    }
```

For simplicity, we assume homogeneous signals (tags are not used). The method goes through all chunks of the packet that have experienced a fixed level of interference. For each chunk of length `nb` bits that has suffered interference `intf`, it assumes that the chunk contains no error with probability $(1 - E)^{\texttt{nb}}$, where E is the error rate at the given interference level, as prescribed by `ber`. As soon as a single bit error is discovered, the method stops and returns `NO`. Having examined all chunks and found no error, the method returns `YES`, which yields a positive assessment of the entire packet.

The above example is merely an illustration. An actual scheme for transforming the reception parameters of a packet into its stochastic assessment decision need not be based on the notion of bit error rate or signal-to-noise ratio. However, this view is a rather standard component of all serious channel models; thus, SIDE goes one step further towards accommodating it as an integral (albeit optional) part of the overall assessment framework. Two special assessment methods, `RFC_erb` and `RFC_erd`, are provided to capture the complete description of bit-error distribution as a function of the reception parameters of an incoming packet. By defining those methods, the user

1. localizes the bit-error model in one place, independent of other model components,
2. enables certain tools, which make it easier to carry out actual packet assessment, and, more importantly, trigger events resulting from possibly complicated configurations of bit errors.

The two methods have identical headers:

```
Long RFC_erb (RATE r, const SLEntry *sl, const SLEntry *rs,
                                                double il, Long nb);
Long RFC_erd (RATE r, const SLEntry *sl, const SLEntry *rs,
                                                double il, Long nb);
```

`RFC_erb` is expected to return a randomized number of error bits within a sequence of `nb` bits received at rate `r`, signal parameters `sl` (level and tag), receiver parameters (gain and tag) `rs`, and interference level `il`. For example, supposing that the function `ber` from the previous example transforms the reception parameters into the probability of a bit error, and that bit errors are independent, here is one possible variant of the method:

```
Long RFC_erb (RATE r, const SLEntry *sl, const SLEntry *rs,
                                                double il, Long nb) {
    return lRndBinomial (ber (sl->Level, rs->Level, il), nb);
};
```

Once `RFC_erb` is defined, the following methods of `RFChannel` become available:

```
Long errors (RATE r, const SLEntry *sl, const SLEntry *rs,
                        const IHist *h, Long sb = -1, Long nb = -1);
Boolean error (RATE r, const SLEntry *sl, const SLEntry *rs,
                        const IHist *h, Long sb = -1, Long nb = -1);
```

Suppose first that the last two arguments are not specified. Then, the first method calculates (by calling `RFC_erb`) a randomized number of error bits in the histogram `h`. The second method returns `YES`, if the histogram contains one or more bit errors, and `NO` otherwise. Formally, it is equivalent to `errors (r, sl, rs, h) != 0`, but executes slightly faster.

The last two arguments allow you to narrow down the error lookup to a fragment of the histogram. If both `sb` and `nb` are nonnegative, then `sb` specifies the starting bit position (remember that bits in a histogram are numbered from 0), and `nb` gives the number of bits to be looked at. If that number falls behind the activity represented by the histogram, or is negative (or simply skipped), the lookup extends to the end of the histogram. If `sb` is negative and `nb` is not, then `nb` refers to the trailing number of bits in the histogram, i.e., errors will be looked up in the last `nb` bits of the activity.

With these tools in place, `RFC_eot` can be simplified as follows:

```
Boolean RFC_eot (RATE r, const SLEntry *sl, const SLEntry *sn,
                                                    const IHist *h) {
    return !error (r, sl, sn, h);
}
```

The two `RFChannel` methods are also available as transceiver methods, in the following variants:

```
Long errors (SLEntry *sl, const IHist *h);
Boolean error (SLEntry *sl, const IHist *h);
```

The missing arguments, i.e., the reception rate, receiver gain, and receiver tag, are taken from the current settings of this transceiver. Note that the packet's reception rate is assumed to be equal to the transceiver's current transmission rate, which need not be always correct.

Two more variants of error/errors are mentioned in section 12.2.5. Note that the availability of these methods (six of them altogether) is the only net advantage of RFC_erb. Thus, you need not define RFC_erb, if your program does not care about the error/errors methods.

The role of RFC_erd is to generate randomized time intervals (in bits) preceding events related to bit errors. To illustrate its possible applications, consider a receiver that extracts symbols (i.e., sequences of bits) from the channel and stops (aborts the reception) upon the first occurrence of an illegal symbol. To model the behavior of such a receiver with a satisfying fidelity, you may want to await simultaneously several events, e.g.,

1. EOT indicating that the packet has been successfully received,

2. SIGLEVEL indicating that the packet signal has been lost,

3. an event representing the occurrence of an illegal symbol in the received sequence

Generally, events of the third type require a custom description capturing the various idiosyncrasies of the encoding scheme. While their timing is strongly coupled to the bit error rate, the exact nature of that coupling must be specified by the user as part of the error model. This is the role of RFC_erd, whose first four arguments describe the reception parameters (exactly as for RFC_erb, while the last argument identifies a custom function (one out of possibly several options) indicating what exactly the function is supposed to determine. This way, RFC_erb provides a single custom generator of time intervals for possibly multiple types of events triggered by various interesting configurations of bit errors.

The second part of this mechanism is a dedicated event named BERROR, which is triggered when the delay generated by RFC_erd expires. The complete incantation includes a call to the following transceiver method:

```
Long setErrorRun (Long em);
```

which declares the so-called *error run*, i.e., the effective value of the last argument to RFC_erd selecting a specific action of the method. Having recognized the beginning of a potentially receivable packet (after BOT), the program should execute follow for it (section 12.2.1). Then, BERROR events will be timed by RFC_erd applied to the reception parameters of the *followed* packet.

As most "set" methods, setErrorRun returns the previous setting of the error run option. Also, an RFChannel variant is available which returns no value and selects the same value of error run for all transceivers interfaced to the channel. There is a matching getErrorRun method of Transceiver, which returns the current setting of the option.

Example

The following method:

```
Long RFC_erd (RATE r, const SLEntry *sl, const SLEntry *rs,
                                                double il, Long nb) {
    return (Long)
       dRndPoisson (1.0/pow (ber (sl->Level, rs->Level, il), nb));
};
```

returns a randomized number of bits preceding the first occurrence of `nb` consecutive error bits. In this case, the last argument gives the length of an error run. Now consider this code fragment:

```
...
  state RCV_WAIT:
    Tcv->wait (BOT, RCV_START);
  state RCV_START:
    Tcv->follow (ThePacket);
    Tcv->setErrorRun (3);
    Tcv->setSigLevel (dBToLin (-40.0));
    Tcv->wait (EOT, RCV_GOTIT);
    Tcv->wait (BERROR, RCV_IGNORE);
    Tcv->wait (SIGLOW, RCV_GONE);
...
```

Having recognized the beginning of a packet, the process awaits one of three events: its successful reception, three consecutive error bits, or a low signal indicating that the packet has disappeared. The second event may be synonymous with an incorrect symbol, which will abort the reception before the last bit of the packet arrives at the transceiver.

The presence of `RFC_erd` among the assessment methods of a radio channel enables `BERROR` as one more event type that can be awaited on a transceiver interfaced to the channel. If the program does not care about this event, `RCF_erd` is not needed.

12.2.5 Transceiver inquiries

In contrast to ports, transceivers offer no inquiries about the past. Two inquiries about the present, methods `busy` and `idle`, were mentioned in section 12.2.

It is possible to examine all activities (packets) being perceived by a transceiver at the current moment. These methods of `Transceiver`:

```
int activities (int &p);
int activities ();
```

return the total number of activities (packets at different stages) currently perceived by the transceiver. The argument in the first variant returns the number of activities being past stage T, i.e., within their proper packets following the preamble. If the population of such activities is nonempty, `ThePacket` returns a pointer to one of them (selected at random).

When a process is awakened by `ANYEVENT` (section 11.2), only one of the possibly multiple conditions being present at the same ITU is presented to the process in `TheEvent` (section 12.2.1). It is possible to learn how many events of a given type occur at the same time by calling this transceiver method:

```
int events (int etype);
```

where the argument identifies the event type in the following way:

`BOP`	beginning of a preamble
`EOP`	end of a preamble
`BOT`	beginning of a packet (stage T)
`EOT`	end of a packet (stage E)

The method returns the number of events of the type indicated by the argument.

The following transceiver method can be used to scan through all activities currently perceived by the transceiver:

```
int anotherActivity ();
```

The exact behavior of `anotherActivity` depends on the circumstances of its call. If the method is invoked after one of the events `BOP`, `EOP`, `BOT`, `EOT`, `BMP`, `EMP`, then its subsequent calls will scan all the activities that have triggered the event simultaneously. The same action is performed after a call to `events` (whose argument specifies an event type). For illustration consider the following code fragment:

```
state WaitPacket:
    Tcv->wait (EOT, LookAtThem);
state LookAtThem:
    while (Tcv->anotherActivity () != NONE)
        handlePacket (ThePacket);
    skipto WaitPacket;
```

which traverses all packets found in stage *E*. Note `skipto` used to transit back to state `WaitPacket`—to remove the `EOT` event from the transceiver (section 8.1).

The method returns `NONE` when it runs out of activities to present. Otherwise, its value is `PREAMBLE` or `TRANSFER`, depending on whether the activity is still in the preamble stage or within the proper packet. In the above example, all activities traversed by `anotherActivity` are bound to be within the proper packet (stage *E*), so there is no need to worry about this detail.

If `anotherActivity` is called after `BOP` or `EOP`, then its subsequent invocations will return `PREAMBLE` as many times as many activities are found in the respective stage, but it will not set `ThePacket` (its value will be `NULL`).

If the method is called after `ANYEVENT`, `ACTIVITY`, or after a call to `activities`, then it will scan through all activities perceived by the transceiver at the current *ITU*, regardless of their stage. Whenever it returns `TRANSFER` (as opposed to `PREAMBLE`), `ThePacket` will be set to point to the respective packet.

Even if `anotherActivity` returns `PREAMBLE` (and `ThePacket` is `NULL`), some further inquiries about the activity just scanned by the method are possible. In particular, the following method:

```
Packet *thisPacket ();
```

returns the pointer to the packet carried by the last activity examined by `anotherActivity`, even if it is before stage *T* (i.e., `anotherActivity` has returned `PREAMBLE`). This may look like cheating, but may also be useful in tricky situations, e.g., when the packet structure carries some complex information pertaining to the properties of the preamble.

For scanning packets, i.e., activities past stage *T*, this method of `Transceiver` can be used as a shortcut, instead of `anotherActivity`:

```
Packet *anotherPacket ();
```

It behaves in a manner similar to `anotherActivity`, except that all activities before stage *T* are ignored. If `anotherPacket` is called after one of the events `BOT`, `EOT`, `BMP`, `EMP`, then its subsequent calls will scan through all the packets that have triggered the event simultaneously. The same action is performed after a call to `events`, whose argument specifies the respective event type. In all other circumstances, `anotherPacket` will scan through all packets (activities past stage *T*) currently perceived by the transceiver. For each packet, the method sets `ThePacket` to the packet pointer and returns the same pointer as its value. If the pointer is `NULL`, it means that the list of packets has been exhausted.

Several types of inquiries deal with the signal level, including various elements of the interference affecting packet reception. In particular, this method of `Transceiver`:

```
double sigLevel ();
```

returns the total signal level perceived by the transceiver, as calculated by RFC_add with the second argument equal NONE (section 12.1.4).

Following a call to anotherActivity or anotherPacket, this method:

```
double sigLevel (int which);
```

will return the signal level related to the last-scanned activity. The value returned by sigLevel depends on the argument (which) in the following way:

which == SIGL_OWN
: The perceived signal level of this activity.

which == SIGL_IFM
: The maximum interference so far. This maximum refers to the current stage, i.e., if the packet is past stage T, the interference suffered by the preamble is no longer relevant.

which == SIGL_IFA
: The average interference so far. This average refers to the current stage, i.e., if the packet is past stage T the interference suffered by the preamble is no longer relevant.

which == SIGL_IFC
: The current interference level.

One more variant of the method,

```
double sigLevel (const Packet *pkt, int which);
```

applies to the activity carrying the indicated packet, if the activity is currently perceived by the transceiver. If the method cannot locate the "current" activity (this applies to the last two variants of sigLevel), it returns a negative value (-1.0). Note that a packet triggering EOT is eligible for this kind of inquiry: its activity is still available.

Similar circumstances apply to this method of Transceiver:

```
IHist iHist (const Packet *pkt);
```

which returns a pointer to the packet's interference histogram (section 12.1.5). This histogram can be used in tricky scenarios (see section 12.2.4 for an illustration), e.g., to

affect the contents of a received packet in a manner depending on the interference that its particular fragments have suffered.

IF `RFC_erb` is present among the assessment methods of the underlying radio channel (section 12.2.4), these two transceiver methods become available:

```
Long errors (Packet *p = NULL);
Boolean error (Packet *p = NULL);
```

If the argument is not NULL, it must point to a packet currently being perceived by the transceiver. The first method returns the randomized number of bit errors found in the packet so far, the second returns `YES` if the packet contains one or more errors, as explained in section 12.2.4. If the argument is NULL, the inquiry applies to the activity last scanned by `anotherActivity` or `anotherPacket`. If `p` does not point to a packet currently perceived by the transceiver, or `p` is NULL and no meaningful activity is available for the inquiry, `errors` returns `ERROR` (a negative value) and `error` returns `NO`.

The last inquiry of interest is implemented by this method of `Transceiver`:

```
Boolean dead (const Packet *pkt = NULL);
```

If the argument is not NULL, the method returns `YES` if the packet has been found not receivable. This will happen in any of the following circumstances:

1. The packet is past the preamble (stage T) and the preamble assessment by `RFC_bot` (section 12.1.4) was negative.
2. The packet has been aborted by the sender and this event has made it to the perceiving transceiver.
3. The packet is in stage E and the assessment by `RFC_eot` has been negative.
4. The packet is not perceived any more, i.e., its last bit has disappeared from the transceiver.
5. The packet is past stage T and before stage E, its preamble assessment (by `RFC_bot`) was positive, but its partial assessment by `RFC_eot` (invoked at the current stage of the packet) is negative.

In all other cases, `dead` returns `NO`.

Note that, if this feature is used, `RFC_eot` must be prepared to sensibly assess packets before stage E. This is similar to the assessment carried out for the `ACTIVITY` event (section 12.2.1).

If `dead` is called without an argument (or with the argument equal `NULL`), then it applies to the activity last scanned by `anotherActivity` or `anotherPacket`. Any activity found

before stage T is never `dead` unless it is an aborted preamble whose last bit has just arrived at the transceiver.

Note: If the receiver is off (section 12.2), no activities can be perceived on it, and no signals can be measured. Specifically `activities` and `events` return zero, `anotherActivity` immediately returns `NONE`, `anotherPacket` immediately returns `NULL`, `sigLevel` returns 0.0, and `dead` returns `YES` regardless of the argument.

The following global function does not formally belong to transceiver inquiries, and is mentioned here for the record:

```
double dist (double x0, double y0, double x1, double y1);
```

It calculates the distance between the two points (`x0`,`y0`) and (`x1`,`y1`). If the simulator has been created with the `-3` option of `mks` (selecting the 3-dimensional spatial model—see section 19.2), the method's header looks like this:

```
double dist (double x0, double y0, double z0,
                        double x1, double y1, double z1);
```

i.e., each point is described by three coordinates.

12.3 Cleaning after packets

The same feature for postprocessing packets at the end of their life time in the radio channel is available for radio channels as for links (section 11.5). Even though, in contrast to links, type `RFChannel` is extensible by the user (and the cleaning function could be accommodated as a virtual method), the idea of a plug-in global function has been retained—for disputable compatibility with links.

13 The `Mailbox` *AI*

One way to synchronize processes is to take advantage of the fact that each process is an autonomous *AI* capable of triggering events that can be perceived by other processes (section 7.7). A more systematic tool for inter-process communication is the `Mailbox` *AI*, which can also serve as an interface of the SIDE program to the outside world (in the real or visualization mode).

13.1 General concepts

The `Mailbox` *AI* provides a frame for defining mailbox types and creating objects of these types. This tool has evolved quite a bit from its original and rather simple version

(from early SMURPH) reaching the stage when it rather desperately call for some drastic measure, like replacing it with two or three separate tools (with different names). While there seems to be a unifying theme for all the different mailbox variants described in this section, one can see three distinct types of functionality, which, in combination with some other aspects, require quite a bit of taxonomy. This is what we have to start with to get the terminology straight (or at least as straight as possible).

A mailbox is primarily a repository for "messages"[39] that can be sent and retrieved by processes. Viewed from the highest level of its role in the program a mailbox can be *internal* or *external* (also called *bound*). An internal mailbox is used as an "internal" synchronization/communication vehicle for SIDE processes, while a bound mailbox is attached to an external device or a socket, thus providing the control program with a reactive interface to the outside world. Bound mailboxes are only available in the real or visualization mode, while internal mailboxes can be used in both modes.

An internal mailbox, in turn, can be a *barrier* or a *fifo*. Both kinds implement different variants of signal passing, with the option of queuing pending signals (possibly accompanied with some extra information) in the second case. A barrier mailbox is a place where any process can block awaiting the occurrence of an identifiable event (or signal), which can be delivered by another process. An arbitrary number of processes can wait on a barrier mailbox at the same time, for the same or different event numbers. When an event with a given number is delivered, all the processes awaiting this particular event number are immediately awakened.

A fifo mailbox acts as a queued repository for items. One attribute of such a mailbox is its capacity, i.e., the maximum number of items that can be stored awaiting acceptance. This capacity can be zero, which gives the mailbox a special (degenerate but useful) flavor.

The taxonomy is further confounded by the fact that a fifo mailbox can be either *counting* or *typed*. In the first case, the items deposited in the mailbox are illusory: all that matters is their *count*, and the entire fifo is fully represented by a single counter. In the second case, the items are actual. Their type must be simple (meaning convertible to `void*`), but it can be a pointer representing possibly complex structures.

13.2 Fifo mailboxes

A fifo mailbox has a capacity, which is usually determined at the moment of its creation, but can also be redefined dynamically, A mailbox with capacity $N >= 0$ can accommodate up to N pending items. Such items are queued, if not immediately awaited, and can be extracted later. If the capacity is huge (e.g., `MAX_long`—section 3.5), the mailbox becomes an essentially unlimited FIFO-type storage.

A fifo mailbox whose capacity is zero (which is the default) is unable to store even a single

[39] Not to be confused with the client messages described in section 10.2.

item. This makes sense because a process can still receive a deposited item if it happens to be already waiting on the mailbox when the item arrives.

The type of elements stored in a fifo mailbox can be declared with the mailbox type extension. Fifo mailboxes belonging to the standard type `Mailbox` are *counting*, i.e., the items are structure-less tokens that carry no information other than their presence. A typed mailbox may specify a simple type for the actual items, which, in particular, can be a pointer type.

At any moment, a fifo mailbox (typed or counting) can be either empty or nonempty. A non-empty counting mailbox holds one or more pending tokens that await acceptance. As these tokens carry no information, they are all represented by a single counter. Whenever a token is put into a counting mailbox, the mailbox element counter is incremented by one. Whenever a token is retrieved from a nonempty counting mailbox, the counter is decremented by one. A non-empty typed mailbox is a FIFO queue of some actual objects.

13.2.1 Declaring and creating fifo mailboxes

The standard type `Mailbox` is usable directly and it describes (by default) a capacity 0, counting (i.e., untyped), fifo mailbox. Thus, with the following sequence:

```
Mailbox *mb;
  ...
mb = create Mailbox;
```

you will create such a mailbox and store its pointer in `mb`.

The mailbox capacity can be specified as the setup argument, e.g.,

```
mb = create Mailbox (1);
```

creates a capacity-1 mailbox.

A user-provided extension of type `Mailbox` is required to define a typed mailbox capable of storing tangible objects of some definite type. The type extension must specify the type of elements to be stored in the mailbox queue. The declaration has the following format:

```
mailbox mtype : itypename ( etype ) {
    ...
    attributes and methods
    ...
};
```

where `mtype` is the new mailbox type. The optional argument in parentheses, if present, must identify a simple C++ type. This argument specifies the type of elements to be stored

in the mailbox. Note that although the element type must be simple (it is internally cast to `void*`), it can be a pointer type. If the type argument is absent, the new mailbox type describes a counting mailbox. Such a declaration is pointless except, possibly, for code clarity (having different names for essentially the same basic mailbox type).

Examples

The following declaration describes a fifo mailbox capable of storing integer values:

```
mailbox IntMType (int);
```

Note the empty body of the declaration. As the type argument is present, the above declaration will not be mistaken for an announcement (section 5.7).

You can control (intercept) the basic operations on a fifo mailbox (to do some extra processing on the side) by declaring within the mailbox class the following two virtual methods:

```
void inItem ( etype par );
void outItem ( etype par );
```

Each of the two methods has one argument whose type should be the same as the mailbox element type. The first method (`inItem`) will be called whenever a new item is added into the mailbox. The argument will then contain the value of the item. Similarly, `outItem` will be called whenever an item (indicated by the argument) is removed from the mailbox.

The two methods are independent which means that it is legal to declare only one of them. For example, if only `inItem` is declared, no extra action will be performed upon element removal.

Note that when a fifo mailbox has reached its capacity (is full), an attempt to add a new item to it will fail (section 13.2.3). In such a case, `inItem` **will not** be called. Similarly, `outItem` will not be executed upon a failed attempt to retrieve an item from an empty mailbox. Formally, the two methods can be declared for a counting mailbox (when no item type is specified in the mailbox declaration), but they will never be called.

A mailbox can be owned either by a station or by a process. The most relevant difference is that a mailbox owned by a station cannot be destroyed, while that owned by a process can. Another difference may come into play when a mailbox is exposed (section 17). This applies to all kinds of mailboxes, not only fifos. Those mailboxes that have been created during the initialization phase of the protocol program (i.e., while the `Root` process was in its first state—section 7.9) have ownership properties similar to those of ports (section 6.3.1), transceivers (section 6.5.1) and packet buffers (section 10.3), i.e., they belong to the stations at which they have been created. The `Id` attribute of such a

mailbox combines the serial number of the station owning it and the creation number of the mailbox within the context of its station. The naming rules for a station mailbox are practically identical to those for ports (section 6.3.1). The standard name of a station mailbox (section 5.2) has the form

mtypename mid `at` *stname*

where *mtypename* is the type name of the mailbox, *mid* is the station-relative mailbox number, and *stname* is the standard name of the station owning the mailbox. The base standard name of a mailbox has the same format as the standard name, except that *mtypename* is always `Mailbox`.

The following `Station` method:

```
Mailbox *idToMailbox (Long id);
```

converts the numerical station-relative mailbox identifier into a pointer to the mailbox object. An alternative way to do the same thing is to call a global function declared exactly as the above method which assumes that the station in question is pointed to by `TheStation` (section 6.1.3).

A mailbox can also be declared statically within the station class, e.g.,

```
mailbox MyMType (ItemType) {
    ...
};
...
station MySType {
    ...
    MyMType Mbx;
    ...
};
```

The capacity of a mailbox, including one that has been declared statically (as in the above example) can be set after its creation by calling the `setLimit` method of `Mailbox`, e.g.,

```
station MyStation {
 ...
  Mailbox MyMType;
 ...
  void setup () {
   ...
    MyMType.setLimit (4);
    MyMType.setNName ("My Mailbox");
```

```
    ...
  };
 ...
};
```

When the protocol program is in the initialization phase (section 7.9), a dynamically created mailbox (with `create`) is assigned to the current station. Note that this station can be `System` (section 6.1.3). This kind of ownership makes sense, e.g., for a "global" mailbox interfacing processes belonging to different regular stations.

A mailbox created dynamically after the initialization phase is owned by the creating process and its `Id` attribute is a simple serial number incremented globally, e.g., as for a process (section 5.2). As we said before, a process mailbox is more dynamic than one owned by a station because it can be destroyed (by `delete`).

A mailbox type declaration can specify private `setup` methods. These methods are only accessible when the mailbox is created dynamically, with the `create` operation. As a private `setup` method subsumes the standard method that defines the mailbox capacity, it should define this capacity explicitly by invoking `setLimit` (see section 13.2.3).

13.2.2 Wait requests

An event identifier for a fifo mailbox is either a symbolic constant (representing a negative number) or a nonnegative number corresponding to item count. In particular, by calling

```
mb->wait (NONEMPTY, GrabIt);
```

the process declares that it wants to be awakened in state `GrabIt` at the nearest moment when the mailbox pointed to by `mb` becomes nonempty. If the mailbox happens to be nonempty at the moment when the request is issued, the event occurs immediately, i.e., within the current *ITU*.

Another event that can be awaited on a fifo mailbox is `NEWITEM` or `UPDATE`, which are two different names for the same event. A process that calls

```
mb->wait (NEWITEM, GetIt);
```

will be restarted (in state `GetIt`) at the nearest moment when an item is put into the mailbox (section 13.2.3).

One should note the difference between these two events. First of all, `NEWITEM` (`UPDATE`) is only triggered when an element is being stored in the mailbox. Unlike `NONEMPTY`, it does not occur immediately if the mailbox already contains some item(s). Moreover, no `NONEMPTY` event ever occurs on a capacity 0 mailbox. By definition, a capacity 0 mailbox is never `NONEMPTY`. On the other hand, a process can sensibly await the `NEWITEM` event on

a capacity 0 mailbox. The event will be triggered at the nearest moment when some other process executes the `put` operation (section 13.2.3) on the mailbox.

Note: The only event that can be awaited on a capacity 0 mailbox is `NEWITEM` (`UPDATE`).

When a process is restarted by `NEWITEM` (on a nonzero capacity mailbox) or `NONEMPTY`, it is not absolutely guaranteed that the mailbox is in fact nonempty. This can happen when two or more processes have been waiting for `NONEMPTY` (or `NEWITEM`) on the same mailbox. When the event is triggered, they will be all restarted within the same *ITU*. Then one of the processes may empty the mailbox before the others are given a chance to look at it.

Here is a way to await a guaranteed delivery of an item. A process calling

```
mb->wait (RECEIVE, GotIt);
```

will be awakened in state `GotIt` as soon as the mailbox becomes nonempty and the process is able to actually retrieve something from it. Similar to `NONEMPTY`, the event occurs immediately, if the mailbox is nonempty when the wait request is issued. Before the process code method is called, the first element is removed from the mailbox and stored in the environment variable `TheItem` (see below).[40]

Suppose that two or more processes are waiting for `RECEIVE` on the same mailbox. When an item is put into the mailbox, only one of those processes will be restarted[41] and the item will be automatically removed from the mailbox. If some other processes are waiting for `NONEMPTY` and/or `NEWITEM` at the same time, all these processes will be restarted even though they may find the mailbox empty when they actually get to run.

If a `RECEIVE` event is triggered on a typed mailbox that defines the `outItem` method (section 13.2.1), the method is called with the removed item passed as the argument. This is because triggering the event results in the automatic removal of an item.

If the first argument of a mailbox wait request is a nonnegative integer number, it represents a *count* event, which is triggered when the mailbox contains **precisely** the specified number of items. In particular, if the specified count is 0 (symbolic constant `EMPTY`), the even will occur when the mailbox becomes empty. If the mailbox contains exactly the specified number of items when the wait request is issued, the event occurs within the current *ITU*. Note that if multiple processes access the mailbox at the same time, the actual number of items when the process is restarted may not match the argument of `wait`.

When a process is restarted due to a mailbox event, the environment variable `TheMailbox` (`Info02`) of type `Mailbox`, points to the mailbox on which the event has occurred. For a typed mailbox, and when it makes sense, the environment variable `TheItem` (`Info01`) of type `void*` contains the value of the item that has triggered the event. Note that for

[40] This does not happen for a counting mailbox, in which case `TheItem` returns NULL.

[41] Nondeterministically or according to the `order` attributes of the wait requests (section 7.4).

some events this value may not be up to date. For a counting mailbox, `TheItem` is always `NULL`.

Note: if a process is waiting on a mailbox and that mailbox becomes deleted,[42] the process will be awakened regardless of the event awaited by it. This is potentially dangerous. If the program admits such a situation at all, the process should make sure that the mailbox is still around when it wakes up. This can be easily determined by looking at `TheMailbox`, which will be `NULL` if the mailbox has been destroyed. This feature applies to all mailbox types and all events that can be awaited on them.

The interpretation of `TheItem` for `NONEMPTY`, `NEWITEM`, and a count event is sensible under the assumption that only one process at a time is allowed to await events on the mailbox and remove elements from it. The interpretation of `TheItem` for `RECEIVE` is always safe. When a process is restarted by this event occurring on a typed mailbox, `TheItem` contains the value of the received item. Note that the item has been removed from the mailbox by the time the process is run, so `TheItem` is the only way to get hold of it.

Yet another event type that can be awaited on a fifo mailbox is `GET`, which is triggered whenever anybody executes a successful `get` operation on the mailbox. This also applies to `erase` (see section 13.2.3). For symmetry, there exists a `PUT` event, which is yet another (in addition to `UPDATE`) alias for `NEWITEM`.

13.2.3 Operations on fifo mailboxes

The following `Mailbox` method is used to add a token to a counting mailbox:

```
int put ();
```

For a typed mailbox, the method accepts one argument whose type coincides with the item type specified with the mailbox type declaration (section 13.2.1).

The method returns one of the following three values:

`ACCEPTED`
: The new element triggers an event that is currently awaited on the mailbox, i.e., a process awaiting a mailbox event will be awakened within the current *ITU*. Note, however, that this event need not be the one that will actually restart the waiting process.

`QUEUED`
: The new element has been accepted and stored in the mailbox, but it does not trigger any events immediately.

[42] This can only happen for a mailbox owned by a process as opposed to one owned by a station.

REJECTED
: The mailbox is full (it has reached its capacity) and there is no room to accept the new item.

Note that the second value (QUEUED) is never returned for a capacity 0 mailbox.

If a put operation is issued for a typed mailbox that defines the inItem method (section 13.2.1), the method is called with the argument equal to the argument of put, but only when

- the mailbox capacity is > 0 and the item is ACCEPTED or QUEUED, or
- the mailbox capacity is zero

For a typed mailbox, this method checks whether the mailbox contains a specific item:

```
Boolean queued (itypename item);
```

where *itypename* stands for the item type. The method returns YES is the specified item is stored in the mailbox, and NO otherwise.

One of the following four methods can be called to determine the full/empty status of a mailbox:

```
int empty ();
int nonempty ();
int full ();
int notfull ();
```

The first method returns YES (1), if the mailbox is empty and NO (0) otherwise. The second method is a simple negation of the first. The third method returns YES, if the mailbox is completely filled. The fourth method is a straightforward negation of the third.

The following method removes the first item from a mailbox:

```
etype get ();
```

For a counting mailbox, *etype* is int and the method returns YES, if the mailbox was nonempty (in which case one pending token has been removed from the mailbox) and NO otherwise. For a typed mailbox, *etype* coincides with the mailbox element type. The method returns the value of the removed item, if the mailbox was nonempty (the first queued item has been removed from the mailbox), or 0 (NULL) otherwise. Note that 0 may be a legitimate value for an item. If this is possible, get can be preceded by a call to empty (or nonempty)—to make sure that the mailbox status has been determined correctly.

If a successful `get` operation is executed for a typed mailbox that defines the `outItem` method (section 13.2.1), the method is called with the removed item passed as the argument.

You may want to have a look at the first item of a typed mailbox without removing the item. The method

```
etype first ();
```

behaves like `get`, except that the first item, if present, is not removed from the mailbox. For a counting mailbox, the method type is `int` and the returned value is the same as for `nonempty`.

Methods `put`, `get`, and `first` set the environment variables `TheMailbox` and `TheItem` (section 13.2.2). `TheMailbox` is set to point to the mailbox whose method has been invoked and `TheItem` is set to the value of the inserted/removed/looked at item. For a counting mailbox, `TheItem` is always `NULL`.

Here is the operation to empty a mailbox:

```
int erase ();
```

The method removes all elements stored in the mailbox, so that immediately after a call to `erase` the mailbox appears `empty`. The value returned by the method gives the number of removed elements. In particular, if `erase` returns 0, it means that the mailbox was empty when the method was called. If `erase` is executed for a typed mailbox that defines the `outItem` method (section 13.2.1), the method is called individually for each removed item, the removed item passed as the argument.

Operations on mailboxes may trigger events awaited by processes (section 13.4.6). In particular, a `put` operation triggers `NEWITEM` (a.k.a. `PUT` and `UPDATE`) and may trigger `NONEMPTY`, `RECEIVE` and a count event. Operations `get` and `erase` trigger `GET` and may trigger count events (the latter also triggers `EMPTY`). Also, remember that deleting the mailbox itself triggers all possible events (see section 13.2.2).

Operations `get`, `first`, and `nonempty` performed on a capacity 0 mailbox always return 0. Note, however, that `full` returns `YES` for a capacity 0 mailbox: the mailbox is both full and empty at the same time (it contains no items and it cannot accommodate more items). The only way for a `put` operation on a capacity 0 mailbox to succeed, is to match a wait request for `NEWITEM` that must be already pending when the `put` is issued. Even then, there is no absolute certainty that the element will not be lost. This is because the process to be awakened may be awaiting a number of events (on different AIs) and some of them may be scheduled at the same ITU as the `NEWITEM` event. In such a case, the actual waking event will be chosen nondeterministically and it does not have to be the mailbox event. One possible way of coping with this problem is discussed in section 13.2.4.

Examples

Even if the NEWITEM event is the only event awaited by the process, synchronization based on capacity 0 mailboxes may be a bit tricky. Consider the following process:

```
process One (Node) {
    Mailbox *Mb;
    int Sem;
    void setup (int sem) { Sem = sem; Mb = &S->Mb; };
    states {Start, Stop};
    perform {
        state Start:
            if (Sem) {
                Mb->put ();
                proceed Stop;
            } else
                Mb->wait (NEWITEM, Stop);
        state Stop:
            terminate;
    };
};
```

and assume that two copies of the process are started at the same station s in the following way:

```
create (s) One (0);
create (s) One (1);
```

At first sight, it seems that both copies should terminate (in state Stop) within the *ITU* of their creation. However, if Mb is a capacity 0 mailbox, it need not be the case. Although the order of the create operations suggests that the process with Sem equal 0 is created first, all we know is that the two processes will be started within the same (current) *ITU*. If the second copy (the one with Sem equal 1) is *actually* run first, it will execute put before the second copy is given a chance to issue the wait request to the mailbox. Thus, the token will be ignored and the second copy will be suspended forever (assuming that the awaited token will not arrive from some other process).

Of course, it is possible to force the right startup order for the two processes, e.g., by rewriting the setup method in the following way:

```
void setup (int sem) {
    setSP (sem);
    Sem = sem;
```

```
        Mb = &S->Mb;
    };
```

i.e., by assigning a lower order to the startup event for the first copy of the process (section 7.4). One may notice, however, that if we know the order in which things are going to happen, there is no need to synchronize them.

Another way to solve the problem is to replace the `put` statement with the following condition:

```
if (Mb->put () == REJECTED) proceed Start;
```

This solution works, but it has the unpleasant flavor of "indefinite postponement." A nicer way to make sure that both processes terminate is to create `Mb` as a capacity 1 mailbox and replace `NEWITEM` with `NONEMPTY` or, even better, `RECEIVE`. Note that with `NEWITEM` the solution still wouldn't work. With `NONEMPTY`, the two processes would terminate properly, but the mailbox would end up containing a pending token.

Now let us have a look at an example of a safe application of a capacity 0 mailbox. The following two processes:

```
process Server (MyStation) {
    Mailbox *Request, *Reply;
    int Rc;
    void setup () {
        Rc = 0;
        Request = &S->Request;
        Reply = &S->Reply;
    };
    states {Start, Stop};
    perform {
        state Start:
            Request->wait (RECEIVE, Stop);
        state Stop:
            Reply->put (Rc++);
            proceed Start;
    };
};
process Customer (MyStation) {
    Mailbox *Request, *Reply;
    void setup () {
        Request = &S->Request;
        Reply = &S->Reply;
    };
```

```
        states { ..., GetNumber,... };
        perform {
          ...
            state GetNumber:
                Request->put ();
                Reply->wait (NEWITEM, GetIt);
            state GetIt:
                Num = TheItem;
              ...
        };
    };
```

communicate via two mailboxes `Request` and `Reply`. Note that when the `Server` process issues the `put` operation for `Reply`, the other process is already waiting for `NEWITEM` on that mailbox (and is not waiting for anything else). Thus `Reply` can be a capacity 0 mailbox. We assume that `Request` is a capacity 1 counting mailbox.

The following method:

```
Long getCount ();
```

returns the number of elements in the mailbox.

The capacity of a fifo mailbox can be modified and checked with these three methods:

```
Long setLimit (Long lim = 0);
Long getLimit ();
Long free ();
```

In particular, a call to `setLimit` can be put into a user-supplied setup method for a mailbox subtype (section 13.2.1). The argument of `setLimit` must not be negative. The method returns the previous capacity of the mailbox. The new capacity cannot be less than the number of elements currently present in the mailbox. Method `free` provides a shorthand for `getLimit()-getCount()`.

13.2.4 The priority put operation

Similar to the case of signal passing (section 7.8.2), it is possible to give a mailbox event a higher priority, to restart the process waiting for the event before anything else can happen. This mechanism is independent of event ranking (section 7.4), which provides another means to achieve similar effects. Besides the obvious solution of assigning the lowest order to the The present approach consists in triggering a *priority event* on the mailbox and assumes no cooperation from the event recipient.

A priority event on a mailbox is issued by the *priority put* operation implemented by the following `Mailbox` method:

```
int putP ();
```

For a typed mailbox, the method accepts one argument whose type coincides with the mailbox item type. The semantics of `putP` is similar to that of of regular `put`, i.e., the operation inserts a new item into the mailbox. The following additional rules apply solely to `putP`:

- At most one operation triggering a priority event (`putP` or `signalP`—see section 7.8.2) can be issued by a process from the moment it wakes up until it goes to sleep. This is equivalent to the requirement that at most one priority event can remain pending at any given time.
- At the moment when `putP` is executed, the mailbox must be empty and there must be exactly one process waiting on the mailbox for the new item. In other words: the exact deterministic fate of the new element must be known when `putP` is invoked.

If no process is waiting on the mailbox when `putP` is called, the method fails and returns `REJECTED`. Otherwise, `putP` returns `ACCEPTED`.

As soon as the process that executes `putP` is suspended, the waiting recipient of the new mailbox item is restarted. No other process is run in the meantime, even if there are other events scheduled at the current *ITU*.

The receiving process of a priority put operation need no special measures to implement its end of the handshake. It just executes a regular wait request[43] for `NEWITEM`, `NONEMPTY`, or a count 1 event. The priority property of the handshake is entirely in the semantics of `putP`.

Examples

Let us return to the second example from section 13.2.3 and assume that the `Customer` process communicates with two copies of the server, i.e., in the following way:

```
state GetNumber:
    NRcv = 0;
    Request1->put ();
    Request2->put ();
    Reply1->wait (NEWITEM, GetIt);
    Reply2->wait (NEWITEM, GetIt);
```

[43] The order of this request is irrelevant.

```
state GetIt:
    Num = TheItem;
    if (!NRcv++) {
        Reply1->wait (NEWITEM, GetIt);
        Reply2->wait (NEWITEM, GetIt);
        sleep;
    }
```

The process wants to make sure that it receives both items; however, unless the servers use `putP` instead of `put`, one item can be lost. Namely, the following scenario is possible:

- Server 1 executes `put`, and the customer is scheduled to be restarted in the current *ITU*.
- Before the customer is actually awakened, server 2 executes `put`. Thus, there are two events that want to restart the customer at the same time.
- The customer is restarted and it perceives only one of the two events. Thus, one of the items is lost.

One may try to eliminate the problem by assigning a very low order to the wait requests issued by the customer (section 7.4). Another solution is to make the servers use `putP` instead `put`. Then, each put operation will be immediately responded to by the customer, irrespective of the order of its wait requests. Yet another solution is to create mailboxes `Reply1` and `Reply2` with capacity 1 and replace `NEWITEM` with `RECEIVE`.

It might seem that the above piece of code could be rewritten in the following (apparently equivalent) way:

```
state GetNumber:
    NRcv = 0;
    Request1->put ();
    Request2->put ();
transient Loop:
    Reply1->wait (NEWITEM, GetIt);
    Reply2->wait (NEWITEM, GetIt);
state GetIt:
    Num = TheItem;
    if (!NRcv++) proceed Loop;
```

One should be careful with such simplifications. Note that `proceed` (see section 8.1) is actually a `Timer` wait request (for 0 *ITU*s). Thus, it is possible that when the second server executes `putP`, the customer is not ready to receive the item (although it will become ready within the current *ITU*). By replacing `proceed` with `sameas` (section 8.1), you will retain

the exact functionality of the original code, as `sameas` effectively "replicates" the body of state `Loop` at its occurrence (no state transition via the simulator is involved).

13.3 Barrier mailboxes

A barrier mailbox is an internal mailbox whose capacity is negative. The exact value of that capacity does not matter: it is used as a flag to tell a barrier mailbox from a fifo.

13.3.1 Declaring and creating barrier mailboxes

A barrier mailbox is usually untyped. The simplest way to set up a barrier mailbox is to use the default mailbox type and set its capacity to something negative, e.g.,

```
Mailbox *bm;
  ...
bm = create Mailbox (-1);
```

It may be convenient to introduce a non-standard type, to improve the clarity of code, e.g.,

```
mailbox Barrier {
  void setup () {
    setLimit (-1);
  };
};
```

Here, the setup method will automatically render the mailbox a barrier upon creation.

It may make some sense to use a typed mailbox for this purpose, with the item type being `int`, which coincides with the type of signals awaited and triggered on the mailbox. Thus, we can modify the above declaration into

```
mailbox Barrier (int) {
  void setup () {
    setLimit (-1);
  };
};
```

The primary advantage of the last type is that it may sensibly define an `inItem` method (section 13.2.1) to be called when a signal is triggered on the mailbox. There is no use for `outItem`, however.

13.3.2 Wait requests

The event argument of a wait request addressed to a barrier mailbox identifies the signal that the process wants to await. The event will occur as soon as some other process executes `signal` (section 13.3.3) whose argument identifies the same event.

At any moment, there may be an unlimited number of pending wait request issued to a barrier mailbox, identifying the same or different events. Whenever an event is `signal`ed, **all** processes waiting for this particular event are awakened. Other processes, waiting for other events, remain waiting until their respective events are signaled.

Note that events triggered on a barrier mailbox are never queued or pending. To be affected, a process must already be waiting on the mailbox at the moment the event is signaled. A signaled event not being awaited by any process is ignored.

When a process is awakened by an event on a barrier mailbox, the environment variable `TheMailbox` (`Info02`) of type `Mailbox`, points to the mailbox on which the event has occurred. `TheBarrier` (`Info01`) of type `int` contains the signal number that has triggered the event. Similar to other mailbox types, when a barrier mailbox is deleted (section 13.2.2), all processes waiting on it will be awakened. If a situation like this is possible, the awakened process should check the value of `TheMailbox`, which will be `NULL` if the mailbox has beed deleted.

13.3.3 Operations on barrier mailboxes

The practically single relevant operation on a barrier mailbox is provided by this method:

```
int signal (int sig);
```

which triggers the indicated signal (the barrier). All processes awaiting the signal are awakened. Similar to `put` (section 13.2.3), `trigger` returns `ACCEPTED` if there was at least one process awakened by the operation, and `REJECTED` otherwise. If the mailbox is typed (and its item type is `int`—section 13.3), `trigger` can be replaced by `put` (note that `put` in that case accepts an `int` argument).

A priority variant of `signal` (see sections 7.8.2 and 9.2) is also available. Thus, by calling

```
mb->signalP (sig);
```

the caller sends the signal as a priority event. This is similar to `putP`. The semantics is exactly as described in section 13.2.4, except that the event in question concerns waking up a process waiting on a barrier.

A barrier mailbox formally appears as empty and not full, i.e., `empty` returns `YES` (and `nonempty` returns `NO`), while `full` returns `NO` (and `notfull` returns `YES`). Its item count is always zero; thus, `erase` performed on a barrier mailbox is void and returns zero.

If the capacity of a barrier mailbox is changed to a nonnegative value (with `setLimit`—section 13.2.3), the mailbox becomes a fifo (section 13.2). This is seldom useful, if at all, but possible, similar to changing a fifo mailbox to a barrier (by resetting a nonnegative capacity to negative), except in a situation when there are outstanding wait requests on the mailbox. If that is the case, `setLimit` will abort the program with an error message.

13.4 Bound mailboxes

A mailbox that has been bound to a device or a TCP/IP port becomes a buffer for messages to be exchanged between the control program and the other party. One way to carry out this communication is to use the same standard operations `put` and `get`, which are available for an internal mailbox (section 13.2.3). For a bound mailbox, these operations handle only one byte at a time. This may be sufficient for exchanging simple status information, but is rather cumbersome for longer messages, and bound mailboxes provide other more efficient tools for operating on larger chunks of data.

13.4.1 Binding mailboxes

Any fifo mailbox (created as explained in section 13.2.1) can be bound to a device or a socket port. This operation is performed by the following method defined in class `Mailbox`:

```
int connect (int tp, const char *h, int port, int bsize = 0,
             int speed = 0, int parity = 0, int stopbits = 0);
```

If it succeeds, the method returns `OK` (zero). The first argument indicates the intended type of binding. It is a sum (Boolean or arithmetic) of several components/flags whose values are represented by the following constants:

`INTERNET`, `LOCAL`, **or** `DEVICE`
: This component indicates whether the mailbox is to be bound to a TCP/IP socket, to a local (UNIX-domain) socket, or to a device.

`CLIENT` **or** `SERVER`
: This flag is only applicable to a socket mailbox. It selects between the client and server mode of our end of the connection. In the client mode, `connect` will try to set up a connection to an already-existing (server) port of the other party. In the server mode, `connect` will set up a server port (without actually connecting it anywhere) and will expect a connection from a client. One additional flag that can be specified with `SERVER` is `MASTER` indicating a "master" mailbox that will be solely used to accept incoming connections (rather than sustain them).

`WAIT` **or** `NOWAIT`
: This flag only makes sense together with `SERVER` (i.e., if the connection is to be established in the server mode). With `WAIT`, the `connect` operation will block until there is a connection from a client. Otherwise, which is the default action if neither `WAIT` nor `NOWAIT` has been specified, the method will return immediately.

`RAW` **or** `COOKED`
: This flag is used together with `DEVICE`. If the device is a `tty` (representing a serial port), `RAW` selects the raw interface without any `tty`-specific preprocessing (line buffering, special characters). With `COOKED`, which is the default, the stream is preprocessed as for standard terminal i/o.

`READ` **and/or** `WRITE`
: These flags tell what kind of operations will be performed on the device to which the mailbox is bound. For a socket mailbox, this specification is irrelevant because both operations are always legitimate on a socket. For a device mailbox, at least one of the two options must be chosen. It is also legal to specify them both.

Upon a successful completion, `connect` returns `OK`. The operation may fail in a fatal way, in which case it will abort the program. This will happen if some formal rules are violated, i.e., the program tries to accomplish something formally impossible. The method may also fail in a soft way, returning `REJECTED` or `ERROR`—as described below. Here are the possible reasons for a fatal failure.

1. the mailbox is already bound, i.e., an attempt is made to bind the same mailbox twice
2. the mailbox is a barrier; it is illegal to bound barrier mailboxes
3. the mailbox is nonempty, i.e., it contains some items deposited while the mailbox wasn't bound that have not been retrieved
4. there are processes waiting for some events on the mailbox
5. there is a formal error in the arguments, e.g., conflicting flags

A bound mailbox (except for a master mailbox—section 13.4.4) is equipped with two buffers: one for input and the other for output. The `bsize` (buffer size) argument of `connect` specifies the common size of the two buffers (i.e., each of them is capable of storing `bsize` bytes). The argument redefines the capacity of the mailbox for the duration

of the connection. If the specified buffer size is zero, the (common) buffer size is set to the current capacity of the mailbox. Unless the mailbox is a master mailbox (section 13.4.4), the resulting buffer size **must be greater than zero**.

Information arriving at the mailbox from the other party will be stored in the input buffer, from where it can be retrieved by the operations discussed in section 13.4.7. If the buffer fills up, the arriving bytes will be blocked until some buffer space is reclaimed. Similarly, outgoing bytes are first stored in the output buffer, which is emptied asynchronously by the SIDE kernel. If the output buffer fills up, the mailbox will refuse to accept more messages until some space in the buffer becomes free. Both buffers are circular: each of them can be filled and emptied independently from two ends.

The contents of a bound mailbox, as perceived by a SIDE program, are equivalenced with the contents of its input buffer. For example, when we say that the mailbox is empty, we usually mean that the input buffer of the mailbox is empty. All operations of acquiring data from the mailbox, inquiring the mailbox about its status, etc., refer to the contents of the input buffer. The bytes deposited/written into the mailbox (the ones that pass through the output buffer) are destined for the other party and, conceptually, do not belong to the mailbox.

An attempt to bind a barrier mailbox, i.e., one whose capacity is negative, is treated as an error and aborts the program. Note that a barrier mailbox can be reverted to the non-barrier status by resetting its capacity with `setLimit` (section 13.3.3.

13.4.2 Device mailboxes

To bind a mailbox to a device, the first argument of `connect` should be `DEVICE` combined with at least one of `READ`, `WRITE`, and possibly with `RAW` or `COOKED`. The `COOKED` flag is not necessary: it is assumed by default if `RAW` is absent.

The second argument should be a character string representing the name (file path) of the device to which the mailbox is to be bound.

The third argument (`port`) is ignored and the fourth one (`bsize`) declares the buffer size for the connected mailbox. This number should be at least equal to the maximum size of a message (in bytes) that will be written to or retrieved from the mailbox with a single `read`/`write` operation (see section 13.4.7). The specified buffer size can be zero (the default), in which case the current capacity of the mailbox (section 13.2.1) will be used. The final length of the buffer must be strictly greater than zero.

The last three arguments are only relevant when the device is attached via a serial port and the first argument includes the `RAW` flag. They specify the port speed, the parity, and the number of stop bits, respectively. The legal values for `speed` are: 0, 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400. Value 0 indicates that the current speed setting for the port, whatever it is, should not be

changed.

The specified parity can be 0 (no parity), 1 (indicating odd parity), or 2 (selecting even parity). If `parity` is nonzero, the character size is set to seven bits. The last argument can be 0 (automatic selection of stop bits), 1, or 2.

Example

With the following call:

```
m3->connect (DEVICE+RAW+READ+WRITE, "/dev/ttyS1", 0, 256);
```

the mailbox pointed to by `m3` is bound to device `/dev/ttyS1`. The interface is raw (no `tty`-specific preprocessing will be done automatically by the system), and the mailbox will be used for both input and output. The buffer size is 256 bytes.

Below is the code of a (somewhat inefficient) process that copies one file to another.

```
mailbox File (int);

process Reader {
  File *Fi, *Fo;
  int outchar;
  setup (const char *finame, const char *foname) {
    Fi = create File;
    Fo = create File;
    if (Fi->connect (DEVICE+READ, finame, 0, 1))
      excptn ("cannot connect input mailbox");
    if (Fo->connect (DEVICE+WRITE, foname, 0, 1))
      excptn ("cannot connect output mailbox");
  };
  states {GetChar, PutChar};
  perform {
    state GetChar:
      if (Fi->empty ()) {
        if (Fi->isActive ()) {
          Fi->wait (UPDATE, GetChar);
          sleep;
        } else
          Kernel->terminate ();
      }
      outchar = Fi->get ();
    transient PutChar:
```

```
        if (Fo->put (outchar) != ACCEPTED) {
          if (Fo->isActive ()) {
            Fo->wait (OUTPUT, PutChar);
            sleep;
          } else
            excptn ("write error on output file");
        }
        proceed GetChar;
    };
};
```

See section 13.4.7 for the semantics of methods `empty` and `isActive`.

As we can see, bound mailboxes can be used to access files, but because of the non-blocking character of i/o, one has to be careful there. For example, it usually does not makes sense to bind the same mailbox to a file with both `READ` and `WRITE` flags, because SIDE will try to read ahead a portion of the file into the buffer even if no explicit `get`/`read` operation has been issued by the program. This may confuse the file position for a subsequent write operation.

If a mailbox is bound to a file with `WRITE` (but not `READ`), the file is automatically erased, i.e., truncated to zero bytes.

The `connect` operation will fail if the specified device/file cannot be opened for the required type of access. Such a failure is non-fatal: the method returns `ERROR`.

13.4.3 Client mailboxes

By a *socket mailbox* we understand a mailbox bound to a TCP/IP port, possibly across the Internet, or to a UNIX-domain socket confined to the current host. A client connection of this kind assumes that the other party (the server) has made its end of the connection already available. For a TCP/IP session, this end is visible as a port number on a specific host, whereas a local server is identified via the path name of a local socket.

For a client-type binding, the first argument of `connect` must be either `INTERNET+CLIENT` or `LOCAL+CLIENT`. The explicit `CLIENT` specification is not required as it is assumed by default if the `SERVER` flag is not used.

If the connection is local, the second argument of `connect` should specify the path name of the server's socket in the UNIX domain. For an Internet connection, this argument will be interpreted as the Internet name of the host on which the server is running. The host name can be specified as a DNS name, e.g., `sheerness.cs.ualberta.ca`, or as an IP address (in a string form), e.g., `129.128.4.33`.

The third argument (`port`) is ignored for a local connection. For an Internet connection,

it specifies the server's port number on the remote host.

The fourth argument (bsize) gives the mailbox buffer size. The rules that apply here are the same as for a device mailbox (section 13.4.2). In particular, the buffer size for a client mailbox must be strictly greater than zero. The remaining arguments do not apply to socket mailboxes and are ignored.

Examples

The following call:

```
m1->connect (CLIENT+INTERNET, "sheerness.cs.ualberta.ca", 2002, 256);
```

binds the mailbox pointed to by m1 to port 2002 on sheerness.cs.ualberta.ca.

A local client-type binding is illustrated by the following call:

```
m2->connect (LOCAL, "/home/pawel/mysocket", 0, 1024);
```

Note that the CLIENT specification is not necessary (it is assumed by default).

A connect operation will fail if the specified port is not available on the remote host, the remote host cannot be located, or the path name does not identify a local UNIX-domain socket (for a local connection). In such a case, connect returns ERROR.

13.4.4 Server mailboxes

With a server mailbox, a SIDE program can accept incoming connections from Internet or UNIX-domain sockets. To set up a server Mailbox, the first argument of connect should be INTERNET+SERVER (for an Internet mailbox) or LOCAL+SERVER (for a UNIX-domain connection). Additionally, at most one of two flags WAIT and MASTER can be included in the first argument.

For a TCP/IP mailbox, the second argument of connect (h) is ignored and the third argument (port) specifies the port number for accepting incoming connections. For a UNIX-domain setup, the second argument specifies the path name of the local socket and the third argument is ignored. The following variant of connect has been provided for setting up server mailboxes bound to the Internet:

```
int connect (int tp, int port, int bsize = 0) {
  return connect (tp, NULL, port, bsize);
};
```

If all that is expected from the server mailbox is to accept one connection, its operation is simple and closely resembles the operation of its client counterpart. This type of a bound

mailbox will be called an *immediate server mailbox*. It is selected by default, if the first argument of `connect` does not include the `MASTER` flag.

For an immediate server mailbox, the `bsize` argument of `connect` is relevant and indicates the buffer size at the server end. The same rules apply as for a client/device mailbox (section 13.4.2). The remaining arguments are ignored.

As soon as it has been successfully bound, an immediate server mailbox becomes usable in exactly the same way as a client mailbox. Until the other party connects to it and sends some data, the mailbox will appear empty to reading operations (section 13.4.7). Also, any data deposited in the mailbox by writing operations (section 13.4.7) will not be expedited until the other end of the connection becomes alive.

While binding an immediate server mailbox, the program can request the `connect` operation to block until the other party connects. This is accomplished by adding `WAIT` to the list of flags comprising the first argument of `connect`.

Examples

The following two operations:

```
m1->connect (INTERNET+SERVER, 3345, 4096);
m2->connect (LOCAL+SERVER+WAIT, "/home/pawel/mysocket", 1024);
```

bind two immediate server mailboxes. The first mailbox is bound to a TCP/IP port (number 3345) and its buffer size is 4096 bytes. In the second case, the mailbox is local (a UNIX-domain socket) and the operation blocks until the client connects. The buffer size is 1024 bytes.

An immediate server mailbox can be recycled for multiple connections, but those connections must be serviced sequentially—one at a time. A connection can be closed without unbinding the mailbox (see operation `disconnect` in section 13.4.5), which will make the mailbox ready to accommodate another client.

To handle multiple client parties requesting service at the same time, the program must set up a *master* mailbox. This is accomplished by adding `MASTER` to the flags specified in the first argument of `connect`.

The sole purpose of a master mailbox is to accept connections. Once it has been accepted, the connection will be represented by a separate mailbox which has to be *bound* to the master mailbox.

As a master mailbox is only used for accepting connections (rather than sustaining them), it needs no buffer storage for incoming or outgoing data. Thus, the buffer size specification for a master mailbox (argument `bsize`) is ignored. Moreover, the `WAIT` flag cannot be specified for a master mailbox.

The following method of Mailbox can be used to tell whether a master mailbox is connection pending (meaning that an incoming connection awaits acceptance):

```
int isPending ();
```

The method returns YES if there is a pending connection on the mailbox, and NO otherwise. It also returns NO if the mailbox is not bound or it is not a master mailbox.

When a master mailbox becomes connection pending, it triggers a NEWITEM (UPDATE) event (section 13.4.6). The same event triggered on a regular (bound) mailbox indicates the availability of input data.

The following variant of connect is used to accept a connection on a master mailbox:

```
int connect (Mailbox *m, int bsize = 0);
```

The first argument specifies the master mailbox on which the connection is pending. The second argument (bsize) specifies the buffer size for the connection. The method binds its mailbox to the connection. The character of this binding (Internet/local) is inherited from the master mailbox.

Example

Below we show the code of a simple process distributing connections arriving on a master mailbox.

```
process Server {
  Mailbox *Master;
  void setup (int Port) {
    Master = create Mailbox;
    if (Master->connect (INTERNET+SERVER+MASTER, Port) != OK)
      excptn ("Server: cannot setup MASTER mailbox");
  };
  states {WaitConnection};
  perform {
    state WaitConnection:
      if (Master->isPending ()) {
        create Service (Master);
        proceed WaitConnection;
      }
      Master->wait (UPDATE, WaitConnection);
  };
};
```

Note that the `Server` process creates a separate process (an instance of `Service`) for every new connection. The recommended structure of such a process is as follows:

```
process Service {
  Mailbox *Local;
  void setup (Mailbox *master) {
    Local = create Mailbox;
    if (Local->connect (master, BUFSIZE) != OK) {
      delete Local;
      terminate ();
    }
  };
  states {...};
  perform;
 ...
};
```

The `connect` operation that binds a mailbox to a master mailbox returns `OK` upon success. It will fail, returning `REJECTED`, if the mailbox specified as the first argument is not bound, is not a master mailbox, or is not connection pending.

13.4.5 Unbinding and determining the bound status

A bound mailbox can be unbound with the following `Mailbox` method:

```
int disconnect (int how = SERVER);
```

Depending on the type of the bound mailbox, the method behaves in the following way:

Device mailbox

The device is closed, the buffers are deallocated, the mailbox is reverted to its unbound status.

Client mailbox

The client's end of the socket is closed, the buffers are deallocated, the mailbox is reverted to its unbound status.

Immediate server mailbox

The current connection (the socket) is closed. If the method has been called with SERVER specified as the argument (the default), the buffers are deallocated and the mailbox is reverted to its unbound status. If the argument is CLIENT, only the current connection is closed. The mailbox remains bound and ready to accommodate another connection (section 13.4.4).

Master mailbox

If the method has been called with SERVER (the default), the master socket is closed and the mailbox is reverted to its unbound status. If the argument is CLIENT, the operation is ignored.

Upon a successful completion, the method returns OK. This also happens when disconnect has been called for an unbound mailbox, in which case it does nothing. The method returns ERROR (and fails to perform its task) in the following circumstances:

1. there are some processes waiting on the mailbox
2. the mailbox is nonempty, i.e., the input buffer contains some pending data
3. the output buffer contains some data that hasn't been expedited from the mailbox

It is possible to force unbinding even if one or more of the above conditions hold. To accomplish this, disconnect should be called with CLEAR specified as the argument. In such a case the method performs the following operations:

1. every process waiting on the mailbox for one of the following events: NEWITEM (UPDATE), SENTINEL, OUTPUT, or a count event (sections 13.4.6 and 13.4.7) receives the awaited event
2. both buffers are erased and reset to the empty state (this operation is void for a master mailbox)
3. the operation continues as if disconnect were called with CLIENT

Note that disconnect (CLEAR) does not change the status of a master mailbox. For an immediate sever mailbox, it only terminates the current connection.

This kind of a formal disconnection is forced automatically by SIDE when it detects that the connection has been dropped by the other party or broken because of a networking problem. The program can monitor such events and respond to them, e.g., with the assistance of the following method:

```
int isConnected ();
```

which returns YES if the mailbox is currently bound and NO otherwise. Note that isConnected will return YES for a bound immediate server mailbox, even though there may be no connection on the mailbox at the moment. The following method:

```
int isActive ();
```

takes care of this problem: it returns YES only if the mailbox is currently "connected" to some party. In particular, it always returns NO for a master mailbox.

To find out that there is a problem with the connection, the program should invoke isActive in response to a NEWITEM, SENTINEL, or a count event, and also when a read or write operation (section 13.4.7) fails. If isActive returns NO, it means that the connection has been closed by the other party or broken. Note that the mailbox is automatically unbound in such a case: there is no need to perform disconnect on it, although it will do no harm.

A drastic but effective way to completely close a connection and forget about it is to simply delete the mailbox.[44] This has the effect of closing any connection currently active on the mailbox, deallocating all buffers, and waking up all processes waiting on it. An awakened process can find out that the mailbox does not exist any more by examining TheMailbox, which will be NULL in that case.

13.4.6 Wait requests

Practically all events that can be awaited on a fifo mailbox (section 13.2.2) make also sense for a bound mailbox, with the interpretation of its input buffer as the item queue, where the items are individual bytes awaiting extraction. In particular, for NONEMPTY, the mailbox is assumed to be nonempty is its input buffer contains at least one byte.

A NEWITEM (a.k.a UPDATE or PUT) event is triggered when anything new shows up in the mailbox's input buffer. Also, for a master mailbox, this event indicates a new connection awaiting acceptance.

For a bound mailbox, the RECEIVE event occurs when the input buffer contains at least one byte. The first byte is removed from the buffer and stored in TheItem before the process responding to RECEIVE is run.

The count event has a slightly different semantics for a bound mailbox. Namely, the event is triggered when the number of bytes in the input buffer **reaches or exceeds** the specified count. Its primary purpose is to be used together with read (section 13.4.7) for extracting messages from the mailbox in a non-blocking way.

[44] This must be a process mailbox rather than one owned by a station (section 13.2.1).

Two events, OUTPUT and SENTINEL, can only be awaited on a bound mailbox. The first one is triggered whenever some portion (at least one byte) of the output buffer associated with the mailbox is emptied and sent to the other party. The event is used to detect situations when it makes sense to retry a write operation (section 13.4.7) that previously failed.

One possible way to retrieve a block of data from a bound mailbox is to use readToSentinel, which operation reads a sequence of bytes up to a declared marker (sentinel) byte (section 13.4.7). A process that wants to know when it is worthwhile to try this operation may issue a wait request for the SENTINEL event. This event will be triggered as soon as the input buffer either contains at least one sentinel byte or becomes completely filled. If this condition holds at the moment when the wait request is issued, the event occurs immediately.

The reason why the SENTINEL event is triggered by a completely filled buffer (even if it does not contain the sentinel byte) is that in such a situation there is no way for the buffer to receive the sentinel until some portion of it is removed. Sentinels are used to separate blocks of data with some logical structure (e.g., lines of text read from the keyboard) and if an entire block does not fit into the input buffer at once, its portions must be read in stages, and only the last of them will be actually terminated by the sentinel byte.

When a SENTINEL event is received, the environment variable TheCount of type int (an alias for Info01) contains the number of bytes in the input buffer preceding and including the nearest occurrence of the sentinel. If the sentinel does not occur in the buffer (the event has been triggered because the buffer is full), TheCount contains zero.

When a bound mailbox becomes forcibly disconnected by disconnect (CLEAR) (section 13.4.5), which will also happen when the connection is closed or broken, or the mailbox is deleted (section 13.4.5), all awaited NEWITEM, SENTINEL, OUTPUT, and count events are triggered. This way, a process waiting to receive or expedite some data through the mailbox will be awakened and it will be able to respond to the new status of the mailbox. Consequently, to make its operation foolproof, the process cannot assume that any of the above events always indicates the condition it was meant for. The recommended way of acquiring/expediting data via a bound mailbox is discussed in section 13.4.7.

13.4.7 Operations on bound mailboxes

In principle, the standard operations put and get described in section 13.2.3 are sufficient to implement network communication via a bound mailbox.

For a bound mailbox, put behaves as follows. If the mailbox is typed (put takes an argument), the least significant byte from the argument is extracted. If the output buffer of the mailbox (section 13.4.1) is not completely filled, the byte is deposited at the end of the outgoing portion of the buffer and put returns ACCEPTED. If the buffer is full, nothing happens and put returns REJECTED. If the bound mailbox is a counting (untyped) mailbox

(`put` takes no argument), a zero byte is expedited—in exactly the same way. This makes generally no sense; thus, if you want to use this operation for serious communication over a bound mailbox, make sure that the mailbox is typed (preferably with `int` being the item type).

An extraction by `get` retrieves the first byte from the input buffer. If the mailbox is a counting (untyped) one, the byte is removed and ignored, so the only information returned by `get` in such a case is whether a byte was there at all (the method returns `YES` or `NO`—section 13.2.3). For a typed mailbox, the extracted byte is returned on the least significant position of the method's value. The operation of `first` mimics the behavior of `get`, except that it only peeks at the byte without removing it from the buffer.

If a typed bound mailbox defines `inItem` and/or `outItem`, the methods will be called as described in sections 13.2.1 and 13.2.3, with the deposited/extracted bytes interpreted as the items. This only happens when the communication involves `put` and `get`. Also, `erase`, which can be used to completely erase the input buffer of a bound mailbox will call `outItem` for every single byte removed from the buffer. The methods `empty`, `nonempty`, `full`, `notfull` tell the status of the input buffer, and their meaning is obvious. Also, `getCount` tells the number of bytes present in the input buffer.

Note: operations `setLimit` and `putP` are illegal while the mailbox is bound. However, the size of the mailbox buffer can be enlarged dynamically, at any time, with this `Mailbox` method:

```
int resize (int newsize);
```

where the argument specifies the new size. If the new size is less than or equal to the present buffer size (limit), the method does nothing. Otherwise, it reallocates the buffers according to the new increased size and preserves their contents. In both cases, the method returns the old size.

Sending and receiving one byte at a time may be inefficient for serious data exchange. Therefore, the following additional methods have been made available for a bound mailbox:

```
int read (char *buf, int nc);
int readAvailable (char *buf, int nc);
int readToSentinel (char *buf, int nc);
int write (const char *buf, int nc);
```

The first operation attempts to extract exactly `nc` bytes from the input buffer of the mailbox and store them in `buf`. If the buffer does not hold that many bytes at the moment, `read` returns `REJECTED` and does nothing. Otherwise, it returns `ACCEPTED` and removes the acquired bytes from the input buffer.

It is an error to specify `nc` larger than the mailbox buffer size (section 13.4.1), because the operation would have no chance of succeeding in that case.

If the operation fails (i.e., returns REJECTED), the issuing process may decide to wait for a count event equal to the requested number of bytes (section 13.4.6). For a bound mailbox, this event will be triggered as soon as the input buffer becomes filled with **at least** the specified number of bytes.

The second operation (readAvailable) extracts at most nc bytes from the mailbox. The method does not fail (as read does) if less than nc bytes are available. The function returns the actual number of extracted bytes. This number will be zero if the mailbox happens to be empty.

The third read operation (readToSentinel) attempts to extract from the input buffer of the mailbox a block of bytes terminated with the sentinel byte (the sentinel will be included in the block). Prior to using this operation, the program should declare a sentinel for the mailbox by calling the following method:

```
void setSentinel (char sentinel);
```

The default sentinel, assumed when setSentinel has never been called, is the null byte.

The second argument of readToSentinel specifies the limit on the number of bytes that can be extracted from the mailbox (the capacity of the buffer pointed to by the first argument). The operation succeeds, and acquires data from the mailbox, if the mailbox buffer either contains at least one occurrence of the sentinel or it is completely filled (section 13.4.6). In such a case, the function returns the number of extracted bytes. Note that if the second argument of the method is less than the number of bytes preceding and including the sentinel, only an initial portion of the block will be read. If the sentinel never occurs in the mailbox buffer and that buffer is not completely filled, the function fails and returns zero. The issuing process can wait for a SENTINEL event (section 13.4.6) and then retry the operation.

Note that readToSentinel succeeds when the mailbox buffer is completely filled, even if no sentinel is present in the buffer. The reason for this behavior was given in section 13.4.6. It makes no sense to wait for the sentinel in such a case, as some room must be freed in the buffer before the sentinel can possibly arrive. The calling process can easily determine what happened (without searching the acquired data for the sentinel), by examining the Boolean environment variable TheEnd (an alias for Info01). Its value is YES if the sentinel is present in the acquired data chunk, and NO otherwise.

The following Mailbox method:

```
sentinelFound ();
```

checks whether the mailbox buffer contains at least one sentinel byte. It returns the number of bytes preceding and including the nearest sentinel, or zero if no sentinel is

present. Note that the method will returns zero also when the mailbox buffer is completely filled.

The `write` operation acts in the direction opposite to `read`. It succeeds (returns `ACCEPTED`) if the specified number of bytes could be copied from `buf` to the output buffer of the mailbox. Otherwise, if there is not enough room in the buffer, the method returns `REJECTED` and leaves the buffer intact.

However, there is some inherent asymmetry between `read` and `write`. Generally it is more natural to fail for `read` than for `write`. In the first case, the expected message may be some sensor data arriving from a remote component of the control system, and such data only arrives when there is something relevant to report. Therefore, it is natural for a process monitoring a bound mailbox to mostly wait—for a count, `NEWITEM`, or `SENTINEL`—until something arrives.

On the other hand, the inability to write, e.g., expedite a status change message to a remote component (e.g., an actuator) is something much less natural and usually abnormal. If the output buffer of a bound mailbox is filled and unable to accommodate a new outgoing message, it typically means that there is something wrong with the network: it cannot respond as fast as the control program would like it to respond.

Because of this asymmetry, there is no count event that would indicate the ability of the output buffer to accept a given number of bytes. Instead, there is a single event (`OUTPUT`—section 13.4.6), which is triggered whenever any portion of the output buffer is flushed to the network. A process that cannot send its message immediately (because `put` or `write` fails) may use this event to determine when the operation can be sensibly retried. In most cases, the `OUTPUT` event indicates that the entire buffer (or at least a substantial part of it) has become available.

It is possible, however, to check whether the output buffer contains any pending data that has not been sent yet to the other party. For example, before deleting a bound mailbox (which operation immediately discards all buffered data), you may want to wait until all output has been flushed. This `Mailbox` method:

```
Long outputPending ();
```

returns the outstanding number of bytes in the output buffer that are still waiting to be expedited.

Example

Assume that we would like to set up a process mapping sensor messages arriving from remote physical equipment into simple events occurring on an internal mailbox. The following mailbox type:

```
mailbox NetMailbox (int);
```

will be used to represent the remote sensor, and the following process will take care of the conversion:

```
process PhysicalToVirtual {
  SensorMailbox *Sm;
  NetMailbox *Nm;
  void setup (SensorMailbox, const char*, int, int);
  states {WaitStatusChange};
};
```

We assume that `SensorMailbox` is a simple internal mailbox on which the status messages arriving from the physical sensor will be perceived in a friendly and uniform way. The setup method of our process is listed below.

```
void PhysicalToVirtual::setup (SensorMailbox *s,
                      const char *host, int port, int bufsize) {
  Sm = s;
  Nm = create NetMailbox;
  if (Nm->connect (INTERNET+CLIENT, host, port, bufsize) != OK)
    excptn ("Cannot bind client mailbox");
  Nm->put (START_UPDATES);
};
```

Its argument list consists of the pointer to the internal mailbox, the host name to connect to, the port number on the remote host, and the buffer size for the network mailbox. The method copies the mailbox pointer to its internal attribute, creates the network mailbox and connects it to the host (in the client mode—section 13.4.3). Then it sends a single-byte message to the remote host. We assume that this message will turn on the remote sensor and force it to send the updates of its status.

Now we can have a look at the following code method of our process:

```
PhysicalToVirtual::perform {
  char msg [STATMESSIZE];
  state WaitStatusChange:
    if (Nm->read (msg, STATMESSIZE) == ACCEPTED) {
      Sm->put (msg [STATUSVALUE0] * 256 + msg [STATUSVALUE1]);
      proceed WaitStatusChange;
    } else
      Nm->wait (STATMESSIZE, WaitStatusChange);
};
```

In its single state, the process expects a message from the remote sensor. If `read` succeeds, the process extracts the new value sent by the the sensor and passes it to the internal mailbox. If the message is not ready (`read` fails), the process suspends itself until `STATMESSIZE` bytes arrive from the sensor.

The above code assumes that nothing ever goes wrong with the connection. To make it foolproof, we should check the reason for every failure of the `read` operation, e.g.,

```
...
 state WaitStatusChange:
   if (Nm->read (msg, STATMESSIZE) == ACCEPTED) {
     Sm->put (msg [STATUSVALUE0] * 256 + msg [STATUSVALUE1]);
     proceed WaitStatusChange;
   } else if (Nm->isActive ()) {
     Nm->wait (STATMESSIZE, WaitStatusChange);
   } else {
     // The connection has been dropped
     ...
     terminate;
   }
...
```

Note that it is legal to issue a `read` or `write` operation (including the other two variants of `read`) to an unbound mailbox, but then the operation will fail (returning `ERROR`). Automatic unbinding (`disconnect (CLEAR)`) is forced by SIDE when the connection is closed by the other party or broken (section 13.4.5). Also, a process waiting for a count event on a mailbox being forcibly disconnected will receive that event (section 13.4.6). One way to find out that there has been a problem with the connection is to execute `isActive` (section 13.4.5).

13.5 Journaling

A mailbox bound to a TCP/IP port or a device can be journaled. This operation consists in saving the information about all relevant transactions performed on the mailbox in a file. This file, called a *journal file*, can be used later (in another run) to feed the same or a different mailbox whose input would be otherwise acquired from a TCP/IP port or a device. The journaling capability is only available if the program has been compiled with the `-J` option of `mks` (section 19.2) and only if one of `-R` or `-W` has been selected as well.

Applications of journal files are two-fold. First, journals can be used to collect event traces from real physical equipment, e.g., to drive realistic models of the physical object. Another use for journal files is to recover from a partial crash of a distributed control program.

Imagine two SIDE programs, P_1 and P_2, such that P_1 sends some information to P_2. The two programs communicate via a pair of mailboxes, one mailbox operating at each party. Assume that the mailbox used by P_1 is journaled. This means that all data expedited by the program via that mailbox are saved in a local file. If for some reason P_2 fails at some point, and all the information it has received from P_1 is destroyed, then P_2 can be rerun with its mailbox driven by the journal file. The net effect of this operation will be the same as if the mailbox of P_2 were filled with the data originally sent by P_1.

Of course, P_2 is free to journal its mailbox as well—saving all data received from P_1 in its local file. If that file is available after the crash, it can be used to the same effect.

13.5.1 Declaring mailboxes to be journaled

The operation of journaling (as well as the operation of driving a mailbox from a journal file) is transparent to the control program, i.e., the program does not have to be modified before its mailboxes can be journaled. When the program is called, the mailboxes to be journaled can be specified through a sequence of call arguments with the following syntax (section 19.3):

`-J` *mailboxident*

where *mailboxident* is either the standard name or the nickname of the mailbox (section 5.2). Nicknames are generally more useful for this purpose, as standard names of mailboxes (section 13.2.1) may be long, contrived and inconvenient. Therefore, it is recommended that important mailboxes—the ones that may be journaled—be assigned (unique) nicknames (section 5.8).

Note that *mailboxident* should appear as a single token on the call argument list; therefore, if the mailbox name includes blanks or other problem characters, then either each of those characters must be escaped or the entire name must be quoted.

For each mailbox marked in the above way, SIDE opens a journal file in the directory in which the program has been called. The name of this file is obtained from *mailboxident* by replacing all non-alphanumeric characters with underscores ('_') and adding the suffix "`.jnj`" to the result of this operation. As all journal files must have distinct names, the user should be careful in selecting the nicknames of journaled mailboxes.

The journal file **must not exist**; otherwise, the program will be aborted. SIDE is obsessive about not overwriting existing journal files.

Formally, the idea behind journaling is very simple. Whenever the mailbox is connected (i.e., the `connect` operation—section 6.3.2—is performed on the mailbox), a new *connect block* is started in the journal file. Whenever something is expedited from the mailbox to the other party (which may be a TCP/IP port or a device) that data is written to the journal file along with the time of this event. Whenever some data arrives from the

other party and is put into the input buffer of the mailbox, that data is also written to the journal file. Of course, sent and received data are tagged differently. Thus, the contents of the journal file describe the complete history of all external activities on the mailbox. Note, however, that the details of the specific internal operations performed by the program (`get`, `put`, `read`, `write`, etc.) are not stored.

13.5.2 Driving mailboxes from journal files

To drive a mailbox from a journal file, the following argument must be included in the call argument list of the program (one per each mailbox to be driven from a journal):

`-I` *mailboxident*

or

`-O` *mailboxident*

In the first case, the mailbox input will be matched with the input section of the journal file. This option makes sense if the file was created by journaling the same mailbox. In the second case, the mailbox input will be fed by the output data extracted from the journal file. This makes sense if the journal comes from a mailbox that was used by some other program to send data to the specified mailbox.

The name of the file to be used for driving the indicated mailbox is obtained in a way similar to that described in section 13.5.1. The file suffix is ".jnx". Note that it is different from the suffix of a created journal file; thus, the original file must be renamed or copied before it can be used to feed a mailbox.

A mailbox declared as driven from a journal file is never connected to a TCP/IP port or a device. The only effect of a `connect` operation on such a mailbox is to advance the feeding journal file to the next connect block (section 13.5.1). The input buffer of the mailbox is fed with the data extracted from the file, at the time specified by the time stamps associated with those data. Whatever is written to the mailbox by the program is simply discarded and ignored. Although this may seem strange at first sight, it is the only sensible thing to do.

By default, the time stamps in a journal file refer to the past, compared to the execution time of the program whose mailbox is driven by this file. Note that this past may be quite distant. This will have the effect of all the data arriving at the mailbox being deemed late and thus made available immediately, i.e., as soon as some data are extracted from the mailbox, new data will be fed in from the journal file. This need not be harmful. Typically, however, during a recovery, the program whose mailboxes are driven from journal files is run in shifted real time. This can be accomplished by using the `-D` call argument (section 19.3). To make sure that the time is shifted properly, the program can be called

with "`-D J`" (section 19.3), which has the effect of setting the real time to the earliest creation time of all journal files driving any mailboxes in the program.

A mailbox being driven by a journal file can be journaled at the same time (note that the suffixes of the two files are different, although their prefixes are the same). This is less absurd than it seems at first sight, if we recall that the output of such a mailbox is otherwise completely ignored.

13.5.3 Journaling client and server mailboxes

Journaling is primarily intended for mailboxes that do not change their bound status during the execution of the control program. However, it may also make sense for mailboxes that are bound and unbound dynamically, possibly several times. For a client-type mailbox (section 13.4.3), the semantics of journaling is quite simple. If the mailbox is being journaled (i.e., its transactions are backed up), each new connection starts a new *connect block* in the journal file. When the same (or another) mailbox is later driven from the the journal file, every new connection advances the file to the next connect block. Having exhausted the data in the current connect block, the mailbox will behave as if the other party has closed the connection, i.e., the mailbox will be disconnected (section 13.4.5). A subsequent `connect` operation on the mailbox will position the journal file at the beginning of the next connect block. If there are no more connect blocks in the file, the `connect` operation will fail returning `REJECTED`—as for a failed connection to a nonexistent server.

Server mailboxes are a bit trickier. If an immediate server mailbox (section 13.4.4) is being backed up to a journal file, the data written there are not separated into multiple connection blocks, even if the session consists of several client connections. Only a `SERVER`-type disconnection and reconnection creates a new connect block. Note that for an immediate mailbox, the program may not be able to tell apart two separate client connections directly following one another. Although the *active* status of the mailbox (determined by `isActive`—section 13.4.5) changes when the connection is dropped, if another connection comes up before the program inquires about that status, the transition will pass unnoticed. Therefore, it is assumed that all these connections appear as a single connection and the journal file does not separate them. When an immediate server mailbox is driven from a journal file, it receives a continuous stream of data that appears to have arrived from a single connection.

When a journaled immediate server mailbox is unbound (by `disconnect (SERVER)`—section 13.4.5), the current connect block is terminated. When the mailbox is bound again (by `connect`) a new connect block is started. Therefore, from the viewpoint of journaling, an immediate server mailbox behaves similarly to a client mailbox, with every explicit `connect` operation marking a new session.

When a master server mailbox is journaled, the only information written to the journal file is the timing of the incoming connections. The only sensible journaling setup involving

a master mailbox is one in which all mailboxes subsequently bound to the master mailbox are also journaled. The right way to feed them from their respective journal files is to create them in the proper order, based on the timing of connections fed from the journal file to the master mailbox. Their names (preferably nicknames—sections 5.2 and 13.5.1) should be distinct and generated in the right order—to match the corresponding journal files, in the order in which they have been created.

14 Measuring performance

One objective of modeling physical systems in SIDE is to investigate their performance. Some performance measures are calculated automatically by the package. The user can easily collect additional statistical data which augment or replace the standard measurements. The calculation of both the standard and user-defined performance measures is based on the concept of a *random variable* represented by a special data type.

14.1 Type `RVariable`

A `RVariable` is a data structure used for incremental calculation of some empirical distribution parameters of a random variable whose values are discretely sampled. The following parameters are taken into account:

- the number of samples (the number of times the random variable was sampled),
- the minimum value encountered so far,
- the maximum value encountered so far,
- the mean value,
- the variance (and standard deviation),
- higher-order central moments.

Type `RVariable` is not extensible by the user and its attributes are hidden. For efficiency reasons, they don't represent the above-listed parameters directly and some operations are required to turn them into a presentable form. Type `RVariable` declares a number of publicly visible methods for performing typical operations on random variables and for presenting their parameters in a legible form. The latter methods are based on the concept of *exposing*, which is common for all *Objects* (see section 17.5.4).

From now on, the objects of type `RVariable` will be called random variables.

14.1.1 Creating and destroying random variables

Random variables are *Objects* (section 5.1): they must be created before they are used, and they may be deallocated when they are no longer needed. A random variable is created in the standard way (section 5.8), e.g.,

```
rv = create RVariable (ct, nm);
```

where `rv` is an `RVariable` pointer and both setup arguments are integer numbers. The first argument (`ct`) specifies the type of the sample counter. The value of this argument can be either `TYPE_long`, in which case the counter will be stored as a `LONG` number, or `TYPE_BIG`, in which case the counter will be an object of type `BIG` (`TIME`).

The second argument (`nm`) gives the number of central moments to be calculated for the random variable. No moments are calculated if this number is 0. Note that the mean value and variance are moments number 1 and 2, respectively. The maximum number of moments is 32.

The setup method of `RVariable` is declared in the following way:

```
void setup (int ct = TYPE_long, int nm = 2);
```

Thus, if no setup arguments are specified at `create`, the random variable has a `LONG` sample counter and keeps track of two central moments, i.e., the mean value and variation.

Being an *Object*, a random variable is assigned an `Id` which is the variable's serial number in the order of creation. `Id`s of random variables are not useful for identification. Unlike for other *Objects*, there is no operation that would convert an `RVariable Id` into the object pointer. A random variable can be assigned a nickname (section 5.2) upon creation. The nickname version of `create` (section 5.8) should be used for this purpose.

A random variable that is no longer needed can (and should) be erased by the standard C++ operation `delete`.

14.1.2 Operations on random variables

When a new random variable is created, the value of its sample counter is initialized to zero. Whenever a new sample, or a number of samples, is to be added to the history of a random variable, the following `RVariable` method should be called:

```
void update (double val, CTYPE cnt = 1);
```

where `CTYPE` is either `LONG` or `BIG`, depending on the type of the random variable's sample counter. The function increments the sample counter by `cnt` and updates the parameters of the random variable according to `cnt` occurrences of value `val`. For example, consider the following sequence of operations:

```
r = create RVariable (TYPE_long, 3);
r->update (2.0, 1);
r->update (5.0, 2);
```

The first call to `update` adds to the variable's history one sample with value 2, the second call adds two more samples with the same value of 5. Thus, after the execution of the above three statements, `s` represents a random variable with three values: 2, 5, 5, and the distribution parameters of this random variable are:

minimum value	2.0
maximum value	5.0
mean value	4.0
variance	2.0
3-rd central moment	–2.0

Owing to the fact the attributes of an `RVariable` do not represent directly the distribution parameters, a special operation is required to produce these parameters. This operation is implemented by the following method of `RVariable`:

```
void calculate (double &min, double &max, double *m, CTYPE &c);
```

All arguments of `calculate` are return arguments. The type of `c` (denoted by `CTYPE`) must be the same as the counter type of the random variable, i.e., either `LONG` or `BIG`. The four return arguments are filled (in this order) with the minimum value, the maximum value, the moments, and the the sample counter. Argument `m` should point to a `double` array with no less elements than the number of moments declared when the random variable was created. The first element of the array (element number 0) will contain the mean value, the second—the variance, the third—the third central moment, and so on.

Two random variables can be combined into one in such a way that the combined parameters describe a single sampling experiment in which all samples belonging to both source variables have been taken into account. The function

```
void combineRV (RVariable *a, RVariable *b, RVariable *c);
```

combines the random variables `a` and `b` into a new random variable `c`. The target random variable must exist, i.e., it must have been created previously.

If the two random variables being combined have different numbers of moments, the resulting random variable has the smaller of the two numbers of moments. If the counter types of the source random variable are different, the resulting counter type is `BIG`.

Note: `combineRV` is a global function, not a method of `RVariable`.

It is possible to erase the contents of a random variable, i.e., initialize its sample counter to zero and reset all its parameters. This is done by the following method of `RVariable`:

```
void erase ();
```

Each random variable is automatically erased when it is created.

Information about presenting the parameters of random variables to the user is given in section 17.5.4.

14.2 `Client` performance measures

A number of performance measures are automatically calculated for each traffic pattern, unless the user decides to switch them off by selecting `SPF_off` when the traffic patterns is created (section 10.6.2). These measures consist of several random variables (type `RVariable`—section 14.1) and counters, which are updated automatically under certain circumstances.

14.2.1 Random variables

Type `Traffic` declares six pointers to random variables. These random variables are created together with the traffic pattern, provided that the standard performance measures are not switched off, and are used to keep track of the following measures (in parentheses we give the name of the corresponding `RVariable` pointer):

Absolute message delay (`RVAMD`) The absolute delay of a message m is the amount of time in ETUs (section 3.2) elapsing from the moment m was queued at the sender (this moment is indicated by the contents of the `Message` attribute `QTime`—see section 10.2) to the moment when the last packet of m has been received at its destination. A packet is assumed to have been received when `receive` (section 11.2.3) is executed for the packet.

From the viewpoint of this measure, each reception of a complete message (`receive` for its last packet) belonging to the given traffic pattern generates one sample.

Absolute packet delay (`RVAPD`) The absolute delay of a packet p is calculated as the amount of time in ETUs (section 3.2) elapsing from the moment the packet became ready for transmission (the message queuing time is excluded) until p is `received` at its destination. The time when a packet becomes ready for transmission is determined as the maximum of the following two values:

- the time when the buffer into which the packet is acquired was last `released` (section 10.3),
- the time when the message from which the packet is acquired was queued at the station (`Message` attribute `QTime`).

Note that it is illegal to acquire a packet into a full (non-released) buffer (section 10.8.1) and the above formula for determining the ready time of a packet is well defined.

Intuitively, as soon as a packet buffer is emptied (by `release`) the buffer becomes ready to accommodate the next packet. In this sense, the next packet becomes automatically ready for transmission, provided that the next packet is pending, i.e., a message is queued at the station. At first sight it might seem natural to assume that the packet delay should be measured from the moment the packet is put into the buffer. However, the actual operation of acquiring the packet into one of the station's buffers can be postponed by the protocol until the very moment of starting the packet's transmission. Thus, the numerical value of the packet delay would be dependent on the programming style of the protocol implementor. The `TTime` attribute of an empty packet buffer (see section 10.2) is used to store the time when the buffer was last `released`.

For this measure, each reception (by `receive`) of a complete packet belonging to the given traffic pattern generates one sample.

Weighted message delay (`RVWMD`) The weighted message delay (also called the message bit delay) is the delay in *ETU*s (section 3.2) of a single information bit measured from the time the message containing that bit was queued at the sender, to the moment when the packet containing that bit is completely received at the destination. Whenever a packet `p` belonging to the given traffic pattern is received, `p->ILength` samples are added to `RVWMD`, all with the same value equal to the difference between the current time (`Time`) and the time when the message containing the packet was queued at the sender. This difference, as all other delays, is expressed in *ETU*s.

Message access time (`RVMAT`) The access time of a message m is the amount of time in *ETU*s elapsing from the moment when the message was queued at the sender, to the moment when the last packet of the message is `released` by the sender. Each operation of releasing the last packet of a message generates a data sample for this measure.

Packet access time (`RVPAT`) The access time of a packet p is the amount of time in *ETU*s elapsing from the moment the packet becomes ready for transmission (see *absolute packet delay*) to the moment when the packet is `released` by the sender. Each operation of releasing a packet generates a data sample for this measure.

Message length statistics (`RVMLS`) Whenever a message of the given traffic pattern is queued at a sender, one data sample containing the message length is generated for this measure. Thus, the random variable pointed to by `RVMLS` collects statistics related to the length of messages generated according to the given traffic pattern.

If the standard performance measures have been switched off upon creation of a traffic pattern, the random variables listed above are not created and their pointers contain NULL.

14.2.2 Counters

Besides the random variables, type Traffic defines the following user-accessible counters:

```
Long NQMessages, NTMessages, NRMessages, NTPackets, NRPackets;
BITCOUNT NQBits, NTBits, NRBits;
```

All these counters are initialized to zero when the traffic pattern is created. If the standard performance measures are effective for the traffic pattern, the above counters are updated in the following way:

- Whenever a message is generated and queued at the sender (by the standard method genMSG—section 10.6.3), NQMessages is incremented by one and NQBits is incremented by the message length in bits.
- Whenever a packet is released, NTPackets is incremented by one. At the same time NTBits is incremented by the length of the packet's information part (attribute ILength) and NQBits is decremented by the same number. If the packet is the last packet of a message, NTMessages is incremented by one and NQMessages is decremented by one.
- Whenever a packet is received, NRPackets is incremented by one and NRBits is incremented by the length of the packet's information part. If the packet is the last packet of its message, NRMessages is incremented by one.

It is possible to print (or display) the standard performance measures associated with a given traffic pattern or the combination of performance measures for all traffic patterns viewed together as a single message arrival process. This part is described in section 17.5.5.

14.2.3 Virtual methods

To facilitate collecting non-standard statistics, type Traffic defines a number of virtual methods which are initially empty. These virtual methods can be redefined in a user extension of type Traffic. They are called automatically at certain events associated with traffic processing for the given traffic pattern. In the list below, we assume that mtype and ptype denote the message type and the packet type associated with the traffic pattern (section 10.6.1).

```
void pfmMQU (mtype *m);
```

The method is called whenever a message is generated (by the standard method `genMSG`—section 10.6.3) and queued at the sender. The argument points to the newly generated message.

```
void pfmPTR (ptype *p);
```

The method is called whenever a packet is `released` (section 10.3). The argument points to that packet.

```
void pfmPRC (ptype *p);
```

The method is called whenever a packet is `received` (section 11.2.3). The argument points to the received packet.

```
void pfmMTR (ptype *p);
```

The method is called whenever the last packet of a message is `released`. The argument points to that packet.

```
void pfmMRC (ptype *p);
```

The method is called whenever the last packet of a message is `received`. The argument points to the received packet.

The virtual methods listed above are called even if the standard performance measures for the given traffic pattern are switched off. If the standard performance measures are effective, the virtual methods supplement the standard functions.

Two more methods, formally belonging to the same class but not used for measuring performance, are described in section 10.6.4.

14.2.4 Resetting performance measures

At any moment, the performance statistics of a traffic pattern can be reset which corresponds to starting collecting these statistics from scratch. This is accomplished by the following `Traffic` method:

```
void resetSPF ();
```

The method does nothing for a traffic pattern created with `SPF_off` (section 10.6.2), i.e., if no performance statistics are gathered for the traffic pattern. Otherwise, it erases (section 14.1.2) the contents of the traffic pattern's random variables (section 14.2.1) and resets its counters (section 14.2.2) in the following way:

```
NTMessages = NTMessages - NRMessages;
NRMessages = 0;
NTPackets = NTPackets - NRPackets;
NRPackets = 0;
NTBits = NTBits - NRBits;
NRBits = 0;
```

The new values of the counters (note that `NQMessages` and `NQBits` are not changed) are obtained from the previous values by assuming that the number of received messages (packets, bits) is now zero. In order to maintain consistency, the counters reflecting the number of transmitted items are not zeroed, but decremented by the corresponding counts of received items. This way, after they have been reset, these counters give the number of items being "in transit." By the same token, the number of queued items is left intact.

Whenever `resetSPF` is executed for a traffic pattern, the traffic pattern's attribute `SMTime` of type `TIME` is set to the time when the operation was performed. Initially, `SMTime` is set to 0. This (user-accessible) attribute can be used to calculate properly the throughput for the traffic pattern. For example, the following formula:

```
thp = (double) NRBits / (Time - tp->SMTime);
```

gives the observed throughput[45] (in bits per *ITU*) for the traffic pattern `tp` measured from the last time the performance statistics for `tp` were reset. This method of `Traffic`:

```
double throughput ();
```

returns the same value normalized to the *ETU*, i.e., multiplied by `Etu` (the number of *ITU*s in one *ETU*—section 3.2). A similar method of the `Client` returns the global throughput over all traffic patterns expressed in bits per *ETU*. It doesn't crash when the measurement time is zero, but returns zero in such a case.

It is possible to reset the standard performance statistics globally for the entire `Client`, by executing the `resetSPF` method of the `Client`. This version of the method calls `resetSPF` for every traffic pattern and also resets the global counters of the `Client` keeping track of the number of items (messages, packets, bits) that have been generated, queued, transmitted, and received so far. These counters are reset in a similar way as the local user-accessible counters of an individual traffic pattern.

The global effective throughput calculated by the `Client` (sections 17.5.5, 17.5.12) is produced by relating the total number of received bits to the time during which these bits have been received. This time is calculated by subtracting from `Time` the contents of an internal variable (of type `TIME`) that gives the time of the last `resetSPF` operation performed for the `Client`.

[45] Assuming the denominator is nonzero.

If the simulation termination condition is based on the total number of messages received (section 15.1), this condition will be affected by the `Client` variant of `resetSPF`. Namely, `resetSPF` zeroes the global counter of received messages, so that it will be accumulating towards the limit from scratch. In particular, if `Client`'s `resetSPF` is executed many times, e.g., in a loop, the termination condition may never be met, although the actual number of messages received during the simulation run may be bigger than the specified limit. The semantics of the '`-c`' SIDE call option (sections 19.3, 15.1) are also affected by the global variant of `resetSPF`. Namely, the total number of generated messages is reduced by the number of messages that were received at the moment when the method was invoked.

Note: The global counters of the `Client` are not affected by calling the `Traffic` variant of `resetSPF`.

Every `resetSPF` operation executed on a traffic pattern triggers an event that can be perceived by a user-defined process. This way the user may define a customized response to the operation, e.g., resetting non-standard random variables and/or private counters. The event, labeled `RESET`, occurs on the traffic pattern, also if the traffic pattern was created with `SPF_off`. A global `resetSPF` operation performed on the `Client` forces the `RESET` event for all traffic patterns, in addition to triggering it on the `Client` *AI*.

14.3 `Link` performance measures

Similar to traffic patterns, certain standard performance statistics are automatically collected for links. Unlike the `Traffic` measures, the link statistics are not random variables: they are just counters (similar to those associated with traffic patterns—section 14.2.2) that keep track of how many bits, packets, and messages have passed through the link. The user may switch off collecting these measures at the moment when the link is created (see section 6.2.2).

The following publicly available `Link` attributes are related to measuring link performance:

```
char FlgSPF;
```

This attribute can take two values: `ON` (1), if the standard performance measures are to be calculated for the link, or `OFF` (0), if the standard measurements are switched off.

```
BITCOUNT NTBits, NRBits;
Long NTJams, NTAttempts;
Long NTPackets, NRPackets, NTMessages, NRMessages;
Long NDPackets, NHDPackets;
BITCOUNT NDBits;
```

All the above counters are initialized to zero when the link is created. Then, if the standard performance measures are effective for the link, the counters are updated in the following way:

- Whenever a jamming signal is inserted into a port connected to the link, `NTJams` is incremented by one.
- Whenever a packet transmission is started on a port connected to the link, `NTAttempts` is incremented by one.
- Whenever a packet transmission on a port connected to the link is terminated by `stop` (section 11.1.2), `NTPackets` is incremented by one and `NTBits` is incremented by the packet's payload length (attribute `ILength`). If the packet is the last packet of its message, `NTMessages` is incremented by one.
- Whenever a packet is `received` from the link (section 11.2.3), `NRPackets` is incremented by one and `NRBits` is incremented by the packet's payload length. If the packet is the last packet of its message, `NRMessages` is incremented by one. **Note:** If `receive` is called without the second argument (identifying the link—section 11.2.3), the counters are not incremented.
- Whenever a *damaged* packet (section 11.4) is inserted into the link, `NDPackets` is incremented by one and the packet's payload length (attribute `ILength`) is added to `NDBits`. If the packet happens to be header-damaged, `NHDPackets` is also incremented by one. Note that a header-damaged packet is also "damaged"; thus, `NHDPackets` cannot be bigger than `NDPackets`.

Type `Link` declares a number of virtual methods which are initially empty and can be redefined in a user extension of a standard link type. These methods, which can be used to collect non standard performance statistics related to the link, are listed below.

```
void pfmPTR (Packet *p);
```

The method is called when a packet transmission on a port belonging to the link is terminated by `stop`. The argument points to the terminated packet.

```
void pfmPAB (Packet *p);
```

The method is called when a packet transmission on a port belonging to the link is terminated by `abort`. The argument points to the aborted packet.

```
void pfmPRC (Packet *p);
```

The method is called when a packet is `received` from the link (see above). The argument points to the received packet.

```
void pfmMTR (Packet *p);
```

The method is called when a packet transmission on a port belonging to the link is terminated by `stop` and the packet turns out to be the last packet of its message. The argument points to the terminated packet.

```
void pfmMRC (Packet *p);
```

The method is called when the last packet of its message is `received` from the link. The argument points to the received packet.

```
void pfmPDM (packet *p);
```

The method is called when a damaged packet is inserted into the link. The user can examine the packet flags (section 11.4) to determine the nature of the damage.

It is possible to print out or display the standard performance measures related to a link. This part is discussed in section 17.5.8.

Note: The counters and methods related to damaged packets are only available if the SIDE program has been created with the `-z` option of `mks` (section 19.2).

14.4 `RFChannel` performance measures

The set of performance-related counters defined by a radio channel closely resembles that for `Link` (section 14.3), and is in fact its subset. Similar to a link, those counters are updated automatically, unless the user decided to switch them off when the radio channel was created (section 6.4.2).

The following publicly available attributes of `RFChannel` are related to measuring performance:

```
char FlgSPF;
```

This attribute can take two values: `ON` (1), if the standard performance measures are to be calculated for the radio channel, or `OFF` (0), if the standard measurements are switched off.

```
BITCOUNT NTBits, NRBits;
Long NTAttempts, NTPackets, NRPackets, NTMessages, NRMessages;
```

All the above counters are initialized to zero when the radio channel is created. Then, if the standard performance measures are effective for the channel, the counters are updated in the following way:

- Whenever a packet transmission is started on a transceiver interfaced to the radio channel, `NTAttempts` is incremented by one.
- Whenever a packet transmission on a transceiver interfaced to the channel is terminated by `stop` (section 12.1.8), `NTPackets` is incremented by one and `NTBits` is incremented by the packet's payload length (attribute `ILength`). If the packet is the last packet of its message, `NTMessages` is incremented by one.
- Whenever a packet is `received` from the radio channel (section 12.2.3), `NRPackets` is incremented by one and `NRBits` is incremented by the packet's payload length. If the packet is the last packet of its message, `NRMessages` is incremented by one. **Note:** If `receive` is called without the second argument (identifying the radio channel—section 12.2.3), the counters are not incremented.

Similar to `Link`, type `RFChannel` declares a number of virtual methods which are initially empty and can be redefined in a user extension of the class. These methods, which can be used to collect non standard performance statistics related to the channel, are listed below.

```
void pfmPTR (Packet *p);
```

The method is called when a packet transmission on a transceiver interfaced to the channel is terminated by `stop`. The argument points to the terminated packet.

```
void pfmPAB (Packet *p);
```

The method is called when a packet transmission on a transceiver interfaced to the channel is terminated by `abort`. The argument points to the aborted packet.

```
void pfmPRC (Packet *p);
```

The method is called when a packet is `received` from the channel (see above). The argument points to the received packet.

```
void pfmMTR (Packet *p);
```

The method is called when a packet transmission on a transceiver interfaced to the channel is terminated by `stop` and the packet turns out to be the last packet of its message. The argument points to the terminated packet.

```
void pfmMRC (Packet *p);
```

The method is called when the last packet of its message is `received` from the channel. The argument points to the received packet.

It is possible to print out or display the standard performance measures related to a radio channel. This part is discussed in section 17.5.10.

15 Terminating execution

The execution of a SIDE program can be terminated explicitly upon request from the program (section 7.9). It can also hit an error condition, be aborted by the user, or run out of events.[46] The last situation occurs when all processes (including system processes) get into a state where no possible event can wake them up, which generally does not happen if the standard `Client` is enabled. Note that such a situation can be intercepted by a process (or multiple processes) waiting for the `STALL` event on the `Kernel` (section 7.9). In such a case, running out of events does not trigger termination.

The program is also terminated automatically when any of three limit conditions specified by the user has been met. The following (global) functions can be used to declare the limits:

```
setLimit (Long MaxNM, TIME MaxTime, double MaxCPUTime);
setLimit (Long MaxNM, TIME MaxTime);
setLimit (Long MaxNM);
```

If `setLimit` is not used, the run will continue indefinitely, until it is terminated explicitly by the protocol program. The primary purpose of the function is to define hard exit conditions for simulation experiments. Control programs driving real physical systems usually have no exit conditions and they don't use `setLimit`.

Although typically `setLimit` is called from `Root` before the simulation is started (section 7.9), it can be called at any moment during the protocol execution. If the parameters of such a call specify limits that have been already reached or exceeded, the experiment is terminated immediately.

15.1 Maximum number of received messages

The first argument of `setLimit` specifies the maximum number of messages that are to be entirely received (their *last packets* `received`—see sections 11.2.3, 14.3) at their

[46] A control program running in the real mode (sections 1.5, 8) is not terminated when it runs out of events. It just sits there waiting for the impossible to happen. Such a program can be aborted by the user or terminated gracefully from DSD (section 18.6.1).

destinations. The simulation experiment will be terminated as soon as the message number limit is reached.

When the '`-c`' SIDE call option is used (section 19.3), the message number limit is interpreted as the maximum number of messages to be generated by the `Client` and queued at senders. In this case, when the limit is reached, the `Client` stops and the simulation continues until all the messages that remain queued have been `received` at their destinations.

There is a standard function,

```
void setFlush ();
```

which, if called before the protocol execution is started (i.e., while the `Root` process is in its initial state—section 7.9), has the same effect as if the program has been invoked with the '`-c`' option.

15.2 Virtual time limit

The second argument of `setLimit` declares the maximum interval of the modeled (virtual) time in *ITU*s. The execution will stop as soon as the modeled time (the contents of variable `Time`) has reached the limit.

Note: If the virtual time limit has been reset to the end time of the tracing interval with `-t` (see section 19.3), `setLimit` can only reduce that limit. Any attempts to extend it will be quietly ignored.

15.3 CPU time limit

The last argument of `setLimit` declares the limit on the CPU time used by the program. Owing to the fact that SIDE checks against violation of this limit every 5000 events, the CPU time limit may be slightly exceeded before the experiment is eventually stopped.

If the specified value of any of the above limits is 0, it actually stands for "no limit," i.e., any previous setting of this limit is canceled. If no value is given for a limit (the last two versions of `setLimit`), the previous setting of the limit is retained.

All three limits are viewed as an alternative of exit conditions, i.e., the execution stops as soon as **any** of the limits is reached. In particular, the process of emptying message queues, when SIDE was called with the '`-c`' option and the message number limit has been reached, can be stopped prematurely, if one of the other two limits is hit in the meantime.

15.4 Exit code

The protocol program, specifically the `Root` process (section 7.9), can learn why it has been terminated. When the `DEATH` event for the `Kernel` process is triggered, the integer environment variable `TheExitCode` (an alias for `Info01`) contains a number that tells what has happened. The following values can be returned via this variable:

`EXIT_msglimit`
: message number limit has been reached

`EXIT_stimelimit`
: simulated time has reached the declared limit

`EXIT_rtimelimit`
: CPU time limit has been reached

`EXIT_noevents`
: there are no more events to process

`EXIT_user`
: the protocol program has terminated the `Kernel` process (section 7.9)

`EXIT_abort`
: the program has been aborted due to an error condition

The actual numerical values assigned to the symbolic constants above are 0–5, in the order in which the constants are listed.

16 Tools for testing and debugging

SIDE is equipped with a few tools that can assist you in the process of debugging and validating a protocol program, especially in those numerous cases when formal correctness proofs are infeasible.

The incorrectness of a protocol or a control program can manifest itself in one of the following two ways:

- under certain circumstances, the protocol crashes and ceases to operate
- the protocol seems to work, but it does not exhibit certain properties expected by the designer and/or implementor

Generally, problems of the first type are rather easy to detect by extensive testing, unless the "certain circumstances" occur too seldom to be caught. Our experience indicates that the likelihood of finding a crashing error decreases much more than linearly with the running time of the protocol prototype. However, the specific nature of protocol programs, especially medium access control protocols, makes them susceptible for errors of the second type, which may be more difficult to diagnose.

16.1 User-level tracing

One simple way to detect run time inconsistencies is to assert Boolean properties involving variables in the program. SIDE offers some tools for this purpose (see section 4.6). Once an error has been found, the detection of its origin may still pose a tricky problem. Often, one has to trace the protocol behavior for some time prior to the occurrence of the error to identify the circumstances leading to the trouble.

The following function can be used to write to the output file a line describing the contents of some variables at the moment of its call:

```
void trace (const char*, ...);
```

It accepts a format string as the first argument, which is used to interpret the remaining arguments (in the same way as in `printf` or `form`). The printout is automatically terminated by a new line, so the format string shouldn't end with one. A line produced by `trace` is preceded by a header, whose default form consists of the modeled time in *ITU*. For example, this command:

```
trace ("Here is the value: %1d", n);
```

may produce a line looking like this:

```
Time: 1184989002 Here is the value: 93
```

The most useful feature of `trace`, which gives it an advantage over standard output tools, is that it can be easily disabled or restricted to a narrow interval of modeled time by a program call argument. This is important because tracing problems that manifest themselves after lengthy execution may generate tonnes of intimidating output. The `-t` program call argument (described in section 19.3) can be used to disable or restrict the output of `trace`. It is also possible to confine that output to the context of a few specific stations. By default, i.e., without `-t`, every `trace` command is effective.

The program can also affect the behavior of `trace` dynamically, by calling the following global functions:

```
void settraceFlags (FLAGS opt);
void settraceTime (TIME begin, TIME end = TIME_inf);
void settraceTime (double begin, double end = Time_inf);
void settraceStation (Long s);
```

The first argument of `settraceFlags` is a collection of additive flags that describe the optional components of the header preceding a line written by `trace`. If `opt` is zero, `trace` is unconditionally disabled, i.e., its subsequent invocations in the program will produce no output until a next call to `settraceFlags` selects some actual header components. Here are the options (represented by symbolic constants):

`TRACE_OPTION_TIME`
: This option selects the standard header, i.e., current simulation time in *ITU*s. It is the only option in effect by default.

`TRACE_OPTION_ETIME`
: Includes the simulated time presented in *ETU*s—as a floating point number.

`TRACE_OPTION_STATID`
: Includes the numerical `Id` of the current station (section 6.1.3).

`TRACE_OPTION_PROCESS`
: Includes the standard name of the current process (section 5.2).

`TRACE_OPTION_STATE`
: Includes the symbolic name of the state in which the current process has been awakened. This option is independent of the previous option.

For example, having executed:

```
settraceFlags (TRACE_OPTION_ETIME + TRACE_OPTION_PROCESS);
```

followed by:

```
trace ("Here is the value: %1d, n);
```

you may see something like this:

```
Time: 13.554726 /Transmitter/ Here is the value: 93
```

Note that the default component of the `trace` header, represented by `TRACE_OPTION_TIME`, will be removed if the argument of `settraceFlags` does not include this value. However, there is no way to produce a `trace` line without a single header component because `settraceFlags` (0) completely disables `trace` output.

With the two variants of `settraceTime` you can confine the output of `trace` to a specific time interval. For the first variant, that interval is described as a pair of `TIME` values in *ITU*s, while the second variant accepts `double` *ETU* values. If the second value is absent, the end of the tracing period is undetermined, i.e., `trace` remains active until the end of run (see section 3.6 for the meaning of constants `TIME_inf` and `Time_inf`). The specified time interval is interpreted in such a way that `begin` describes the first moment of time at which `trace` is to be activated, while `end` (if not infinite) points to the first instant at which `trace` becomes inactive. This means that `trace` invoked exactly at time `end` will produce no output. Every call to `settraceTime` (any variant) overrides all previous calls. The only way to define multiple tracing intervals is to issue a new `settraceTime` call for the next interval exactly at the end of the previous one.

Tracing can be restriced in space as well as in time. The argument of `settraceStation` specifies the station to which the output of `trace` should be confined. This means that the environment variable `TheStation` (section 6.1.3) must point to the indicated station at the moment `trace` is invoked. Multiple calls to the function add stations to the tracing pool. This way it is possible to trace an arbitrary subset of the network. By calling `settraceStation` with `ANY` as the argument, you revert to the default case of tracing all stations.

Note: The `-t` program call argument (section 19.3) is interpreted after the network has been built (section 7.9) and thus overrides any `settraceTime` and `settraceStation` calls made from the first state of `Root`. This postponement in the processing of `-t` is required because the interpretation of an *ETU* value (as a time bound) can only be meaningful after the *ETU* has been defined by the program (section 3.2). The `-t` argument, if it appears on the program call line, effectively cancels the effect of all `settraceTime` and `settraceStation` calls issued in the first state of `Root`. However, any calls to those functions issued later on (after the network has been built) will override the effects of `-t`.

By the same token, a sensible `settraceTime` call with *ETU* arguments can only be made after the *ETU* has been set. Otherwise, the function will transform the specified values into `TIME` using the default setting of 1 $ITU = 1\ ETU$.

You can easily take advantage of the restriction mechanism for `trace` in your own debugging operations. The global macro `Tracing` with the properties of a `Boolean` variable is `YES`, if `trace` would be active in the present context, and `NO` otherwise.

16.2 Simulator-level tracing

In addition to explicit tracing described in section 16.1, SIDE has a built-in mechanism for dumping the sequence of process states traversed by the simulator during its execution to the output file. This feature is only available when the program has been compiled with `-g` or `-G` (see section 19.2).

One way of switching on the state dumping is to use the `-T` program call argument (see section 19.3). Its role is similar to `-t` (which controls the user-level tracing—section 16.1), except that, in the present case, the tracing is disabled by default. The same effect can be accomplished dynamically from the program, by invoking functions from this list:

```
void setTraceFlags (Boolean full);
void setTraceTime (TIME begin, TIME end = TIME_inf);
void setTraceTime (double begin, double end = Time_inf);
void setTraceStation (Long s);
```

For the first function, if `full` is `YES`, it selects the "full" version of state dumps, as described below. The remaining functions restrict the tracing context in the same way as for the corresponding user-level functions discussed in section 16.1.

Any episode of a process being awakened that falls into the confines of the tracing parameters (in terms of time and station context) results in one line being written to the output file. That line is produced before the awakened process gains control and includes the following items:

- the current simulated time
- the *type name* and the `Id` of the *AI* generating the event
- the event identifier (in the *AI*-specific format, see section 17.5.1)
- the station `Id`
- the output name of the process (section 5.2)
- the identifier of the state to be assumed by the process

With "full" tracing, that line is followed by a dump of all links and radio channels accessible via ports and transceivers from the current station. This dump is obtained by requesting mode 2 exposures of `Link` (section 17.5.8) and `RFChannel` (section 17.5.10).

The global macro `DebugTracing` has the appearance of a `Boolean` variable whose value is `YES` if 1) the program has been compiled with `-g` or `-G`, i.e., debugging is enabled, and 2) state dumping is active in the current context. Otherwise, `DebugTracing` evaluates to `NO`.

Note: normally, the waking episodes of SIDE internal processes are not dumped with simulator-level tracing (they are seldom interesting to the user). To include them alongside the protocol processes, use the `-s` program call option (section 19.2).

16.3 Observers

Observers are tools for expressing global assertions that involve combined actions of many processes. An observer is a dynamic object that resembles a regular protocol process. However, observers never respond directly to events perceived by regular processes nor do they generate events that may be perceived by regular processes. Instead, they react to the so-called *meta-events*. By a meta-event we understand the operation of awakening a regular process. An observer may declare that it wants to be awakened whenever a specific regular process is restarted at a specific state. This way, the observer is able to monitor the behavior of the protocol viewed as a collection of finite-state machines.

An observer is an object belonging to an observer type. An observer type is defined in the following way:

```
observer otype : itypename {
    ...
    local attributes and methods
    ...
    states { list of states };
    ...
    perform {
        the observer code
    };
};
```

where *otype* is the name of the declared observer type and *itypename* identifies an already known observer type (or types), as described in section 5.3.

The above definition closely resembles a process type declaration (section 7.2). One difference is that the structure of observers is flat, i.e., an observer does not belong to any specific station, has no formal parents and no children. Observer types are extensible and they can be derived from other user-defined observer types.

A `setup` method can be declared in the standard way (see section 5.8)—to initialize the local attributes when an observer instance is created. The default observer `setup` method is empty and takes no arguments. Observers are created by `create`—in the regular way.

The observer's code method has the same layout as a process code method, i.e.,

```
perform {
    state OS_0:
```

```
            ...
        state OS_1:
            ...
        state OS_{p-1}:
            ...
    };
```

where $OS_0, \ldots, OS_{p-1}$ are state identifiers listed with the `states` statement within the observer type declaration.

An awakened observer, similar to a process, performs some operations, specifies its new waking conditions, and goes to sleep. The waking conditions for an observer are described by executing the following operations:

```
inspect (s, p, n, ps, os);
timeout (t, os);
```

Operation `inspect` identifies a class of scenarios when a regular process is restarted. The arguments of `inspect` have the following meaning:

`s`
: identifies the station to which the process belongs; it can be either a station object pointer or a station `Id` (an integer number)

`p`
: is the process type identifier

`n`
: is the character string containing the process *nickname* (section 5.2)

`ps`
: is the process state identifier

`os`
: identifies the observer's state at which the observer wants to be awakened

For simplicity, let us assume that an observer has issued exactly one inspect request and put itself to sleep. The inspect request is interpreted as a declaration that the observer wants to remain suspended until SIDE wakes up a process matching the `inspect` parameters. Then, immediately **after** the process completes its action, the observer will be run and the

global variable `TheObserverState` will contain the value passed to `inspect` through `os`. This value will be used to select the proper state within the observer's `perform` method.

If any of the first four arguments of `inspect` is `ANY`, it means that the actual value of that attribute in the process awakening scenario is irrelevant. For example, with the request

```
inspect (ANY, transmitter, ANY, ANY, WakeMeUp);
```

the observer will be restarted after any process of type `transmitter` is awakened at any state.

Note that if the process type is insufficient to identify the process (e.g., the observer wants to monitor the operation of a specific single process), the process must be assigned a unique *nickname* (section 5.2). This nickname (if it is unique among processes of the given type) can be used to identify exactly one process. It is also possible to identify a subclass of processes within a given type (or even across different types) by assigning the same nickname to several processes.

The process state identifier can only be specified, if the process type identifier has been provided as well (i.e., it is not `ANY`). A state is always defined within the scope of a specific process type and specifying a state without a process type makes no sense.

There exist abbreviated versions of `inspect` accepting 1, 2, 3 and 4 arguments. In all these versions, the last argument specifies the observer's state (corresponding to `os` in the full five-argument version) and all the missing arguments are assumed to be `ANY`. Thus, the single-argument version of `inspect` says that the observer is to be restarted after any event awaking any protocol process at any station. In the two-argument version, the first argument identifies the station to which the awakened process should belong (`s`). The three-argument version accepts the station identifier as the first argument and the process type identifier (`p`) as the second argument. For the four-argument version, the first two arguments are the same as in the two-argument case, and the third argument identifies the process state (`ps`).

When an observer is created (by `create`), it is initially started in the first state occurring on its `states` list—in a way similar to a process. The analogy between observers and processes extends onto inspect requests which are somewhat similar to wait requests. In particular, an observer can issue a number of inspect requests before it puts itself to sleep. Like a process, an observer puts itself to sleep by exhausting the list of commands at its current state or by executing `sleep`.

Unlike for multiple wait requests issued by the same process, the order in which multiple inspect requests are issued is significant. The pending inspect requests are examined in the order in which they have been issued, and the first one that matches the current process wakeup scenario is used. Then the observer is awakened in the state determined by the value of `ms` specified with that inspect.

Example

Assume that an observer has issued the following sequence of inspect requests:

```
inspect (ThePacket->Receiver, receiver, ANY, Rcv, State1);
inspect (TheStation, ANY, ANY, ANY, State2);
inspect (ANY, ANY, ANY, ANY, State3);
```

The observer will wake up at `State1`, if the next process restarted by SIDE

- is of type `receiver`
- belongs to the station determined by the `Receiver` attribute of `ThePacket`
- wakes up in state `Rcv`

The three conditions must hold all at the same time. Otherwise, if the restarted process belongs to the current station, the observer will be awakened at `State2`. Finally, if the process wakeup scenario does not match the arguments of the first two `inspects`, the observer will be awakened in `State3`.

The three requests can be rewritten using the abbreviated versions of `inspect`, in the following way:

```
inspect (ThePacket->Receiver, receiver, Rcv, State1);
inspect (TheStation, State2);
inspect (State3);
```

Of course, a restarted observer has access to the environment variables of the process that has been awakened. These variables reflect the configuration of the process environment **after** the process completes its action in the current state.

Being in one state, an observer can branch directly to another state by executing

```
proceed nstate;
```

where `nstate` identifies the new state. The semantics of this operation is different from that of `proceed` for a process (section 8.1). Namely, the observer variant of `proceed` branches to the indicated state immediately, and absolutely nothing else can happen in the meantime.

Whenever an observer is restarted, its entire inspect list is cleared, so that new waiting conditions have to be specified from scratch. This also applies to the `timeout` operation (discussed below), which is used by observers to implement alarm clocks.

By executing `timeout (t, ms)` an observer sets up an alarm that will wake it up unconditionally `t` *ITU*s after the current moment, if no process matching one of the pending inspect requests is restarted in the meantime.

Note: Observer clocks are always accurate, i.e., the `timeout` interval is always precisely equal to the specified number of *ITU*s and is never subject to randomization (section 8.3).

An arbitrary number of observers can be defined and active at any moment. Different observers are completely independent, in the sense that the behavior of a given observer is not affected by the presence or absence of other observers. An observer can terminate itself in exactly the same way as a process, i.e., by executing `terminate ()` (section 7.3) or by going to sleep without issuing a single `inspect` or `timeout` request.

When the protocol program is built with the '`-v`' option of `mks` (see section 19.2) all observers are deactivated: all `create` operations for observers are then void.

17 Exposing objects

By *exposing* an object we mean either printing out some information related to the object or displaying this information on the terminal screen. In the former case, the exposing is done exclusively by SIDE; in the latter case, the exposing is performed by a separate display program (possibly running on a different host) communicating with the SIDE program via IPC tools.

17.1 General concepts

Each *Object* type carrying some dynamic information that may be of interest to the user can be made *exposable*. An exposable type defines a special method that describes how the information related to the objects of this type should be printed out and/or how it should be displayed on the terminal screen. By "printing out" information related to an object we mean including this information in the results file (section 4.2), By "displaying" information on the terminal screen we understand sending this information to DSD—a special display program (section 18) that organizes it into a collection of windows presented on the screen. In the later case, the information is displayed dynamically, in the sense that it is updated periodically and in any moment reflects a snapshot situation in the middle of execution of the protocol program. Printing out, in the sense of the above definition, will also be called exposing "on paper," whereas displaying will be called exposing "on screen." The property of an exposure that says whether the information is to be "printed" or "displayed" is called the *exposure form*.

The way an object is exposed is described by the `exposure` declaration associated with the object type. This declaration specifies the code to be executed when the object is exposed.

An object can be exposed in a number of ways, irrespective of the form (i.e., whether the exposure is "on paper" or "on screen"). Each such a way is called an exposure *mode* and is identified by a (typically small) nonnegative integer number. Different exposure modes can be viewed as different fragments (or types) of information associated with the object.

Moreover, some exposure modes may be optionally *station-relative*. In such a case, besides the mode, the user may specify a station to which the information printed (or displayed) is to be related. If no such station is specified, the global variant of the exposure mode is used.

For most of the standard *Object* types, the display modes coincide with the printing modes, so that, for a given mode, the same information is sent to the "paper" and to the "screen." However, each mode for any of the two exposure forms is defined separately. For example, the standard exposure of the `Client` defines the following four "paper" modes:

0
: Information about all processes that have pending wait requests to the `Client`. If a station-relative variant of this mode is chosen, the information is restricted to the processes belonging to the given station.

1
: Global performance measures taken over all traffic patterns combined. This mode cannot be made station-relative.

2
: Message queues at all stations, or at one specified station (for the station-relative variant of this mode).

3
: Traffic pattern definitions. No station-relative variant of this mode exists.

The first three modes are also applicable for exposing the `Client` "on screen" (and they display the same information as their "paper" counterparts), but the last mode is not available for the "screen" exposure.

In general, the format of the "paper" exposure need not be similar to the format of the corresponding "screen" exposure. The number of modes defined for the "paper" exposure need not be equal to the number of the "screen" exposure modes.

17.2 Making objects exposable

In principle, objects of any *Object* type (section 5.1) can be *exposed*, but they must be made *exposable* first. All standard *Object* types are made exposable automatically. The

user may declare a non-standard subtype of *Object* as exposable and describe how objects of this type are to be exposed. It is also possible to declare a non-standard exposure for an extension of an exposable standard type. This non-standard exposure can either replace or supplement the standard one.

The remainder of this section and the entire next section may be irrelevant for the user who is not interested in creating non-standard exposable types. However, besides instructing on how to expose objects of non-standard types, this section also explains the mechanism of exposing objects of the standard types. This information may be helpful in understanding the operation of the display program described in section 18.

All user-created subtypes of *Object* must belong to type `EObject` (see sections 5.1, 5.8). To make such a subtype exposable, you have to declare an *exposure* for this type. The exposure declaration should appear within the type (class) definition. It resembles the declaration of a regular method and has the following general format:

```
exposure {
    onpaper {
        exmode mp_0:
            ...
        exmode mp_1:
            ...
        ...
        exmode mp_{k-1}:
            ...
    };
    onscreen {
        exmode ms_0:
            ...
        exmode ms_1:
            ...
        ...
        exmode ms_{n-1}:
            ...
    };
};
```

As for a regular method, it is possible to merely announce the exposure within the definition of an *Object* type to be exposed and specify it later. An *Object* type is announced as exposable by putting the keyword `exposure;` into the list of its publicly visible attributes.

Example

The following declaration (see section 5.1) defines an exposable non-standard *Object* type:

```
eobject MyStat {
    Long NSamples;
    RVariable *v1, *v2;
    void setup () {
        v1 = create RVariable;
        v2 = create RVariable;
    };
    exposure;
};
```

The exposure definition, similar to the specification of a method announced in a class declaration, must appear below the type definition in one of the program's files, e.g.,

```
MyStat::exposure {
    onpaper {
        exmode 0: v1->printCnt ();
        exmode 1: v2->printCnt ();
    }
    onscreen {
        exmode 0: v1->displayOut (0);
        exmode 1: v2->displayOut (1);
    }
}
```

The meaning of particular statements from the exposure code is explained further in the present section.

An exposure specification consists of two parts: the "paper" part and the "screen" part. Any of the two parts can be omitted; in such a case the corresponding exposure form is undefined and the type cannot be exposed that way.

Each fragment starting with `exmode` *m*`:` and ending at the next `exmode`, or at the closing brace of its form part, contains code to be executed when the exposure with mode *m* is requested for the given form.

The exposure code has immediate access to the attributes of the exposed object: it is effectively a method declared within the exposed type. Additionally, the following two variables are accessible from this code (they can be viewed as arguments passed to the `exposure` method):

```
Long SId;
char *Hdr;
```

If `SId` is not `NONE` (–1) it gives the `Id` of the station to which the exposed information is to be related. It is up to the exposure code to interpret this value, or ignore it (e.g., if it makes no sense to relate the exposed information to a specific station).

Variable `Hdr` is only relevant for a "paper" exposure. It points to a character string representing the header to be printed along with the exposed information. If `Hdr` contains `NULL` it means that no specific header is requested. Again, the exposure code must perform an explicit action to print out the header; in particular, it may ignore the contents of `Hdr`, print a default header if `Hdr` contains `NULL`, etc. The contents of `Hdr` for a "screen" exposure are irrelevant and they should be ignored.

Any types, variables, and objects needed locally by the exposure code can be declared immediately after the opening `exposure` statement.

If an exposure is defined for a subtype of an already exposable type, it overrides the supertype's exposure. The user may wish to augment the standard exposure by the new definition. In such a case, the new definition may include a reference to the supertype's exposure in the form

```
supertypename::expose;
```

Such a reference is equivalent to invoking the supertype's exposure in the same context as the subtype's exposure. The most natural place where the supertype exposure can be referenced is at the very beginning of the subtype exposure declaration, e.g.,

```
exposure {
    int MyAttr;
    SuperType::expose;
    onpaper {
        ...
    };
    onscreen {
        ...
    };
};
```

If the supertype's exposure does not contain the mode for which the subtype's exposure has been called, the reference to the supertype's exposure has no effect. Thus, the subtype exposure can just add new modes—ones that are not serviced by the supertype exposure.

17.3 Programming exposures

The "paper" part of the exposure body should contain statements that output the requested information to the *results* file (section 4.2). The use of the `print` function (section 4.2.1) is recommended.

The issues related to programming the "screen" part are somewhat more involved, although, in most cases this part is shorter and simpler than the "paper" part. The main difference between the two exposure forms is that while the "paper" exposure can be interpreted as a regular function that is called explicitly by the user program to write some information to the output file, the "screen" exposure is called many times to refresh information displayed on the terminal screen—in an asynchronous way from the viewpoint of the SIDE program. One simplification with respect to the "paper" case is that the exposure code need not be concerned with formatting the output: it just sends out "raw" data items which are interpreted and organized by the display program.

Below we list the basic functions that can be accessed by "screen" exposures.

```
void display (LONG ii);
void display (double dd);
void display (BIG bb);
void display (const char *tt);
void display (char c);
```

Each of the above four functions sends one data item to the display program. For the first three functions, the data item is a numerical value (integer, double, or `BIG`–also `TIME`—section 3.3). The data item sent by the fourth method is a character string terminated by a null byte. Finally, the last method sends a simple character string consisting of a single character (e.g., `display ('a');` is equivalent to `display ("a");`).

A simple data item passed to the display program by one of the above functions is converted by that program into a character string and displayed within a window (section 18). It is possible to send to the display program graphic information in the form of a collection of curves to be displayed within a rectangular area of a window. Such an area is called a *region*. The following functions:

```
void startRegion (double xo, double xs, double yo, double ys);
void startRegion ();
```

are used to start a region, i.e., to indicate that the forthcoming display operations (see below) will build the region contents.

The first function starts the so-called *scaled region*. The arguments have the following meaning:

`xo`
: The starting value for the x co-ordinate, i.e., the x co-ordinate of the left side of the region's rectangle.

`xs`
: The terminating value for the x co-ordinate, i.e., the x co-ordinate of the right side of the region's rectangle.

`yo`
: The starting value for the y co-ordinate, i.e., the y co-ordinate of the bottom side of the region's rectangle.

`xs`
: The terminating value for the y co-ordinate, i.e., the y co-ordinate of the top side of the region's rectangle.

Calling `startRegion ()` is equivalent to calling

```
startRegion (0.0, 1.0, 0.0, 1.0);
```

i.e., by default, the region comprises a unit square at the beginning of the coordinate system.

A region is terminated by calling

```
endRegion ();
```

which informs the display program that nothing more will be put into the region.

Display statements executed between `startRegion` and `endRegion` send to the display program information describing a number of *segments*. A segment can be viewed as a single curve delineated by a collection of points. The following function can be used to start a segment:

```
startSegment (Long att = NONE);
```

where the argument, if specified, is a bit pattern defining segment attributes (described in section 18.6.3). If the attribute argument is not specified (or if its value is `NONE`), default attributes are used.

A segment is terminated by calling

```
endSegment ();
```

The contents of a segment are described by calling the following function:

```
void displayPoint (double x, double y);
```

which gives the co-ordinates of one point of the segment. If (based on the segment attribute pattern) the segment points are to be connected, a line will be drawn between each pair of points consecutively displayed by `displayPoint`. The first and the last points are not connected, unless they are the only points of the segment.

It is possible to display an entire segment with a single function call, but all the point co-ordinates must be prepared in advance in two arrays. One of the following two functions can be used to do the job:

```
void displaySegment (Long att, int np, double *x, double *y);
void displaySegment (int np, double *x, double *y);
```

The first argument of the first function specifies the segment attributes; default attributes (see above) are used for the second function. The two `double` arrays are expected to contain x and y co-ordinates of the segment points. The size of these arrays is determined by `np`.

It is also possible to display an entire region with a single statement, using the following variants of the `display` function introduced above:

```
void display (double xo, double xs, double yo, double ys,
                        Long att, int np, double *x, double *y);
void display (Long att, int np, double *x, double *y);
void display (int np, double *x, double *y);
```

Such a region must consist of exactly one segment (which is not uncommon for a region). The first variant specifies the scaling parameters (the first four arguments), the attribute pattern, the number of points, and the point arrays. The second variant uses default scaling, and the last one assumes default scaling and default segment attributes.

Examples

Consider the following declaration of a non-standard exposable type:

```
eobject MyEType {
    TIME  WhenCreated;
    int NItems;
    void setup () { WhenCreated = Time; NItems = 0; };
    exposure;
};
```

and the following simple exposure definition for objects of this type:

```
MyEType::exposure {
    onpaper {
        exmode 0:
            if (Hdr)
                Ouf << Hdr << '\n';
            else
                Ouf << "Exposing " << getSName () << '\n';
            print (WhenCreated, "Created at");
            print (NItems, "Items =");
    };
    onscreen {
        exmode 0:
            display (WhenCreated);
            display (NItems);
    };
};
```

Only one exposure mode is defined for both forms. The data items sent to the display program are the same as those printed out with the "paper" exposure.

For another exercise, suppose that we want to have an exposable object representing a number of random variables. The only exposure mode of this object will display the mean values of those variables in a single-segment region. This is how the type of our object can be defined:

```
eobject RVarList {
  int VarCount, MaxVars;
  RVariable **RVars;
  void setup (int max) {
    VarCount = 0;
    RVars = new RVariable* [MaxVars = max];
  };
  void add (RVariable *r) {
    Assert (VarCount < MaxVars, "Too many random variables");
    RVars [VarCount++] = r;
  };
  exposure;
};
```

The setup method of `RVarList` initializes the random variable list as empty. New random variables can be added to the list with the `add` method. This is how their mean values

can be exposed in a region:

```
RVarList::exposure {
  double min, max, mean, Min, Max;
  int i;
  LONG count;
  onscreen {
    exmode 0:
      if (VarCount == 0) return;
      Min = HUGE; Max = -HUGE;
      for (i = 0; i < VarCount; i++) {
        RVars [i] -> calculate (&min, &max, &mean, &count);
        if (min < Min) Min = min;
        if (max > Max) Max = max;
      }
      startRegion (0.0, (double)(VarCount-1), Min, Max);
      startSegment (0xc0);
      for (i = 0; i < VarCount; i++) {
        RVars [i] -> calculate (&min, &max, &mean, &count);
        displayPoint ((double)i, mean);
      }
      endSegment ();
      endRegion ();
  }
};
```

The first loop calculates the scaling parameters for the vertical coordinate: the y-range is set to the difference between the minimum and maximum value assumed by any variable in the list. The second loop displays the mean values of all variables. Of course, we could avoid re-calculating the parameters of the random variables in the second loop by saving the mean values calculated by the first loop in a temporary array.

Sometimes, in the "screen" part of an exposure, one would like to tell when the exposure has been called for the first time. In the above example, we could use a temporary array for storing the mean values calculated by the first loop, so that we wouldn't have to recalculate them in the second. It would make sense to allocate this array once, when the exposure is called for the first time, possibly deallocating it at the last invocation.

Generally, a "screen" exposure code may need some dynamically allocatable data structure. Technically, it is possible to allocate such a structure within the object to be exposed (e.g., by the `setup` method), but methodologically, it is better to do it in the right place, i.e., within the exposure code. This way, the object will not need to worry about the irrel-

evant from its perspective issues of exposing, and the data structure will only be allocated when actually needed.

The following global variables, useful for this purpose, are available from the exposure code:

```
int DisplayOpening, DisplayClosing;
void *TheWFrame;
```

If `DisplayOpening` is `YES`, it means that the exposure mode has been invoked for the first time. Otherwise, `DisplayOpening` is `NO`. If, upon recognizing the first invocation, the exposure code decides to create a dynamic data structure, the pointer to this structure should be stored in `TheWFrame`. In subsequent invocations of the same mode `TheWFrame` will contain the same pointer.

If `TheWFrame` is not `NULL`, then, when the window corresponding to a given exposure mode is closed, the exposure code is called for the last time with `DisplayClosing` set to `YES`. No display information should be sent in that case: the last invocation should be used to deallocate the structure pointed to by `TheWFrame`. Note that the closing invocation is only made when `TheWFrame` is not `NULL`, i.e., it points to something.

Example

The exposure from the previous example can be rewritten in the following way:

```
RVarList::exposure {
  double min, max, mean, *Means, Min, Max;
  int i;
  LONG count;
  onscreen {
    exmode 0:
      if (DisplayOpening)
        TheWFrame = (void*) (Means = new double [MaxVars]);
      else if (DisplayClosing) {
        delete (double*) TheWFrame;
        return;
      } else
        Means = (double*) TheWFrame;
      if (VarCount == 0) return;
      Min = HUGE; Max = -HUGE;
      for (i = 0; i < VarCount; i++) {
        RVars [i] -> calculate (&min, &max, &mean, &count);
        if (min < Min) Min = min;
```

```
            if (max > Max) Max = max;
            Means [i] = mean;
          }
          startRegion (0.0, (double)(VarCount-1), Min, Max);
          startSegment (0xc0);
          for (i = 0; i < VarCount; i++)
            displayPoint ((double)i, Means [i]);
          endSegment ();
          endRegion ();
      }
    };
```

Note that it is impossible for `DisplayOpening` and `DisplayClosing` to be set at the same time. The only action performed by the exposure when `DisplayClosing` is set is the deallocation of the scratch array.

17.4 Invoking exposures

An object is exposed by calling one of the following three methods defined within *Object*:

```
void printOut (int m, const char *hdr = NULL, Long sid = NONE);
void printOut (int m, Long sid);
void displayOut (int m, Long sid = NONE);
```

The first two methods expose the object "on paper," the last one makes a "screen" exposure. The second method behaves identically to the first one with `hdr` equal `NULL`. In each case, the first argument indicates the exposure mode. Depending on whether the exposure is "on paper" (the first two methods) or "on screen" (the last method), the appropriate code fragment from the object's exposure definition, determined by the form and mode (`m`), is selected and invoked. For the first method, the contents of the second and the third arguments are made available to the exposure code via `Hdr` and `SId`, respectively (section 17.2). With the second method, `Hdr` is set to `NULL` and the second argument is stored in `SId`.

Usually, the protocol program never requests "screen" exposures directly, i.e., it does not call `displayOut`. One exception is when the "screen" exposure code for a compound type T requests exposures of subobjects (see section 17.2). In such a case, the exposure code for T may call `displayOut` for a subobject: this will have the effect of sending the subobject's display information to the display program.

The "screen" exposure of an object can be requested practically at any moment. SIDE cooperates with the display program (DSD) using a special protocol and, generally, this

communication is transparent to the user. There are a number of variables and functions that make it visible to the extent of providing some potentially useful tools.

If the global integer variable `DisplayActive` contains `YES`, it means that DSD is connected to the protocol program. During such a connection, DSD maintains a number of *windows* whose contents are to be refreshed periodically. A single window corresponds to one display mode of a single object. SIDE maintains some internal description of the set of windows currently requested by the display program. From the point of view of a SIDE program, such a window is a pair: an object pointer and a display mode (an integer number). The SIDE program is not concerned with the window layout: it just sends to DSD the "raw" data, according to the object's "screen" exposure.

This information is sent at some intervals that can be changed upon a request from the display program. If SIDE operates in the virtual mode (as a simulator), the default interval separating two consecutive updates is 5000 events. In the real mode, this interval is expressed in milliseconds and its default length is 1000 (i.e., one second). In the visualization mode, the display interval coincides with the "resync grain" (section 7.10) and cannot be changed.

The current length of the *display interval* (i.e., the amount of time separating two consecutive updates sent by the SIDE program to DSD expressed in events or in milliseconds) is kept in the global integer variable `DisplayInterval`. Both `DisplayActive` and `DisplayInterval` are read-only variables which should not be changed by the protocol program.

A protocol program may request connection to the display program explicitly, e.g., upon detecting a situation that may be of interest to the user. This is done by calling the following function:

```
int requestDisplay (const char *msg = NULL);
```

which halts the simulation until a connection to the display program is established (see section 18.6.2). The function returns `OK` when the connection has been established and `ERROR`, if the connection was already established when the function was called. Note that `requestDisplay` blocks the protocol program and waits indefinitely for DSD to come in. Therefore, the function is not recommended for control programs—at least it should be used with due care.

The optional argument is a character string representing a textual message to be sent to the display program immediately after establishing connection. The protocol program may send a textual message to the display program at any moment during the connection by calling the function

```
void displayNote (const char *msg);
```

where `msg` is a pointer to the message. The function does nothing if DSD is not connected to the protocol program.

While the protocol program is connected to the display program, it may request that the screen contents be updated before the end of the display interval is reached. By calling the function

```
void refreshDisplay ();
```

the protocol program sends the exposure information for the currently defined configuration of windows to the display program.

The protocol for sending window updates to DSD is based on credits. The SIDE program is typically allowed to send a number of successive updates before receiving anything back from the display program. Each update takes one credit and if the program runs out of credits, it has to receive some from DSD to continue sending window updates. This way DSD may reduce the actual frequency of updates arriving from the SIDE program below the nominal rate determined by the current value of `DisplayInterval`.

17.5 Standard exposures

Starting from section 17.5.2, we will be describing standard exposures defined for the built-in *Object* types of SIDE. There are some conventions that are obeyed (with minor exceptions) by all standard exposures.

Unless explicitly stated, it is assumed by default that the number of "screen" exposure modes is the same as the number of "paper" exposure modes, and that the corresponding modes of both forms send out exactly the same items. Thus, we will not discuss "screen" exposures, except for the few special cases when their information contents differ from the information contents of the corresponding "paper" exposures.

The default header (section 17.2) of a standard "paper" exposure contains the *output name* of the exposed object (section 5.2). In most cases, the simulated time of the exposure is also included in the header.

All standard exposable types define mnemonic abbreviations for requesting "paper" exposures. For example, calling

```
prt->printRqs ();
```

where `prt` points to a `Port`, is equivalent to calling

```
prt->printOut (0);
```

i.e., to requesting the port's "paper" exposure with mode 0 (section 17.4). The mode 0 exposure for a port prints out information about the pending wait requests to the port.

In general, the type specification of a method that provides an abbreviated way of requesting "paper" exposures may have one of the following forms:

```
print xxx (const char *hdr = NULL, Long sid = NONE);
print xxx (const char *hdr = NULL);
```

where *xxx* stand for three or four letters related to the information contents of the exposure. With the first version, by specifying a station `Id` as the second argument, it is possible to make the exposure station-relative (sections 17.1, 17.2); no station-relative exposure is available for the second version. For both versions, the optional first argument may specify a non-standard header.

While describing the information contents of an exposure, we will ignore headers and other "fixed" text fragments. The user will have no problems identifying these parts in the output file. Fixed text fragments are never sent to the display program as part of a "screen" exposure. We will discuss the particular data items in the order in which they are output which, unless stated otherwise, is the same for the two exposure forms.

Quite often the output size of a list-like exposure may vary, depending on the number of elements in the list. All such exposures organize the output information into a sequence of rows, all rows obeying the same layout rules. Thus, in such cases, we will describe the layout of a single row understanding that the actual number of rows may vary.

17.5.1 Event identifiers

Many standard exposures produce information about (pending) wait requests and various events. By default, only the wait requests issued by protocol processes and events perceptible by those processes are visible in such exposures. There are some internal processes of SIDE that issue internal wait requests and respond to some special events which are rather exotic from the user's point of view. To include information about these internal requests and events in the exposed data, the protocol program should be called with the '`-s`' option. This also concerns the event information generated with simulator-level tracing (see section 16.2).

Information produced by standard exposures often contains *event identifiers*. Event identifiers are *AI*-specific: they provide textual representations for events triggered by the *AI*s. We list those identifiers below, separately for each *AI* type.

`Timer`

> A `Timer` event (section 8.1) is triggered when the delay specified in the corresponding wait request elapses. If `BIG_precision` is 1, the event identifier is the numerical value of the delay in *ITU*s. Otherwise, the character string "`wakeup`" is used to represent all events.

`Mailbox`

`Mailbox` events (section 13.4.6) are generated when items are stored in mailboxes or removed from mailboxes. The event identifier can be one of the character strings: "`EMPTY`", "`GET`", "`NEWITEM`", "`NONEMPTY`", "`OUTPUT`", "`RECEIVE`", or a number identifying a count event.

`Port`

The identifier of a port event is a character string containing the symbolic name of the event, according to section 11.2, i.e., "`SILENCE`", "`ACTIVITY`", "`COLLISION`", etc.

`Transceiver`

The identifier of a transceiver event is a character string containing the symbolic name of the event, according to section 12.2.1, i.e., "`SILENCE`", "`ACTIVITY`", "`BOP`", etc.

`Client`

The event identifier for the `Client` is one of the following character strings: "`ARRIVAL`", "`INTERCEPT`" (see section 10.9), "`SUSPEND`", "`RESUME`" ((section 10.6.3), "`RESET`" ((section 14.2.4), or "`arr_Trf`*xxx*." The sequence *xxx* in the last string stands for the `Id` of the traffic pattern whose message arrival is awaited. This case corresponds to a `Client` wait request with the first argument identifying a traffic pattern (section 10.9).

`Traffic`

The event identifier for a `Traffic` *AI* is one of the following four strings: "`ARRIVAL`", "`INTERCEPT`" (see section 10.9), "`SUSPEND`", "`RESUME`" (see (section 10.6.3), "`RESET`" ((section 14.2.4).

`Process`

A process can be viewed as an *AI* (section 7.7). The event identifier for the process *AI* is one of the following character strings: "`START`", "`DEATH`", "`SIGNAL`", "`CLEAR`", or a state name. The first string represents the virtual event used to start the process: a process that appears to be waiting for it is a new process that has just been created—within the current *ITU*. The "`DEATH`" event is triggered by the process termination. Similar to "`START`", it occurs only once in the lifetime of a process. Events "`SIGNAL`" and "`CLEAR`" are related to the process's signal repository (section 7.8.1). The state name represents an event that will be generated by the process *AI* when the process gets into the named state (section 7.7).

System events

Neither links nor radio channels generate any events that can be directly awaited by protocol processes.

There exists a system process called `LinkService` which takes care of removing obsolete activities from links and their archives. The following two system events are generated by link *AI*s and perceived by the `LinkService` process:

`LNK_PURGE`
: to indicate that an activity should be removed from the link and possibly added to the archive

`ARC_PURGE`
: to indicate that an activity should be removed from the link archive and destroyed

Another system process, called `RosterService`, takes care of various administrative duties related to radio channels. The following system events perceived by that process appear to originate at transceivers rather than radio channels:

`DO_ROSTER`
: responsible for advancing activity stages at perceiving transceivers (section 12.1)

`BOT_TRIGGER`
: initiating proper packet transmission at the sending transceiver following the end of preamble

`RACT_PURGE`
: responsible for cleaning activities that have disappeared for the channels

The following two internal events are generated by the `Client` *AI* and sensed by the system process called `ClientService` which is responsible for traffic generation:

`ARR_MSG`
: to indicate that a new message is to be generated and queued at a station

`ARR_BST`
: to indicate that a new burst is to be started

One more system process responding to some internal event is `ObserverService`. This process is responsible for restarting an observer after its `timeout` delay has elapsed. This event is triggered by a virtual *AI* called `Observer` and its identifier is "`TIMEOUT`."

17.5.2 `Timer` exposure

Mode 0: full request list

Calling: `printRqs (const char *hdr = NULL, Long sid = NONE);`

The full list of processes waiting for the `Timer` is produced. The list has the form of a table with each row consisting of the following entries (in this order):

1. The modeled time in *ITU*s when the `Timer` event will occur.
2. A single-character flag that describes the status of the `Timer` request: '`*`' means that according to the current state of the simulation, the `Timer` event will actually restart the process, blank means that another waking event will occur earlier than the `Timer` event, and '`?`' says that the `Timer` event *may* restart the process, but another event has been scheduled at the same *ITU* and has the same order as the `Timer` event.
3. The `Id` of the station owning the process or "`Sys`", for a system process. This data item is not included if the station-relative variant of the exposure is selected.
4. The process type name.
5. The process `Id`.
6. The identifier of the process's state associated with the `Timer` request, i.e., the state where the process will be restarted by the `Timer` event, should this event actually restart the process.
7. The type name of the activity interpreter that, according to the current state of the simulation, will restart the process.
8. The `Id` of the activity interpreter or blanks if the *AI* has no `Id` (`Client`, `Timer`).
9. The identifier of the event that will wake the process up.
10. The identifier of the state where, according to the current configuration of activities and events, the process will be restarted.

The station-relative version of the exposure restricts the list to the processes belonging to the indicated station.

Exactly the same items are sent to the display program with the "screen" version of the exposure.

For all *AI*s, the mode 0 exposure produces similar information, according to the above layout. When the time of the event occurrence (item 1) is not known at the moment of exposure (it cannot happen for a `Timer` wait request), the string "`undefined`" is printed in its place.

Mode 1: abbreviated request list

Calling: `printARqs (const char *hdr = NULL, Long sid = NONE);`

The processes waiting for `Timer` events are listed in the abbreviated format. The list has the form of a table with each row consisting of the following entries (in this order):

1. The simulated time in *ITU*s when the `Timer` event will occur.
2. A character flag describing the status of the `Timer` request (see above).
3. The `Id` of the station owning the process. This data item is not included if the station-relative variant of the exposure is selected.
4. The process type name.
5. The process `Id`.
6. The identifier of the process's state associated with the `Timer` request.

Only user processes are included in the abbreviated list, even if the '`-s`' call option is used. The station-relative version of the exposure restricts the list to the processes belonging to the indicated station.

Exactly the same items are sent to the display program with the "screen" version of the exposure.

17.5.3 `Mailbox` exposure

Mode 0: full request list

Calling: `printRqs (const char *hdr = NULL, Long sid = NONE);`

The full list of processes awaiting events on the `Mailbox`. The description is the same as for the `Timer` exposure with mode 0, except that the word "`Timer`" should be replaced by "`Mailbox`."

Mode 1: abbreviated request list

Calling: `printARqs (const char *hdr = NULL, Long sid = NONE);`

The processes awaiting events on the `Mailbox` are listed in the abbreviated format. The description is the same as for the `Timer` exposure with mode 1, except that the word "`Timer`" should be replaced by "`Mailbox`."

Mode 2: mailbox contents

Calling: `printCnt (const char *hdr = NULL);`

The exposure produces information about the mailbox contents. The following items are printed in one line:

1. the header argument or the mailbox output name (section 5.2), if the header argument is not specified
2. the number of elements currently stored in the mailbox (for a bound mailbox—section 13.4.1–this is the number of bytes in the input buffer)
3. the mailbox capacity (for a bound mailbox, this is the size of the input buffer)
4. the number of pending wait requests issued to the mailbox

The line containing the value of the above-listed items is preceded by a header line.

The "screen" version of the exposure sends to the display program the last three items, i.e., the mailbox name is not sent. Of course, the header line is also absent.

Mode 3: short mailbox contents

Calling: `printSCnt (const char *hdr = NULL);`

The paper version of the exposure produces the same output as the mode 2 exposure (see above), except that no header line is printed. Thus, the output can be used as part of a longer output, e.g., listing the contents of several exposures. No "screen" version of this exposure is provided. Note that the "screen" version of the mode 2 exposure is already devoid of the header line.

17.5.4 `RVariable` exposure

Mode 0: full contents

Calling: `printCnt (const char *hdr = NULL);`

The exposure outputs the contents of the random variable (section 14.1). The following data items are produced (in this order):

1. The number of samples.
2. The minimum value.
3. The maximum value.
4. The mean value (i.e., the first central moment).
5. The variance (i.e., the second central moment).
6. The standard deviation (i.e., the square root of the second central moment).
7. The relative confidence margin for probability $P_a = 0.95$. This value is equal to half the length of the confidence interval at $P_a = 0.95$ divided by the absolute value of the calculated mean.
8. The relative confidence margin for probability $P_a = 0.99$.

If the random variable was created with more than two moments (section 14.1.1), the above list is continued with their values. If the number of moments is 1, items 5–8 do not appear; if it is 0, item 4 is skipped as well.

The "screen" version of the exposure contains the same numerical items as the "paper" exposure without item 8. This list is preceded by a region (section 17.3) that displays graphically the history of the 24 last exposed mean values of the random variable. The region consists of a single segment including up to 24 points. It is scaled by (0, 23) in the x axis and (min, max) in the y axis, where min and max are the current minimum and maximum values of the random variable.

Mode 1: abbreviated contents

Calling: `printACnt (const char *hdr = NULL);`

The exposure outputs the abbreviated contents of the random variable. The following data items are produced (in this order):

1. the number of samples
2. the minimum value
3. the maximum value
4. the mean value, i.e., the first central moment (not included if the random variable does not have at least one moment)

The values are preceded by a header line.

The "screen" version of the exposure contains the same numerical items as the mode 0 "paper" exposure, i.e., the mode 0 "screen" exposure without the region.

Mode 2: short contents

Calling: `printSCnt (const char *hdr = NULL);`

The exposure outputs the short contents of the random variable. The items produced are identical to those for mode 1. The difference is that no header line is printed above the items which allows to expose multiple random variables in the form of a table. This can be done by exposing the first random variable with mode 1 and the remaining ones with mode 2.

The "screen" version of the exposure contains the same numerical items as the "paper" exposure followed additionally by the standard deviation. If the random variable has less than two standard moments, the missing values are replaced by dashes (`---`).

17.5.5 `Client` exposure

Mode 0: request list

Calling: `printRqs (const char *hdr = NULL, Long sid = NONE);`

This exposure produces the list of processes waiting for `Client` and `Traffic` events. The description is the same as for the `Timer` exposure with mode 0, except that the word "`Timer`" should be replaced by "`Client` or `Traffic`."

Mode 1: performance measures

Calling: `printPfm (const char *hdr = NULL);`

This exposure lists the standard performance measures for all standard traffic patterns viewed globally, as if they constituted a single traffic pattern. The exposure is produced by *combining* (section 14.1.2) the standard random variables belonging to individual traffic patterns (section 14.2.1) into a collection of global random variables reflecting the performance measures of the whole standard `Client`, and then exposing them with mode 0 (`printCnt`—section 17.5.4) in the following order:

1. `RVAMD`—the absolute message delay
2. `RVAPD`—the absolute packet delay
3. `RVWMD`—the weighted message delay
4. `RVMAT`—the message access time
5. `RVPAT`—the packet access time
6. `RVMLS`—the message length statistics

The data produced by the above list of exposures are followed by the client statistics containing the following items:

1. The number of all messages ever generated and queued at their senders.
2. The number of messages currently queued awaiting transmission.
3. The number of all messages completely received (section 11.2.3).
4. The number of all transmitted packets (terminated by `stop`—section 11.1.2).
5. The number of all received packets.
6. The number of all message bits queued at stations awaiting transmission. This is the combined length of all messages currently queued (item 2).
7. The number of all message bits successfully transmitted so far. This is the combined length of all packets from item 4.
8. The number of all message bits successfully received so far. This is the combined length of all packets from item 5.
9. The global throughput of the network calculated as the ratio of the number of received bits (item 8) to the simulated time in *ETU*.

The "screen" version of the exposure does not include the last 9 items. The 6 random variables representing the global performance measures are exposed with mode 1 (section 17.5.4).

Mode 2: message queues

Calling: `printCnt (const char *hdr = NULL, Long sid = NONE);`

This exposure produces information about message queues at all stations, or at one indicated station, for the station-relative variant. The global variant prints for each station one line consisting of the following three items:

1. the station `Id`
2. the number of messages queued at the station
3. the number of message bits queued at the station, i.e., the combined length of all messages queued at the station

The station-relative variant of the exposure produces one row for each message queued at the indicated station. This row contains the following data:

1. time in *ITU*s when the message was queued
2. `Id` of the traffic pattern to which the message belongs
3. message length in bits
4. `Id` of the station to which the message is addressed, or "`bcast`" for a broadcast message

The global variant of the "screen" version produces a region that displays graphically the length of message queues at all station. The region consists of a single segment containing `NStations` points (section 6.1.2) scaled by (0, `NStations`–1) on the x axis and (0, L_{max}) on the y axis, where L_{max} is the maximum length of a message queue at a station. The y co-ordinate of a point is equal to the number of messages queued at the station whose `Id` is determined by the x co-ordinate.

Mode 3: traffic definition/client statistics

Calling: `printDef (const char *hdr = NULL);`

The "paper" variant of this exposure prints out information describing the definition of all traffic patterns. This information is self-explanatory.

The "screen" variant of mode 3 displays the client statistics containing the same items as the client statistics printed by the mode 2 exposure. The order of these items is also the same, except that the throughput is moved from the last position to the first.

17.5.6 `Traffic` exposure

The exposure modes, their "paper" abbreviations, and contents are identical to those of the `Client` with the following exceptions:

- With mode 0, the list of processes is limited to the protocol processes awaiting events from the given traffic pattern.
- The performance measures produced by mode 1 refer to the given traffic pattern. The client statistics printed by mode 1 and displayed by mode 3 refer to the given traffic pattern and they do not contain the throughput item.
- The output produced by the station-specific variant of mode 2 does not contain the traffic pattern `Id`: this output, similarly as the output of the station-relative variant is restricted to the given traffic pattern.
- The information printed by the mode 3 exposure describes the definition of the given traffic pattern.

17.5.7 `Port` exposure

Mode 0: request list

Calling: `printRqs (const char *hdr = NULL);`

The exposure produces the list of processes waiting for events on the port. The description is the same as for the `Timer` exposure with mode 0, except that the word "`Timer`" should be replaced by "`Port`." Note that no station-relative exposures are defined for ports: a port always belongs to some station and this way all its exposures are implicitly station-relative.

Mode 1: list of activities

Calling: `printAct (const char *hdr = NULL);`

This exposure produces the list of all activities currently present in the link to which the port is connected, and in the link's archive. The list is sorted in the non-decreasing order of the time when the beginning of the activity was, is, or will be heard on the port. Each activity takes one row consisting of the following items:

1. A single-letter activity type designator: '`T`'—a transfer (or aborted transfer attempt), '`J`'—a jamming signal, '`C`'—a collision (see below).

2. The *starting time* of the activity in *ITU*s, as perceived by the port.
3. The *finished time* of the activity in *ITU*s, as perceived by the port. If this time is not known at present (i.e., the activity is still being inserted into the link), the string "`undefined`" is written.
4. The `Id` of the station that started the activity.
5. The station-relative port number (section 6.3.1).
6. The `Id` of the receiver station or "`bcst`" for a broadcast packet.
7. The traffic pattern `Id`.
8. The packet signature (see section 10.2).

Items 6–8 are meaningful for packets only. If the activity is not a packet transmission, "`---`" appears in place of each of the three items. In the "paper" version of the exposure, item 8 is only printed if the program has been created with the `-g` (or `-G`) option of `mks` (section 19.2). In the "screen" version, item 8 is always displayed,[47] but unless the program has been created with `-g/-G`, item 8 is filled with dashes.

If, according to the port's perception, the current configuration of activities results in a future or present collision, a virtual activity representing the collision is included in the activity list. Only the first two attributes of this activity (i.e., the activity type designator and the starting time) are meaningful; the remaining items are printed as "`---`."

Exactly the same items are displayed with the "screen" form of the exposure.

Mode 2: predicted events

Calling: `printEvs (const char *hdr = NULL);`

The exposure describes the timing of future or present events on the port (section 11.2). The description of each event takes one row consisting of the following items:

1. the event identifier (e.g., "`ACTIVITY`", "`BOT`", "`COLLISION`")
2. the time when the event will occur or "`---`", if no such event is predictable at this moment
3. the type of activity triggering the event ("`T`" for a packet transmission, "`J`" for a jamming signal)

[47] Window templates (section 18.4) are independent of the protocol program, so their layout cannot be influenced by the options of `mks`.

4. the `Id` of the station that started the triggering activity
5. the station-relative number of the port on which the activity was started
6. the `Id` of the receiving station, if the triggering activity is a packet transmission ("`---`" is printed otherwise)
7. the traffic pattern `Id` of the triggering activity, if the activity is a packet transmission ("`---`" is printed otherwise)
8. the total length of the packet in bits, if the triggering activity is a packet transmission ("`---`" is printed otherwise)
9. the packet signature (see the comments regarding item 8 for mode 1 `Port` exposure)

The list produced by this exposure contains exactly 10 rows, one row for each of the 10 event types discussed in section 11.2. If, based on the current configuration of link activities, no event of the given type can be anticipated, all entries in the corresponding row, except for the first one, contain "`---`."

The information displayed by the "screen" form of the exposure contains the same data as the "paper" form.

17.5.8 `Link` exposure

Mode 0: request list

Calling: `printRqs (const char *hdr = NULL, Long sid = NONE);`

The exposure produces the list of processes waiting for events on ports connected to the link. The description is the same as for the `Timer` exposure with mode 0, except that the word "`Timer`" should be replaced by "a port connected to the link." If the '`-s`' run-time option is selected (section 19.3), the system processes waiting for the internal events generated directly by the link *AI* (section 17.5.1) are included in the list.

The station-relative version of the exposure restricts the output to the processes belonging to the indicated station.

Mode 1: performance measures

Calling: `printPfm (const char *hdr = NULL);`

The exposure outputs statistical data describing the amount of information passed through the link. The following items are produced (in this order):

1. The total number of jamming signals ever inserted into the link.

2. The total number of packet transmissions ever attempted (i.e., started—see section 11.1.2) in the link.

3. The total number of packet transmissions terminated by `stop`.

4. The total number of information bits (packet's attribute `ILength`—see section 10.2) transmitted via the link. A bit becomes transmitted if the transmission of the packet containing it is terminated by `stop`.

5. The total number of packets received on the link. A packet becomes received when `receive` (section 11.2.3) is executed for the packet. Note that the second argument of this `receive` must identify the link (or one of its ports).

6. The total number of information bits received on the link (see 5).

7. The total number of messages transmitted via the link. A message becomes transmitted when the transmission of its last packet is terminated by `stop`.

8. The total number of messages received on the link. A message becomes received when `receive` is executed for its last packet. Note that the second argument of this `receive` must identify the link (or one of its ports).

9. The number of damaged packets inserted into the link (see section 14.3).

10. The number of header-damaged packets inserted into the link.

11. The number of damaged bits inserted into the link.

12. The *received* throughput of the link determined as the ratio of the total number of bits received on the link (item 6) to the simulated time expressed in *ETU*s.

13. The *transmitted* throughput of the link determined as the ratio of the total number of bits transmitted on the link (item 4) to the simulated time expressed in *ETU*s.

The "screen" version of the exposure displays the same items. The two throughput measures are displayed first and followed by the remaining items, according to the above order.

Items 9, 10, and 11 do not appear in the link exposure if the program was not created with the `-z` option of `mks` (section 19.2).

Mode 2: activities

Calling: `printAct (const char *hdr = NULL, Long sid = NONE);`

This exposure produces the list of activities currently present in the link and the link's archive. The description of each activity takes one row that looks exactly as a row produced by the `Port` exposure with mode 1, except for the following:

- Type "`C`" virtual activities are omitted as collisions occur on ports, not in links.
- The starting and ending times reflect the actual starting and ending times of the activity, at the port responsible for its insertion.
- In the station-relative version of the exposure, the `Id` of the station inserting the activity is not printed. Only the activities inserted by the indicated station are printed in this mode.

The "screen" form of the exposure displays the same items as the "paper" form.

17.5.9 `Transceiver` exposure

Mode 0: request list

Calling: `printRqs (const char *hdr = NULL);`

The exposure produces the list of processes waiting for events on the transceiver. The description is the same as for the `Timer` exposure with mode 0, except that the word "`Timer`" should be replaced by "`Transceiver`." Note that no station-relative exposures are defined for transceivers: a transceiver always belongs to some station and this way all its exposures are implicitly station-relative.

Mode 1: list of activities

Calling: `printRAct (const char *hdr = NULL);`

This exposure produces the list of all activities currently perceived by the transceiver. The title line specifies the total number of those activities and the total signal level (as returned by `sigLevel`—section 12.2.5). If the transmitter is active, the title also includes the word `XMITTING`. Each activity is described by a single line with the following items:

1. The station `Id` of the sender.

2. The station `Id` of the intended recipient, "`none`" (if the recipient is unspecified), or "`bcst`" for a broadcast packet.

3. The packet type, i.e., the `TP` attribute (section 10.2).

4. The total length of the packet (attribute `TLength`).

5. The status consisting of two characters. If the activity has been assessed as non-receivable (section 12.1.2), the first character is an asterisk, otherwise it is blank. The second character can be: `D` for "done" (the packet is past its last bit and it does not count any more), `P`—before stage P (section 12.1.1), `T`—before stage T, `A`—before stage T, but stage T will not occur because the preamble has been aborted, E—past stage T, `O`—own packet, i.e., transmitted by this transceiver.

6. The time when the transmission of the proper packet commenced at the sender. Note that this time may be undefined.

7. Perceived signal level of the activity. For an "own" activity, this is the current transmission power (`XPower`) of the transceiver.

8. Current interference level suffered by the activity.

9. Maximum interference level suffered by the activity, according to its present stage (section 12.1).

10. Average interference level suffered by the activity.

11. The packet's signature (section 10.2), included if the simulator has been compiled with `-g` or `-G`.

Items 8–10 do not show up for an "own" activity and are substituted with dashes.

Exactly the same items are displayed with the "screen" form of the exposure, except that item 11 is always included. Unless the program has been created with `-g` or `-G`, item 11 is filled with dashes.

Mode 2: the neighborhood

Calling: `printNei (const char *hdr = NULL);`

The exposure lists the current population of neighbors (as determined by `RFC_cut`—section 12.1.3) of the transceiver. The title line presents the transceiver's location (two or three coordinates in DUs) followed by one line per neighbor, including:

- The station `Id`.

- The station-relative transceiver `ID`.
- The distance in DUs from the current transceiver.
- The coordinates (in DUs) specifying the neighbor's location. These can be two or three numbers depending on the dimensionality of the node deployment space (section 6.5.1).

The information displayed by the "screen" form of the exposure contains the same data as the "paper" form. Owing to the rigid structure of the window template (section 18.4), the list of node coordinates always includes three items (with the third item being cropped out by the standard geometry of the window). In the 2-D case, the third coordinate is blank.

17.5.10 `RFChannel` exposure

Mode 0: request list

Calling: `printRqs (const char *hdr = NULL, Long sid = NONE);`

The exposure produces the list of processes waiting for events on ports connected to the link. The description is the same as for the `Timer` exposure with mode 0, except that the word "`Timer`" should be replaced by "a transceiver interfaced to the radio channel." If the '`-s`' run-time option is selected (section 19.3), system processes waiting for internal events (section 17.5.1) are included in the list.

The station-relative version of the exposure restricts the output to the processes belonging to the indicated station.

Mode 1: performance measures

Calling: `printPfm (const char *hdr = NULL);`

The exposure outputs statistical data describing the amount of information passed through the radio channel. The following items are produced (in this order):

1. The total number of packet transmissions ever attempted (i.e., started—see section 12.1.8) in the channel.
2. The total number of packet transmissions terminated by `stop`.
3. The total number of information bits (packet's attribute `ILength`—see section 10.2) transmitted over the channel. A bit becomes transmitted if the transmission of the packet containing it is terminated by `stop`.

4. The total number of packets received on the channel. A packet becomes received when `receive` (section 12.2.3) is executed for the packet. Note that the second argument of this `receive` must identify the channel (or one of its transceivers).

5. The total number of information bits received on the channel (see 4).

6. The total number of messages transmitted via the channel. A message becomes transmitted when the transmission of its last packet is terminated by `stop`.

7. The total number of messages received on the channel. A message becomes received when `receive` is executed for its last packet. Note that the second argument of this `receive` must identify the channel (or one of its transceivers).

8. The *received* throughput of the channel determined as the ratio of the total number of bits received on the channel (item 5) to the simulated time expressed in *ETU*s.

9. The *transmitted* throughput of the link determined as the ratio of the total number of bits transmitted on the link (item 3) to the simulated time expressed in *ETU*s.

The "screen" version of the exposure displays the same items. The two throughput measures are displayed first and followed by the remaining items, according to the above order.

Mode 2: activities

Calling: `printRAct (const char *hdr = NULL, Long sid = NONE);`

This exposure produces the list of activities currently present in the radio channel. Each activity is represented by a single line comprising the following items:

1. The station `Id` of the sender. This item is not included for a station-relative variant of the exposure.

2. The station-relative transceiver `Id` of the sender.

3. The starting time of the proper packet transmission (this time can be undefined).

4. The ending time of the packet transmission (this time can be undefined).

5. The station `Id` of the intended recipient, "`none`" (if the recipient is unspecified), or "`bcst`" for a broadcast packet.

6. The packet type, i.e., the `TP` attribute (section 10.2).

7. The total length of the packet (attribute `TLength`).

8. The packet's signature (section 10.2), included if the simulator has been compiled with `-g` or `-G`.

An asterisk following the ending time of transmission indicates an aborted packet.

Exactly the same items are displayed with the "screen" form of the exposure, except that item 8 is always included. Unless the program has been created with `-g` or `-G`, item 8 is filled with dashes.

Mode 3: topology

Calling: `printTop (const char *hdr = NULL);`

This exposure presents the configuration of transceivers interfaced to the channel. The title line shows the number of transceivers, the coordinates of the minimum rectangle encompassing them all, and the length of the diagonal of that rectangle. This is followed by the list of transceivers (one line per transceiver), followed in turn by the list of transceiver neighborhoods. One entry from the transceiver list contains the following items:

1. The `Id` of the station to which the transceiver belongs.
2. The station-relative `Id` of the transceiver.
3. The transceiver's transmission rate, i.e., attribute `TRate` (section 6.5).
4. The current setting of the transmission power (attribute `XPower`).
5. The current setting of the receiver gain (attribute `RPower`).
6. The number of neighbors, as determined by `RFC_cut` (section 6.5.2).
7. The X and Y coordinates of the transceiver in DUs. For the 3-D variant on the node deployment space, one more coordinate (Z) is included.

A neighborhood description begins with the identifier of a transceiver followed by the identifiers of all transceivers in its neighborhood with their associated distances from it in DUs. Each transceiver identifier is a pair: station `Id`/station-relative transceiver `Id`. A list like this is produced for every transceiver configured into the radio channel.

The "screen" variant of this exposure presents a region with different transceivers marked as points (black circles). There exists a station-relative variant of this exposure, which marks the transceivers belonging to the indicated station red and their neighbors green. For the 3-D variant of node deployment space, the Z coordinate is ignored, i.e., the nodes are projected onto the 2-D plane with Z= 0.

17.5.11 `Process` exposure

Mode 0: request list

Calling: `printRqs (const char *hdr = NULL, Long sid = NONE);`

The exposure produces the list of processes waiting for events to be triggered by the `Process` *AI* (see section 7.7). The description is the same as for the `Timer` exposure with mode 0, except that the word "`Timer`" should be replaced by "`Process`."

The station-relative version of the exposure restricts the list to the processes belonging to the indicated station.[48]

Mode 1: wait requests of this process

Calling: `printWait (const char *hdr = NULL);`

The exposure outputs the list of pending wait requests issued by the process. Each wait request is described by one row containing the following items (in this order):

1. type name of the activity interpreter to which the request has been issued
2. `Id` of the activity interpreter or blanks, if the *AI* has no `Id` (e.g., `Timer`)
3. identifier of the awaited event (see section 17.5.1)
4. identifier of the process state to be assumed when the event occurs
5. time in *ITU*s when the event will be triggered or "`undefined`", if the time is unknown at present

The "screen" version of the exposure displays the same items in exactly the same order.

17.5.12 `Kernel` exposure

The `Kernel` process (see section 7.9) defines a separate collection of exposure modes. These modes present information of a general nature.

[48] In the vast majority of cases a process waiting for an event from a process *AI* belongs to the same station as the process *AI*; thus, there seems to be little need for the station-relative version.

Mode 0: full global request list

Calling: `printRqs (const char *hdr = NULL, Long sid = NONE);`

This exposure prints out the full list of processes waiting for some events. Each wait request is represented by one row containing the following data (in this order):

1. The `Id` of the station owning the waiting process or "`Sys`", for a system process. This item is not printed by the station-relative version of the exposure.
2. The process type name.
3. The process `Id`.
4. The simulated time in *ITU*s when the event will occur or "`undefined`", if this time is unknown at present.
5. A single-character flag that describes the status of the wait request: '`*`' means that according to the current state of the simulation, the awaited event will actually restart the process, blank means that another waking event will occur earlier than the awaited event, and '`?`' says that the awaited event *may* restart the process, but another event awaited by the process is scheduled at the same *ITU* with the same order.
6. The type name of the activity interpreter expected to trigger the awaited event.
7. The `Id` of the activity interpreter or blanks, if the *AI* has no `Id` (`Client`, `Timer`).
8. The event identifier (section 17.5.1).
9. The state where the process will be restarted by the awaited event, should this event actually restart the process.

The station-relative version of this exposure lists only the wait requests issued by the processes belonging to the indicated station.

For multiple wait requests issued by the same process, the first three items are printed only once—at the first request of the process—and blanks are produced instead of them in the subsequent rows representing requests of the same process.

The "screen" version of the exposure displays the same items in exactly the same order.

Mode 1: abbreviated global request list

Calling: `printARqs (const char *hdr = NULL, Long sid = NONE);`

This is an abbreviated variant of mode 0 exposure which produces a single row of data per one process that awaits some events. This row has the same layout as for the mode 0 exposure. It describes the wait request that, according to the current state of the simulation, will restart the process.

The same items are produced by the "screen" version of the exposure.

Mode 2: simulation status

Calling: `printSts (const char *hdr = NULL);`

The exposure prints out global information about the status of the simulation run. The following data items are produced (in this order):

1. process id of the SIDE program
2. CPU execution time in seconds
3. simulated time in *ITU*s
4. total number of events processed by the program so far
5. event queue size (the number of queued events)
6. total number of messages ever generated and queued at the senders by the standard `Client`
7. total number of messages entirely `received` (section 11.2.3)
8. total number of queued bits, i.e., the combined length of all messages queued at this moment
9. the global throughput determined as the total number of bits `received` so far divided by the simulation time in *ETU*
10. output name (see section 5.2) of the last active station, i.e., the station whose process was last awakened
11. output name of the last awakened process
12. output name of the *AI* that awakened the process
13. waking event identifier
14. state at which the process was restarted

Only items 1 through 10 (inclusively) are displayed by the "screen" version of the exposure—in the above order.

Mode 3: last event

`No "paper" form.`

This mode exists in the "screen" form only. It displays the information about the last awakened process, i.e., the last 5 items from the "paper" form of mode 3, in the same order.

17.5.13 `Station` exposure

Mode 0: process list

Calling: `printPrc (const char *hdr = NULL);`

This exposure prints out the list of processes belonging to the station and waiting for some events. Note that an alive process always waits for some event(s). Each wait request is represented by one row containing the following data (in this order):

1. The process output name (section 5.2).
2. The type name of the activity interpreter expected to trigger the awaited event.
3. The `Id` of the activity interpreter or blanks, if the *AI* has no `Id` (`Client`, `Timer`).
4. The event identifier (section 17.5.1).
5. The state where the process will be restarted by the awaited event, should this event actually restart the process.
6. The simulated time in *ITU*s when the event will occur or "`undefined`", if this time is unknown at present.
7. A single-character flag that describes the status of the wait request: '`*`' means that according to the current state of the simulation, the awaited event will actually restart the process, blank means that another waking event will occur earlier than the awaited event, and '`?`' says that the awaited event *may* restart the process, but another event awaited by the process has been scheduled at the same *ITU* with the same order.

For multiple wait requests issued by the same process, the first item is printed only once—at the first request of the process—and replaced with blanks in the subsequent rows representing requests of the same process.

The "screen" version of the exposure displays the same items in exactly the same order.

Mode 1: buffer contents

Calling: `printBuf (const char *hdr = NULL);`

This exposure prints out the contents of the packet buffers at the station. One row of information is printed per each buffer, in the order in which the buffers have been declared (section 10.3). The following data items are included:

1. The buffer number (from 0 to n–1), where n is the total number of packet buffers owned by the station.
2. The time when the message from which the packet in the buffer has been acquired was queued at the station.
3. The time when the packet became ready for transmission (attribute `TTime`—see section 14.2.1).
4. The `Id` of the packet's receiver or "`bcst`", for a broadcast packet.
5. The `Id` of the traffic pattern to which the packet belongs.
6. The *information length* of the packet (attribute `ILength`—see section 10.2).
7. The total length of the packet (attribute `TLength`).
8. Two standard flags of the packet: `PF_broadcast` (section 11.2.3) and `PF_last` (section 10.3). The third flag (`PF_full`), associated with the packet buffer, is not printed. When this flag is 0, the buffer is empty and no packet information is printed at all (see below). This item is a piece of text consisting of up to two possibly combined letters: `B` (for a broadcast packet) and `L` (for the last packet of a message). If neither of the two packet flags is set, the text is empty.
9. The packet signature (see the comments regarding item 8 for mode 1 `Port` exposure).

The items 2 through 9 are only printed if the packet buffer is nonempty; otherwise, the string "`empty`" is written in place of item 2 and "`---`" replaces the remaining items.

The "screen" version of the exposure has exactly the same contents.

Mode 2: mailbox contents

Calling: `printMail (const char *hdr = NULL);`

The exposure prints information about the contents of all mailboxes owned by the station. One row of data is produced for each mailbox with the following items (in this order):

1. station-relative serial number of the mailbox reflecting its creation order, or the mailbox nickname (section 5.2), if one is defined for the mailbox
2. number of elements currently stored in the mailbox (for a bound mailbox—13.4.1– this is the number of bytes in the input buffer)
3. mailbox capacity (or the input buffer size, if the mailbox is bound)
4. number of pending wait requests issued to the mailbox

The "screen" version of the exposure produces the same items.

Mode 3: link activities

Calling: `printAct (const char *hdr = NULL);`

The exposure prints out information about port (link) activities started by the station. All links are examined and all activities that were originated by the station and are still present in the link or the link archive are exposed. One row of data per activity is produced with the following contents:

1. station-relative number of the port (section 6.3.1) on which the activity was started
2. `Id` of the link to which the port is connected
3. *starting time* of the activity
4. *finished time* of the activity or "`undefined`", if the activity has not been finished yet
5. activity type: "`T`" for a transfer, "`J`" for a jam
6. `Id` of the receiver (or "`bcst`") if the activity is a packet transmission
7. `Id` of the traffic pattern to which the packet belongs
8. total length of the packet in bits
9. packet signature (see the comments regarding item 8 for mode 1 `Port` exposure)

Items 6 through 9 are only printed if the activity represents a packet transmission; otherwise, these items are replaced with "`---`."

Exactly the same data are displayed by the "screen" version of the exposure.

Mode 4: port status

Calling: `printPort (const char *hdr = NULL);`

The exposure produces information about the current status of the station's ports. One row of 10 items is printed for each port. The first item is the port identifier. It can be either the station-relative serial number reflecting the port creation order, or the port's nickname, if one is defined for the port (section 5.2). Each of the remaining items corresponds to one event that can be present or absent on the port. If the item is "`***`", the event is present (i.e., the port is currently perceiving the event). Otherwise, the item is "`...`" and the event is absent. The nine events are (in this order): `ACTIVITY`, `BOT`, `EOT`, `BMP`, `EMP`, `BOJ`, `EOJ`, `COLLISION`, `ANYEVENT` (see section 11.2).

The "screen" version of the exposure displays the same information, except that "`***`" is reduced to "`*`" and "`...`" to "`.`", respectively.

Mode 5: radio activities

Calling: `printRAct (const char *hdr = NULL);`

The exposure prints out information about radio activities started by the station. It examines all radio channels (`RFChannel`s) and produces one line for each activity found there that was transmitted on one of the station's transceivers. The line contains (in this order):

1. The numerical `Id` of the radio channel
2. The station-relative `Id` of the originating transceiver.
3. The starting time of the proper packet transmission (this time can be undefined).
4. The ending time of the packet transmission (this time can be undefined).
5. The station `Id` of the intended recipient, "`none`" (if the recipient is unspecified), or "`bcst`" for a broadcast packet.
6. The packet type, i.e., the `TP` attribute (section 10.2).
7. The total length of the packet (attribute `TLength`).
8. The packet's signature (section 10.2), included if the simulator has been compiled with `-g` or `-G`.

An asterisk following the ending time of transmission indicates an aborted packet.

Exactly the same items are displayed with the "screen" form of the exposure, except that item 8 is always included. Unless the program has been created with `-g` or `-G`, item 8 is filled with dashes.

Mode 6: transceiver status

Calling: `printPort (const char *hdr = NULL);`

The exposure produces information about the status of activities on the station's transceivers. One row of 13 items is printed for each transceiver. The first item is the transceiver identifier. It can be either the station-relative serial number reflecting the transceiver's creation order, or the nickname, if one is defined for the transceiver (section 5.2). The remaining items have the following meaning:

1. The receiver status: "`+`" (on) or "`-`" (off). Note that if the receiver is off, the remaining indications (except for "own" activity) correspond to no activity being heard by the transceiver.
2. The busy status of the receiver, as returned by `RFC_act` (section 12.2): "`Y`" (busy), "`N` (idle).
3. Own activity indicator: "`P`"—preamble being transmitted (packet before stage T), "`T`"—packet past stage T, blank—no packet being transmitted.
4. The total number of activities perceived by the transceiver.
5. The total number of simultaneous `BOP` events (section 12.2.1).
6. The total number of simultaneous `BOT` events.
7. The total number of simultaneous `EOT` events.
8. The total number of simultaneous `BMP` events.
9. The total number of simultaneous `EMP` events.
10. The total number of "any" events (see `ANYEVENT`—section 12.2.1).
11. The total number of preambles being perceived simultaneously.
12. The total number of packets (past stage T) being perceived simultaneously (including the non-receivable ones).
13. The total number of non-receivable packets based on assessment by `RFC_eot` (section 12.1.4).

The "screen" version of the exposure displays exactly the same information.

17.5.14 `System` exposure

The `System` station has its own two exposure modes that are used to print out information about the network topology/geometry. These modes have no "screen" versions.

Mode 0: full network description

Calling: `printTop (const char *hdr = NULL);`

Full information about the network configuration is printed out in a self-explanatory form.

Mode 1: abbreviated network description

Calling: `printATop (const char *hdr = NULL);`

Abbreviated information about the network configuration is printed. The output is self-explanatory.

Note: The contents of the mailboxes owned by the `System` station (section 13.2.1) can be printed/displayed using the mode 2 `Station` exposure.

17.5.15 `Observer` exposure

Mode 0: information about all observers

Calling: `printAll (const char *hdr = NULL);`

Information about all observers is printed out. This information includes the `inspect` lists and pending timeouts. A single row contains the following items:

1. *output name* of the observer
2. `Id` of the station specified in the `inspect` request or "`ANY`" (see section 16.3)
3. process type name or "`ANY`"
4. process *nickname* or "`ANY`"
5. process state or "`ANY`"
6. observer state to be assumed when the `inspect` succeeds

A pending `timeout` is printed in the form: "`Timeout at` *time*", where the text "`Timeout at`" is printed as item 2 and the time (in *ITU*s) as item 3. The remaining items are blank.

For multiple entries corresponding to one observer, the observer's name (item 1) is printed only once (at the first entry) and it is blank in the subsequent entries belonging to the same observer.

Exactly the same items are displayed by the "screen" version of the exposure.

Mode 1: `inspect` list

Calling: `printIns (const char *hdr = NULL);`

The `inspect` list of the observer is printed (possibly including the description of a `timeout` request). The layout of the list is exactly as for mode 0 with exception that the first item (the observer's output name) is absent.

The same information is produced by the "screen" version of the exposure.

18 DSD: the dynamic status display program

DSD is a stand-alone program implemented as a Java applet that can be used to monitor the execution of a SIDE program on-line.

18.1 Basic principles

A SIDE program communicates with DSD by receiving requests from the display program and responding with some information in a device-independent format. The SIDE program and DSD need not execute on the same machine; therefore, the information sent between these two parties is also machine-independent.

A typical unit of information comprising a number of logically related data items is a *window*. At the SIDE end, a window is represented by the following parameters:

- standard name of the exposed object (section 5.2)
- display mode of the exposure (an integer number—section 17.1)
- `Id` of the station to which the information displayed in the window is to be trimmed

The last attribute is generally optional and it may not apply to some windows (see section 17.5). In any case, if the station `Id` is absent, it is assumed that the window is global, i.e., unrelated to any particular station.

The above elements correspond to one mode of the "screen" exposure associated with the object, or rather with the object's type (section 17.4). Additionally, this exposure mode can be made station-relative. Whenever the contents of a window are to be updated (refreshed—section 17.4) the corresponding exposure mode is invoked.

The graphical layout of a window is of no interest to a SIDE program. The program only knows which data items are to be sent to DSD to produce a window. These data items are sent periodically, in response to a request from the display program specifying the window parameters. The actual layout of the information as displayed on the screen is then arranged by DSD—based on *window templates* (section 18.4).

From now on, we will assume that a *window* is something intentionally displayable described on the SIDE's end by the combination of the object's standard name, display mode, and (optional) station `Id`.

A window (we mean *window*) can be put into a step mode, in which the SIDE program will update the window after processing every event that "has something to do with the window contents." For example, for a station window, it means "every event that awakes one of the processes owned by the station."

At any moment during its execution, a SIDE program may be connected to DSD (we say that the display is active) or not. While the display is active, the SIDE program maintains the list of active windows whose contents are periodically sent (section 17.4) to DSD (by invoking the exposures of the exposed objects). Note that SIDE sends to the display program only raw data representing the information described by the object's exposure method.

DSD may wish to activate a new window, in which case the SIDE program adds it to the active list, or to deactivate one of the active windows, in which case the window is removed. The SIDE program never makes any decisions on its own as to which windows are to be active/inactive.

DSD is menu driven. Objects to be exposed and their modes (i.e., windows to be displayed) are selected from the current menu of objects. This menu can be navigated through based on the concept of the ownership hierarchy of objects maintained by the SIDE program. To simplify things, it is assumed that each object belongs to at most one other object. This assumption seems reasonable, although, e.g., a port naturally belongs to two objects: a station and a link. In this case, it is assumed that ports belong to stations, as this relationship is usually more relevant from the user's point of view.

Objects (the nodes of the object tree) are represented by their standard names, supplemented by base names and nicknames (section 5.2) wherever nicknames are defined. The display program uses this structure to create menus for locating individual exposable objects.

A typical operation performed by the user of DSD is a request to display a window associated with a specific object. Using the menu hierarchy of the program, the user gets to the

proper object and issues the request. This operation may involve descending or ascending through the ownership hierarchy of objects, which in turn may results in information exchange between DSD and the SIDE program. When the user finally makes a selection, the request is turned by DSD into a window description understandable by SIDE. This description is then sent to the SIDE program, which adds the requested window to its list of active windows. From now on, the information representing the window contents will be sent to the display program every `DisplayInterval` events (section 17.4).

18.2 The monitor

The SIDE program and DSD need not run on the same machine to be able to communicate. DSD does not even have to know the network address of the host on which the SIDE program is running. When the SIDE package is installed, the user has to designate one host to run the SIDE *monitor*—a daemon providing identification service for all SIDE programs to be run by the user. This host can be an arbitrary machine visible through the Internet, but usually the monitor host is located within the user's local-network domain. The user may want to run a web server on the same host, to make it possible for DSD (called from Java-capable browsers) to set up a TCP/IP connection to the monitor.[49] See section 19.1 for notes on starting up the monitor.

The purpose of the monitor is to keep track of all SIDE programs started by the user who owns the monitor. Whenever a SIDE program is invoked (on any machine visible to the monitor), it reports to the monitor; when the program terminates, the monitor also learns about this fact and removes the program's description from its directory. The following items of information are kept by the monitor for each active SIDE program:

- Internet name of the host on which the program is running
- program call line, i.e., the program name and the call arguments
- date and time when the program was started
- process `id` of the program
- description of the channel (socket) connecting the monitor to the program

The first four elements identify the run for the user; the last item makes it possible for the monitor to pass requests to the program.

[49] For security reasons, most browsers unconditionally restrict applet-initiated TCP/IP connections to the host running the web server that has delivered the applet page.

18.3 **Invoking** DSD

DSD is a Java applet that can be invoked, e.g., from any Java-capable web browser (see section 18.2). Its startup panel consists of two buttons: one to start the applet, the other to unconditionally terminate it and remove all its windows from the screen.

When started, DSD opens its main control panel (the so-called *root window*),[50] which looks as shown in figure 6.

This window consists of a menu bar at the top and two display areas. Initially, both display areas are empty and the single item selectable from the menu bar is `Get List` from menu `Navigate`.

The lower display area of the root window is the so-called *alert area*, where the applet displays messages addressed to the user. Some of those messages originate in DSD itself (they may—but don't have to—refer to errors or problems encountered by the applet); some others may be notes (e.g., sent by `displayNote`—section 17.4) or error diagnostics arriving from the SIDE program.

The upper display area (the *selection area*) is used to present the list of objects that the user can select from at the moment.

The only sensible thing that can be done at the beginning, right after the root window has been brought up, is to select `Get List` from the `Navigate` menu. This will ask DSD to try to connect to the monitor and acquire information about all SIDE programs currently running. This information will be displayed in a list (one program per line) in the selection area. If no SIDE programs are running at the moment, the message "`Nothing there`" will appear in the alert area and nothing more will happen.

If the list of programs displayed in the selection area is nonempty, the user can select one program from the list and choose an item from the `Navigate` menu. This menu now consists of the following two items:

`Connect`
: By hitting this item, the user asks DSD to establish a display session with the indicated program.

`Status`
: By hitting this item, the user asks DSD to display short status information about the indicated program.

In the latter case, the applet will poll the indicated program (using the monitor to mediate this communication) for its status. In response, the program will send the following information that will appear in the alert area of the root window:

[50] This shouldn't be confused with the `Root` process discussed in section 7.9.

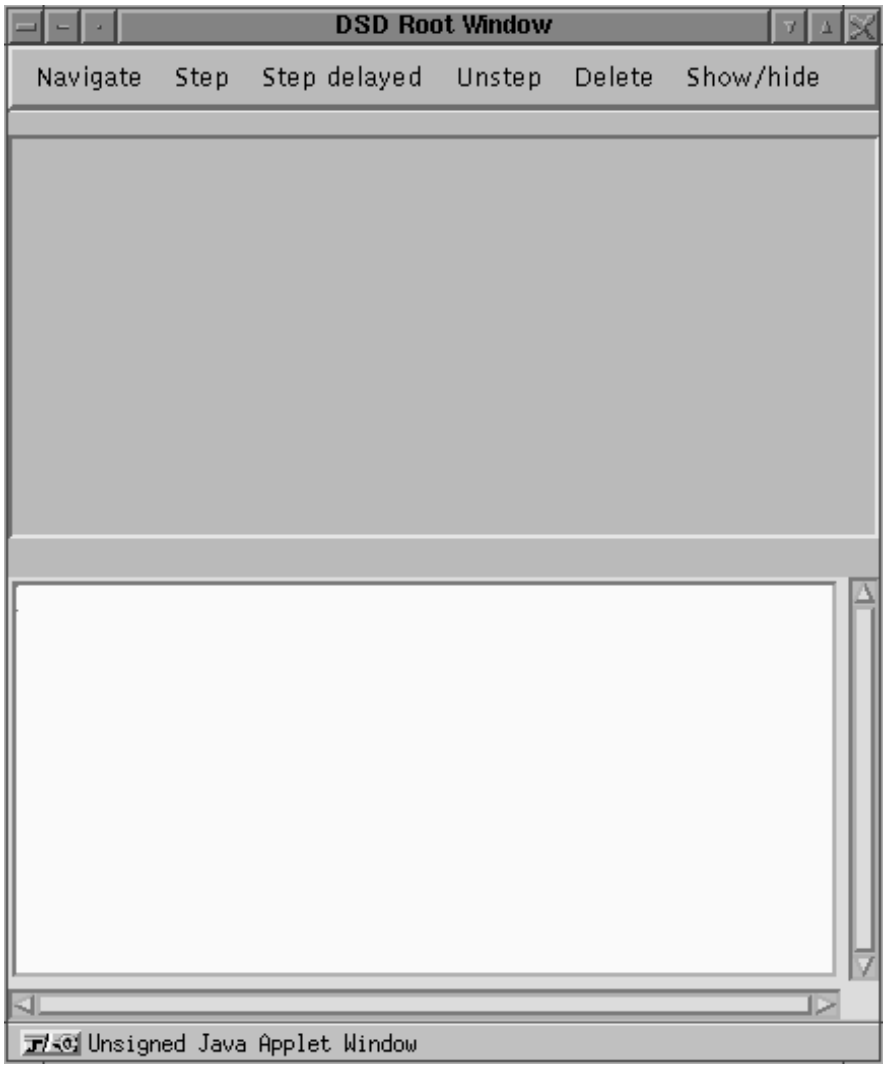

Figure 6: The main window of DSD.

- the number of messages received so far by the standard `Client` (section 11.2.3)
- the amount of virtual time (in *ITU*s) elapsed since the program was started
- the amount of CPU time (in seconds) used by the program

If the user selects `Connect` from the `Navigate` menu, the applet will try to set up a *display session* with the selected SIDE program.

During a display session, DSD presents in the selection area a list of objects whose dynamic exposures can be requested at the moment. The list starts up with an initial selection of objects sent by the SIDE program, and the `Navigate` menu provides a collection of operations for navigating through these objects and requesting their exposures (section 17). Depending on the current configuration of those exposures, other menus from the menu bar may also become useful.

Each object exposure requested by DSD appears in a separate window. As explained in section 17.1, such an exposure corresponds to a triplet: <object, display mode, station>. The raw exposure information sent by the SIDE program is organized by DSD into a window based on the *window template*.

18.4 Window templates

The material in this section may be useful for the user who would like to create non-standard windows and/or define non-standard exposable types (section 17.2). It also helps to understand how the information sent by the SIDE program to the DSD applet is turned into windows.

The layout of a window is determined by the *window template*, which is a textual pattern supplied in a *template file*. A standard template file (providing templates for all standard exposable objects of SIDE) is provided with the monitor (section 18.2). The monitor reads this file upon startup (section 19.1), and associates standard templates with all object selections sent by a SIDE program to the DSD applet. A SIDE program itself may use a private template file (section 19.3), which provides templates for its non-standard exposable objects and/or overrides (some of) the standard templates.

A template file consists of template definitions; each template describing one window layout associated with a combination of the object type, the display mode, and a designator that determines whether the window should be station-relative.

18.4.1 Template identifiers

The definition of a template starts with its identifier, which consists of up to three parts. Whenever the user requests a specific exposure, DSD searches the list of templates associated with the currently selectable objects, matching their identifiers to the parameters

of the requested exposure. These parameters specify the object, the display mode, and (optionally) the station to which the exposure should be related. Not surprisingly, the general format of a template identifier captures the three parameters:

name mode sflag

The first (mandatory) part of the template identifier should be a type name or a nickname. The template will be used to display windows associated with objects of the given type, or objects whose nickname (section 5.2) matches the given name.

The second argument specifies the display mode of the template. If this argument is missing, the default mode 0 is assumed.

The last argument, if present, tells whether the window can be made station-relative. If this argument is absent, the window is global, i.e., the template cannot be used to display a station-relative version of the window. An asterisk in this place defines a station-relative window template. Such a template can only be used, if the exposure request specifies a station (any station) to which the window is to be related. Two asterisks mean that the window represented by the template can (but does not have to) be station-relative. We say that such a template is "flexible."

When the user requests a new exposure (identifying a selectable object—section 18.5.2), DSD presents a menu listing all templates that match the object to be exposed. This menu consists of the following templates (appearing in this order):

- templates whose *name* matches the nickname of the object
- templates whose *name* matches the type name of the object
- templates whose *name* matches the base type name of the object (section 5.2)

Within each class of templates, the ones defined privately by the SIDE program take precedence over the standard ones (known by the monitor).

18.4.2 Template structure

The best way to describe how templates are defined is to look at a specific example. The following template resembles the standard template describing the layout of the mode 0 `Timer` window (see section 17.5.2):

```
Timer 0
"Wait requests (absolute)"
// Displays Timer wait requests coming from all stations
B012345678901234567890123456789012345678901234567890123456789012345+B
|~~~~~~~~Time~~~St~~~~Process/Idn~~~~~TState~~~~~~~~AI/Idn~~~~~Event~~~~~State| r
```

```
|%%%%%%%%%%%& %%% %%%%%%%%%%/&&& %%%%%%%%%% %%%%%%%%%%/&&& %%%%%%%%% %%%%%%%%%|
*                                                                                |
*                                                                                |
*8                                                                               |+
**                                                                               |
E0123456789012345678901234567890123456789012345678901234567890123456789012345678E
```

The second line of a template definition must be a quoted string describing briefly the window's purpose. This description will identify the template on the menu used to select the exposure mode. The description string must not contain newline characters.

Any text following the closing quote of the description string and preceding the first line starting with '`B`' or '`b`' is assumed to be a comment and is ignored. Similarly, all empty lines, i.e., those containing nothing except possible blanks and/or `tab` characters are always skipped.

The next relevant line of the template, starting with the letter '`B`', begins the description of the window layout. The first and the last non-blank characters from this line are removed and the length of whatever remains determines the width of the window. All characters other than '`+`' are ignored: they are just counted to the window width. A '`+`' character is also counted, but it has a special meaning: if present, it describes the initial width of the window (the column tagged with '`+`' is included). Only one '`+` may appear in the width line. In the above example, the '`+`' tag is superfluous: by default, the initial width is equal to the full width.

The line starting with '`B`' does not belong to the window frame and it does not count to the window height. Similarly, the closing line of the template (the one starting with '`E`') is not considered part of the window frame. In fact, only the first character (the letter '`E`') from the closing line is relevant and the rest of that line is ignored.

Note: '`b`' and '`e`' can be used instead of '`B`' and '`E`', respectively.

All nonempty lines, not beginning with '`X`' or '`x`' (see below), occurring between the starting and closing lines, describe the window contents. Each such a line should begin with '`|`' or '`*`'. A line beginning with '`|`' is a *format* line: it gives the layout of the corresponding window line. A layout line terminates with a matching vertical bar. The two bars do not count to the layout: they are just delimiters. The terminating bar can be followed by additional information, which does not belong directly to the layout, but may associate certain attributes with the items defined within the line.

18.4.3 Special characters

Within the layout portion of a template line, some characters have special meaning. All non-special characters stand for themselves, i.e., they will be displayed directly on the

positions they occupy within the template. Below we list the special characters and describe their meaning.

`~`
: This character stands for a virtual blank. It will be displayed as a blank in the window. Regular blanks are also displayed as blanks; however, two items separated by a sequence of regular blanks are considered separate, in the sense that each of them can have an individually definable set of attributes (section 18.4.8).

`%`
: A sequence of '`%`'s reserves space for one right-justified data item sent by the SIDE program to be displayed in the window.

`&`
: A sequence of '`&`'s reserves space for one left-justified data item sent by the SIDE program to be displayed in the window.

`@`
: This character is used to define a *region*, i.e., a semi-graphic subwindow (sections 17.3, 18.4.7).

Non-blank fields separated by a sequence of '`~`'s appear as different items, but they are viewed as a single item from the viewpoint of their attributes. Moreover, the separating sequence of virtual blanks receives the same attributes as the fields it separates. For example, the header line of the `Timer` template contains a number of items separated by virtual blanks. The letter `r` occurring after the terminating bar of the header line says that the first item of this line is to be displayed in *reverse video*. However, from the viewpoint of this attribute, all the items of the header line are viewed as a single item. Therefore, the entire header line will be displayed in reverse.

A sequence of '`%`'s reserves an area to contain a single data item sent to DSD by the SIDE program as part of the information representing the object's exposure (section 17.3). The type of this item is irrelevant from the viewpoint of the template declaration and will be determined upon arrival. The program will attempt to contain the item within its field. The item will be truncated, if it does not fit there, and right-justified, if some space is left. The only difference between '`&`' and '`%`' is that an item displayed in a field described by a sequence of '`&`'s is left-justified.

Data items arriving from the SIDE program are assigned to their fields in the order in which they arrive. The order of fields is from left to right within a line and then, when the line is filled, DSD switches to the next line. Superfluous data items are ignored.

18.4.4 Exception lines

A special character can be *escaped*, i.e., turned into a non-special one in a way that does not affect the visible length of the layout line. Namely, a layout line can be preceded by a line starting with the letter '`X`' or '`x`', for *exceptions*. A position marked in the exceptions line by '`|`' indicates that a special character occurring on that position in the next layout line is to be treated as a regular character.

Sometimes two fields that are supposed to contain two different data items arriving from the protocol program must be adjacent. A column marked with '`+`' in an exception line indicates the forced end of a field occurring in the next layout line. Any field crossing the position pointed to by the '`+`' will be split into two separate fields; the second field starting at the position following the position of the '`+`'.

A single layout line can be preceded by a number of exception lines, the multiple exception lines being cumulative.

18.4.5 Replication of layout lines

Sometimes the same layout line has to be replicated a number of times. In some cases this number is quite arbitrary. Such a situation occurs with the `Timer` window, which is actually a table consisting of rows with exactly the same format (the header line excepted).

A line starting with an asterisk stands for a replication of the last regular layout line. The asterisk can be followed by a positive integer number that says how many replications are needed. If the number is missing, 1 is assumed. Double asterisk means that the number of replications is undefined: the program is free to assume any nonnegative number.

A replication line need not contain any characters other than the asterisk possibly followed by a number, or two asterisks. However, if such a line is to contain an initial height indicator (see below), it should be closed by the vertical bar, as the height indicator can only appear after the bar.

18.4.6 Window height

The window height is determined by the number of the proper layout lines (excluding the `B`-line, the `E`-line, empty lines, and exception lines). A layout line whose "contents" part terminates with a vertical bar can include '`+`'—the initial height indicator, which should immediately follow the terminating bar. If the height indicator does not occur, the initial number of rows to be displayed in the window is equal to the number of the proper layout lines of the template.

A height indicator occurring at a replication line with replication count bigger than 1 is associated with the last line of this count. A height indicator appearing at a '`**`'-line or past this line is ignored.

If the window height is undefined and no default height indicator occurs within the defined part, the initial height of the window is determined by the number of proper layout lines preceding the first replication line with undefined count.

18.4.7 Regions

A region is a rectangular fragment of a window used to display graphic information. For an example of a region, let us look at the following template describing a window for displaying the contents of a random variable:

```
RVariable 0
"Full contents"
B------------------------B
|@                      @|
|                        |
|                        |
|                        |
|                        |
|                        |
|                        |
|                        |
|                        |
|                        |
|@                      @|
|------------------------| r
|Count:  %%%%%%%%%%%%%%%%| h n
|Min:    %%%%%%%%%%%%%%%%| h n
|Max:    %%%%%%%%%%%%%%%%| h n
|Mean:   %%%%%%%%%%%%%%%%| h n
|StDev:  %%%%%%%%%%%%%%%%|+ h n
X    |                    X
|CI95%:  %%%%%%%%%%%%%%%%| n
|        %%%%%%%%%%%%%%%%| n
**                       |
E------------------------E
```

The template defines one region delimited by the four '`@`' characters that mark the region's corners. The rows and columns containing these characters belong to the region. Except for the delimiting characters, the rest of the rectangle representing the region is ignored, i.e., any characters appearing there are treated as a comment.

A single template may define a number of regions. Regions must not overlap and they must be perfectly rectangular. The ordering of regions among themselves and other fields (important for the correct interpretation of the data items arriving from SIDE) is determined by the positions of their left top corners.

18.4.8 Field attributes

A layout line terminated by '|' can specify attributes to be associated with its individual display items. The optional specification of these attributes should follow the initial height indicator, if one is associated with the line.

At most one attribute specification can be associated with a single item, including regions. Specifications for different items are separated by blanks. The correspondence between specifications and items is determined by the order of their occurrence from left to right. Note that regions are represented for this purpose by their left top corners. Superfluous specifications are ignored. Items without specifications are assigned default attributes.

The following attributes are defined:

`n` normal display (the default)

`r` reverse video

`h` highlighted display

`b` blinking display

Four attributes are applicable to a region; their specification can be put at the region's first line, according to the same rules as for regular items. Note that parentheses must be used if more than one attribute specification is given.

Example

Below we give an example of a window template defining two regions.

```
Mytype 0 **
"Just an example"
B                     |                            B
!Station~%%%%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~| h
|@                    @  Number~of~packets: %%%%%%%%| r n b
!                       Number~of~bits:    %%%%%%%%| n b
|                       Acknowledgments:   %%%%%%%%| n h
|                       Retransmissions:   %%%%%%%%| n n
|                       Errors:            %%%%%%%%| n b
|                       Average~delay:     %%%%%%%%| n h
|                       ============================| r
|@                    @  @                 @  Status| h r
|=======Variance========                        A %%%%| r h n
|Hits:      %%%%%%%%%%%%                       B %%%%| h n h n
|Misses:    %%%%%%%%%%%%                       C %%%%| h n h n
|Total:     %%%%%%%%%%%%                       D %%%%| h n h n
|Successes: %%%%%%%%%%%%                       E %%%%| h n h n
```

```
|Failures:  %%%%%%%%%%%                        F %%%%| h n h n
|Total:     %%%%%%%%%%%  @                  @  G %%%%| h n h n
|Fairness:  %%%%%%%%%%%  =======Mean========  H %%%%| h n r h n
|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~| r
E                                                    E
```

Note that the first region (the upper left one) is going to be displayed in reverse, whereas the second one will be highlighted.

18.5 Requesting object exposures

As soon as a display session has been established, the SIDE program sends to DSD an initial list of exposable objects. These objects are elements of the tree of all exposable objects currently defined within the SIDE program. The user can browse through that tree using the `Navigate` menu from the root window.

18.5.1 The hierarchy of exposable objects

All exposable objects known to the SIDE program are logically organized into a tree reflecting their *ownership* relation. The root of this tree is the `System` station (section 6.1.2) which is assumed to own (directly or indirectly) all exposable objects. The ownership relation is illustrated in figure 7. The ordering of subnodes reflects the order in which the objects represented by the subnodes appear in the selection area of the root window (section 18.3).

The first object owned by `System` is `Kernel`—the root of the process hierarchy (section 7.9). Note that the father-child hierarchy of processes need not coincide with the ownership tree structure. The `System` station also owns all regular stations created by the protocol program. These stations constitute a flat layer: they are all direct descendants of `System`. Similarly, all links, radio channels, traffic patterns, `Client`, and `Timer` also belong to the `System` station.

A regular station owns its mailboxes, ports and transceivers. It also owns all regular processes, `RVariables`, and `EObjects` that were created by `Root` in the context of the station (section 7.3). The rules for determining the ownership of such objects created while the protocol is running are different (see below).

Each traffic pattern is the owner of its collection of random variables used to keep track of the standard performance measures (section 14.2.1).

The `Kernel` process owns all system processes as well as the user-defined `Root` process responsible for initialization (section 7.9). The system processes are invisible to the user, unless the SIDE program has been called with the '`-s`' option (sections 17.5.1, 19.3).

The `Root` process is the owner of all observers (section 16.3). It also owns all `RVariables`

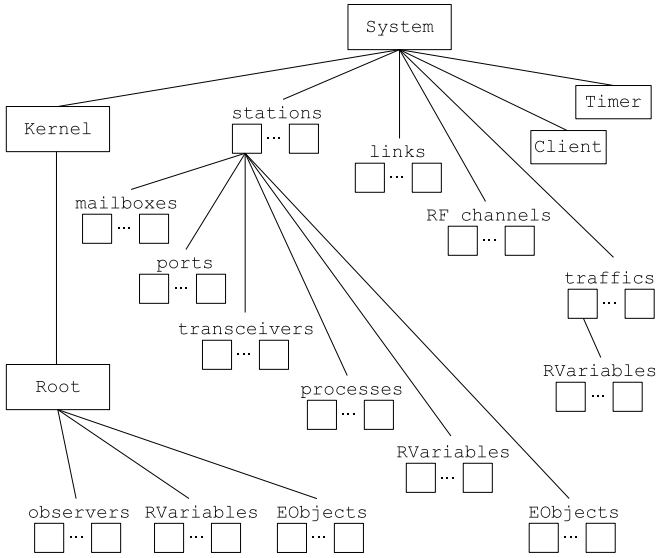

Figure 7: The ownership hierarchy of exposable objects.

and `EObjects` that were created by `Root`, but outside the context of any regular station.

There are three categories of exposable objects that can be created (and possibly destroyed) dynamically by the protocol program after the initialization phase, namely: processes, random variables, and `EObjects`. Any such object is owned by the process directly responsible for creating it. Thus, the ownership structure of processes created by other protocol processes coincides with the father-child relationship.

18.5.2 Navigating through the ownership tree

At any moment during a display session, the list of objects presented in the selection area consists of all the descendants of exactly one object in the ownership tree. The top item on that list (displayed in a different color) is the owner itself. Initially, immediately after the display session is established, the SIDE program sends to DSD the list of descendants of the `System` station, i.e., the root layer of the tree.

Each line in the selection area corresponds to one object and includes up to three names of the object (if they all exist and are different): the standard name, the base name, and the nickname. An object can be selected by clicking on its line and unselected by clicking once again. The following three items from the `Navigate` menu refer to the objects listed in the selection area:

Figure 8: A sample display dialog.

`Descend`
: DSD asks the SIDE program to send the list of descendants of the selected object. This way the user descends one level down in the ownership tree.

`Ascend`
: DSD asks the program to send the list of descendants of the object whose immediate descendant is the owner of the currently presented list of objects. This way the user ascends one level up in the ownership tree. Of course, it is impossible to ascend above the `System` station.

`Display`
: DSD opens a dialog to select the display mode and/or the station-relative status of an exposure for the selected object.

Each list of objects arriving from the SIDE program in response to a descend/ascend request is accompanied by a list of templates applicable to those objects. Those templates are selected by the SIDE program itself (if it uses a private template file) and/or by the monitor (using the standard set of templates).

In response to `Display`, DSD shows a dialog listing all the templates applicable to the selected object. For example, figure 8 presents the display dialog for the `Kernel` process (the first child item on the initial selection list).

The scrollable list includes all templates (their description strings—section 18.4.2) potentially applicable to the selected object. Note that each template implicitly determines a display mode. Thus, the only additional element that must be specified to complete the

exposure request is the station `Id`. This item can be typed in the text area (the rightmost area in the bottom panel).

The template description string starts with two characters that indicate the station-relative status of the template. Two periods mark an absolute template, i.e., one that cannot be related to a station. In such a case, the station field in the dialog is irrelevant and the template itself fully describes the exposure. For the remaining two cases ('`.*`' meaning "station-relative" and '`**`' meaning "flexible"—section 18.4.1), the field is meaningful. If it contains a legal station number (it must for a station-relative template), the exposure is station-relative; otherwise, it is absolute.

Having selected the template and (possibly) the station `Id`, the user may click `Display` to actually request the exposure. In response to this action, DSD will create a new window on the screen (based on the template—section 18.4) and notify the protocol program that a new exposure should be added to its internal list of active exposures. DSD refuses to create a new exposure if exactly the same exposure is already present.

18.6 Exposure windows

Typically, an exposure window contains a number of static components (extracted from the template—section 18.4.2) and some dynamic components that are filled dynamically by the protocol program (section 17.4). Note that the the protocol program only sends the raw data—the layout of the dynamic components within the window is determined by the template.

18.6.1 Basic operations

As soon as it is created, a new exposure appears in four menus selectable from the root window: `Step`, `Step delayed`, `Delete`, and `Show/hide`. The exposure is identified in the menu by the name of the exposed object,[51] followed by the exposure mode in square brackets. If the exposure is station-relative, this string is further followed by the station `Id` in parentheses.

By selecting the exposure from the `Show/hide` menu, the user toggles its "visible" status, i.e., the window becomes invisible if it was visible and vice versa. A window that becomes visible is automatically forced to the top. An invisible exposure window remains active and it continues receiving updates from the protocol program.

An exposure can be canceled by selecting it from the `Delete` menu. In response to this action, DSD removes the exposure's window from the screen and notifies the SIDE program to remove the exposures from its internal list.

An exposure window can be naturally resized. If the entire contents of the exposure cannot

[51] This is the nickname, if one is defined, or the standard name, otherwise.

fit into whatever frame is made available for the window,[52] the visible area can be scrolled. This is done by clicking and dragging in the visible area.

To reduce the amount of traffic between the protocol program and DSD, the protocol program maintains two counters for each active exposure. One of them tells the number of the first visible item in the current version of the exposure's window, the other specifies the number of the first item that is not going to show up. Whenever a window is resized or scrolled, DSD sends to the protocol program new values of the two counters. As only those items that actually appear in the window are sent in window updates, the scrolled-in area of a window may initially appear blank. It will become filled as soon as the protocol program learns about the new parameters of the window and begins sending the missing items.

18.6.2 Stepping

A protocol program can be put into the so-called *step mode* in which it stops on some (or all) events. An explicit user action is then required to continue execution.

The step mode is always requested for a specific exposure window. If a window is *stepped*, the SIDE program is halted whenever an event occurs that is somehow "related" to the exposure. At that moment, the contents of the window reflect the program state **after** the event has been processed.

Multiple exposures can be stepped at the same time. The occurrence of any event related to any of the stepped windows stops the protocol program.

The following rules describe what we mean by an "event related to the exposure":

- If the exposed object is a station, the related event is any event waking any process owned by the station.
- If the object is a process, the related event is any event waking the process. One exception is the `Kernel` process. It is assumed that all events are related to `Kernel`; thus, by stepping any `Kernel` exposure you will effectively monitor all events.
- If the object is a random variable or an exposable object of a user-defined type, the related event is any event related to the object's owner (in the sense of section 18.5.1).
- If the object is an observer, the related event is any event that results in restarting the observer.
- If the object is an *AI* not mentioned above, the related event is any waking event triggered by the *AI*.

[52] This is rather typical for windows with undefined height—section 18.4.6.

To put an exposure window immediately into the step mode the user should select it from the `Step` menu.[53] The frame of a stepped window changes its color from blue to red.

It is also possible to step an exposure at some later moment, by selecting it from the `Step delayed` menu. This may be useful for debugging: the user may decide to run the program at the full speed until it has reached some interesting stage. In response to `Step delayed`, DSD presents a dialog shown in figure 9.

Figure 9: The step dialog.

The step dialog allows the user to select the virtual time when the stepping should commence, or the number of events to be processed before it happens. This number can be either absolute or relative, i.e., represent an offset with respect to the current time or event count. Note that the first stepped event has to "have something to do" with the stepped exposure, so it may occur later (but no earlier) than the specified time.

The occurrence of a stepped event forces the protocol program to immediately (regardless of the value of `DisplayInterval`—section 17.4) send the update information for all active exposures and halt. The program will not resume its execution until the user selects `ADVANCE` from the `Step` menu. Then the program will continue until the next stepped event and halt again. Each time this happens, the message `STEP` is displayed in the alert area of the root window (section 18.3).

The user can cancel the stepping for one exposure—by selecting it from the `Unstep` menu, or globally for all stepped exposures—by selecting `ALL` from that menu. If this is done when the program is halted, `ADVANCE` is still required to let it go. Alternatively, the user may select `ALL+GO` from the `Unstep` menu to cancel all stepping and resume the normal execution of the program.

Note: Stepping affects the real-time character of the SIDE program and is not recom-

[53] Key shortcuts are available—see section 18.10.

mended for monitoring the behavior of a control program driving real equipment.

Note: When the display session is requested by the SIDE program (section 17.4), also when the program is called with `-d` (section 19.3), the program appears halted at the very beginning of the display session, although no exposure is formally being stepped. `ADVANCE` or `ALL(+GO)`—selected from the `Unstep`—menu can be used to resume the execution of such a program.

18.6.3 Segment attributes

The attribute pattern of segments (section 17.3) is interpreted by DSD in the following way (bits are numbered from zero starting from the less significant end):

Bits 0–1
: This field is interpreted as a number from 0 to 3 specifying the segment type. The following types are supported: 0–discrete points, 1–points connected with lines, 2–discrete stripes extending to the bottom of the region area.

Bits 2–5
: Thickness. This field is ignored for type–1 segments (i.e., points connected with lines). Otherwise, it represents the thickness of points or stripes, which, expressed in pixels, is equal to the numerical value of this field +2.

Bits 6–13
: Color. The numerical value of this field is used as an index into the internal color table of DSD and determines the color of the segment.

18.7 Other commands

In this section, we discuss briefly some other commands of DSD that are not directly related to exposure windows.

18.8 Display interval

As we mentioned in section 17.4, a SIDE program engaged in a display session with DSD sends exposure updates at intervals determined by the value of `DisplayInterval`. In the simulation (virtual) mode, this interval is expressed in events, and in the real (and also visualization) mode it is in milliseconds. In section 18.6.2, we noted one exception from this rule: when the program is stepped, it sends an update as soon as it is halted, regardless of the prescribed interval between regular updates.

The value of `DisplayInterval` can be changed by selecting `Reset Refresh Interval` from the `Navigate` menu. In response to this selection, DSD presents a simple dialog shown in figure 10.

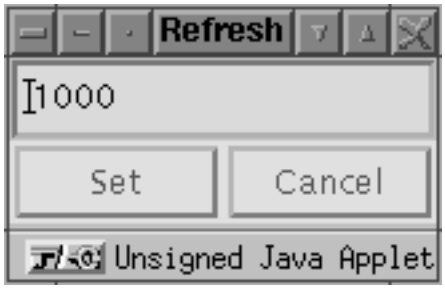

Figure 10: The display interval dialog.

The number entered in the text area represents the new requested value of the display interval, which is either in events or in milliseconds, depending on whether the program runs as a simulator, or executes in the real mode. The display interval cannot be reset in the visualization mode where it is coupled to the resync grain (section 7.10).

18.9 Disconnection and termination

By selecting `Disconnect` from the `Navigate` menu, the user terminates the display session. DSD notifies the SIDE program that it should not send any more updates and it should not expect any more commands from DSD.

All the windows of DSD remain on the screen until either the root window is canceled (by hitting the `Stop` button on the applet's startup panel) or a new selection is made from the `Navigate` menu (which at this stage can only be `Get List`—section 18.3).

It is possible to terminate the SIDE program from DSD—by selecting `Terminate` from the `Navigate` menu (during a display session). As this action is potentially dangerous, `Terminate` must be selected twice in a row to be effective.

18.10 Shortcuts

The following key strokes can be used as shortcuts for some DSD actions (normally selectable from menus). They are only effective if hit while within an exposure window.

`s` immediate step for this exposure (`Step`)

`S` delayed step for this exposure (`Step delayed`)

`u` unstep this exposure (`Unstep`)

`U` unstep this exposure and go

`g` continue until next step (`ADVANCE`)

The last key ("`g`") can be hit in any window, including the root window, and its semantics is always the same: "continue until the next stepped event." Upper case "`G`" and the space bar can be used instead of "`g`" and their meaning is exactly the same.

19 SIDE under UNIX and Windows

This section contains information on using SIDE in the UNIX environment and under CYGWIN on Windows 98, XP and Vista.

In principle, SIDE can be installed on any machine running a BSD or POSIX-compatible UNIX system, equipped with the GNU C++ compiler. The package has been also successfully installed on many machines and UNIX systems using several versions of C++ compilers. Under Windows, SIDE runs within the Cygwin environment available from http://www.cygwin.com/.

To take advantage of DSD, a Java environment is needed. This environment (and DSD) may be installed on the machine (or machines) on which SIDE experiments are to be run. Alternatively, DSD may be invoked from another machine and communicate with the experiment (or experiments) via the network.

19.1 Installation

The package comes as a collection of files organized into the directory structure presented in figure 11. The root directory of the package contains the following items:

`MANUAL`
: This directory contains the PDF version of this manual.

`SOURCES`
: This directory contains the source code of the package.

`Examples`
: This directory includes a number of sample protocols programmed in SIDE.

`README`
: This file contains the copyright notice and the log of changes introduced to the package since version 0.9.

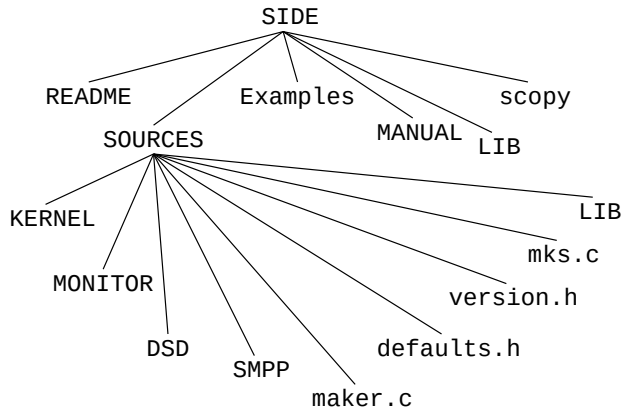

Figure 11: The structure of SIDE directories.

`scopy`
: This is a shell script used to create a copy of the package by linking some files to the original (see below).

Additionally, the root directory includes file `INSTALL` with brief installation instructions for the package.

The vital parts of the package are contained in directory `SOURCES` which consists of the following entries:

`KERNEL`
: This is a directory that contains the source code of the SIDE kernel. These files are used to create SIDE libraries that are configured with the user-supplied protocol files into stand-alone SIDE programs.

`MONITOR`
: This directory contains the source files of the monitor used to keep track of SIDE programs in execution and to connect them to DSD (section 18.2).

`DSD`
: This directory contains the source and Java-machine code of the display applet DSD (section 18).

`SMPP`
: This directory contains the source code of the preprocessor (SIDE compiler)

which is run before the C++ compiler to turn SIDE constructs into C++ code.

`LIB`
: This directory contains the source code of the SIDE i/o and XML library.

`maker.c`
: This is the source code of `maker`—the program used to configure the package (see below).

`defaults.h`
: This file contains definitions of default values to be used by the configuration program `maker`.

`version.h`
: This file contains the version number of the package and the code for recognizing the C++ compiler version.

`mks.c`
: This is the source code of `mks`—the driver for the SIDE compiler.

Another directory named `LIB` appears at the root level of the directory tree. It is initially empty, and its purpose is to contain binary versions of the kernel (section 19.2).

Before the package is installed, the user must make sure that the target machine is equipped with the tools needed to compile the package. For the UNIX environment, this is usually the case. Some old versions of the GNU C++ compiler may have problems with SIDE—we recommend at least version 3.4.4.[54] The version of DSD included with the package assumes JDK (Java Development Kit) version 1.2.x or later. This kit can be downloaded from http://www.javasoft.com/ or another site pointed from that page. DSD is precompiled to be 1.2 compliant. Note that the specification and execution system of SIDE can be used without DSD.

To install the package, the user should unpack the SIDE directory tree anywhere in his directory, e.g., by executing:

```
zcat sidexxx.tar.gz | tar -xvf -
```

where "`xxx`" stands for the current version of the package.

A single copy of the package can be shared by a number of users. In such a case, instead of copying all files, the user should perform the following two steps:

[54] It is always recommended to use the most recent available version of the compiler.

1. create the `SIDE` directory and move there
2. execute the `scopy` script by entering

   prfx`/scopy`

   where *prfx* is the path to the `scopy` script in the original `SIDE` directory

In consequence of the above operations, a copy of the `SIDE` directory tree will be built, but only those files that may change in different instances of the package will be physically duplicated. All other files, notably all source files of the SIDE kernel, monitor, and DSD, will be represented by links to the originals.

Then the user should move to the `SOURCES` subdirectory and compile the `maker` program by executing

```
g++ -o maker maker.cc
```

Having created `maker`, the user is ready to configure SIDE. This is done by executing

```
maker
```

and responding to a few questions asked by the program. In most cases, the default answers (indicated by `maker`) make sense and are recommended. The program asks about the following things:

- The name of the Java compiler. This information is only relevant and needed if the user wishes to recompile the DSD applet. Normally, the applet need not be recompiled because the precompiled Java code is platform-independent.
- The path to the `SOURCES` directory, i.e., to the directory containing the source files of the package. The default (which practically never should be changed) is the current directory.
- The path to the directory where binary libraries of the kernel modules will be kept. The default path is "`../LIB`", i.e., the libraries will be kept in subdirectory `LIB` in the root directory of the package. This default practically never needs to be changed.
- The maximum number of versions of the binary library to be kept in the library directory (see section 19.2). The default number is 25.
- The name of the directory containing `include` files for protocol programs. It is possible to have several simultaneous versions of such `include` libraries, which will be searched in the order of their specification. By default, the standard library in

`Examples/IncLib` is made available. It is needed by the example protocol programs in `Examples`, but it also contains many handy pieces that may be of general value, so it always makes sense to have it on the list. When you hit Enter (empty line) in response to this prompt, the standard `include` library will be the only library used. You can also enter paths to your personal libraries, one path per line, and then they will add to the list. You can eliminate the standard library (at any moment) by entering a line containing '`-`' (minus) as the only character. An empty line always terminates the list.

- The Internet name of the host running the SIDE monitor (section 18.2). Note that the monitor is needed to provide an interface between programs in SIDE and the DSD applet. Even if all three parties are to be run on the same machine, the host name of that machine must be specified here. By default, there is no monitor host, which means that the DSD applet will never be used. Enter `localhost` to be able to run all three components on the same local machine.

- The number of the monitor socket port. This should be an unused port number for implementing an `AF_INET` socket available to non-privileged users. The default number is 4442.

- The name of the SIDE compiler. The default name is "`mks`" and there is little sense in changing it.

- The directory where `mks` is to be put. The default is "`~/bin`" where "`~`" stands for the user's home directory. This directory should be included in the `PATH`.

- Whether the monitor is to be created. If the answer is "`yes`" (which is the default), `maker` will create the binary version of the monitor and put it into directory `MONITOR`.

When `maker` is done, it updates the contents of file `version.h` in directory `SOURCES` to reflect some of the user's selections.

The library directory (`LIB`) will be used by `mks` to keep binary versions of the kernel files (see section 19.2). This directory is initially empty. If it already exists and contains something, `maker` will erase all its contents. The maximum number of versions tells how many different versions of the binary files (corresponding to different configurations of `mks` parameters) can be kept simultaneously in the library directory. The bigger this number, the less time it takes on the average to generate a new simulation instance, at the expense of disk space (section 19.2). On Linux, one library version takes about 4MB of disk space.

The monitor should be created on one designated host. Preferably, it should be a workstation owned by the user. The executable version of the monitor is put into file `monitor` in directory `MONITOR`. The user should move there and run the monitor in the background by executing

```
monitor standard.t &
```

The parameter identifies the standard template file, which is supplied in the monitor's directory (see sections 18.2 and 18.4).

The monitor should be running all the time. It uses very little resources and its impact on the workstation's performance is completely negligible.

There is a small chance that the monitor will exit immediately with the following diagnosis:

```
port probably taken, rebuild with another socket port
```

which means that the monitor has failed to open an `AF_INET` domain socket. Such a problem may occur if the port number assigned by `maker` to the socket is in use. The user should execute `maker` once again selecting another port number. Typically, legal values are between 2000 and 32000. Note that selecting a new port generally involves a recompilation of the package.

The package includes two versions of the monitor: one implemented as a typical UNIX daemon that spawns a separate process for each non-trivial service, and the other implemented as a single process emulating multiple service threads. By default, the first version is used under UNIX and the second under CYGWIN.

With the operation of starting the monitor, the installation of the package is complete.

The DSD applet can be invoked from an applet viewer or from a web browser. File `index.htm` in directory `DSD` provides an `html` anchor to the applet. Note that most browsers will not let the applet connect to the monitor unless the web server delivering the applet runs on the same host as the monitor. Therefore, it may make sense to install a web server (e.g., Apache—http://www.apache.org/) on the monitor host. Alternatively, the monitor can be run on a host on which a web server has been already installed.

It may be better and more convenient to run DSD from `appletviewer` provided with the JDK distribution. To do it, the user should move to the applet directory (`SOURCES/DSD`) and execute

```
appletviewer index.htm
```

To let the applet freely connect to the monitor (even if both parties happen to run on the same host, the appletviewer may think that the session involves communication across the network), the user should select **unrestricted** access to the network and classes from the "properties" dialog of the appletviewer's menu.

Another minor problem related to the DSD applet is the different look of windows on different Java platforms. This problem mostly concerns exposure windows and is not very serious, as the window contents can be scrolled manually (section 18.6.1) if they appear misaligned. Ultimately, the user can edit file `DSD.java` in `SOURCES/DSD` and

modify the constants `TOPMARGIN` and `BOTTOMMARGIN` declared at the beginning of class `DisplayWindow`—to adjust the initial location of the window contents permanently for the preferred platform.

19.2 Creating executable programs in SIDE

A user-supplied protocol program in SIDE must be compiled and linked with the kernel files to create a stand-alone executable file.

The protocol program may consist of a number of C++ files, the name of a file ending with the suffix "`.cc`". All these files should be kept in one directory. They may `#include` some user-created "`.h`" files. The SIDE preprocessor (`smpp`) will automatically insert some standard include files (from `SOURCES/KERNEL`) in front of each protocol file.

To create an executable program for a given protocol source, the user should move to the directory containing the protocol files and execute `mks` (we assume that the default name of the program has not been changed at installation). The program accepts the following arguments (in an arbitrary order):

`-a`

All references to `assert` are turned into empty statements (see section 4.6). This may speed up the execution a little at the expense of some potential errors passing undetected. If this argument is not used, `asserts` are active.

`-b`

This argument must be followed by a single digit indicating the selected precision of type `BIG` (see section 3.3). The number should be separated from '`-b`' by a space. The default precision of `BIG` is 2.

`-d`

Type `DISTANCE` (section 3.6) will be defined as `BIG`. If this argument is not used, type `DISTANCE` is equivalent to `LONG`.

`-f`

The C++ compiler will be called with the optimization option. The default is "no optimization."

`-g`

This argument indicates that a "debug" version of the program is to be created. The C++ compiler will be called with the debugging option; simulator-level tracing (section 16.2) will be enabled.

An alternative variant of this option is `-G`. It is almost identical to `-g`, but it additionally disables catching signals (like SIGSEGV) by SIDE and makes sure to trigger a hard error (like SIGSEGV) when `excptn` (section 4.6) is called. This makes it easier to catch and diagnose some errors (including violations of internal assertions) in `gdb`.

`-i`

Type `BITCOUNT` (section 3.6) will be defined as `BIG`. If this argument is not used, type `BITCOUNT` is equivalent to `LONG`.

`-m`

Error checking for operations on `BIG` numbers, for precision higher than 1, will be suppressed (see section 3.4). Using this argument may slightly reduce the execution time at the price of possibly missing some arithmetic errors.

`-n`

Clock tolerance (section 8.3) will be set to 0, irrespective of the requirements of the protocol program. This argument effectively removes all code for randomizing time delays and makes all local clocks absolutely accurate.

`-p`

This option enables the three-argument variants of wait requests. A wait request may now specify an optional `order` parameter that assigns a priority to the wait request (sections 7.1, 7.4).

`-q`

This option enables the message queue size limit described in section 10.7. By default this feature is disabled.

`-z`

This option enables "faulty links" described in section 11.4. By default "faulty links" are disabled.

`-o`

This argument must be followed by a file name (separated from '`-o`' by a space). The specified file will contain the executable SIDE program. The default file name, assumed in the absence of '`-o`', is "`side`."

`-r`

Type `RATE` (section 3.6) will be defined as `BIG`. If this argument is not used, type `RATE` is equivalent to `LONG`.

`-u`

The standard client is permanently disabled, i.e., no traffic will be generated automatically, regardless of how the traffic patterns are defined (see section 10.6.2).

`-v`

Observers are disabled, i.e., they are never started and they will not monitor the protocol execution. This argument can be used to speed up the execution without having to remove the observers (see section 16.3).

`-L`

The code dealing with links and link activities is never compiled. It is assumed that the protocol (control) program does not use links and/or ports.

`-X`

The code dealing with radio channels and radio activities is never compiled. It is assumed that the protocol (control) program does not use radio channels and/or transceivers.

`-C`

The client code is never compiled and '`-L`' is implied. It is assumed that the program never uses the traffic generator, messages, and/or packets, as well as links and/or ports.

`-S`

The code dealing with random variables is never compiled and '`-C`' is implied. It is assumed that the program does not use random variables, the traffic generator, messages, and/or packets, as well as links and/or ports. This setup is often appropriate for complete control programs.

`-t`

The protocol files are recompiled even if their binary (`.o`) versions are up to date. This option can be used after a library file included by the protocol program has been modified, to force the recompilation of the protocol files, even if they appear unchanged.

`-8`

This option replaces the SIDE internal random number generator with a generator based on the `rand48` family. The `rand48`-based generator is slightly slower than the SIDE generator, but it has a longer cycle.

`-3`

This option selects the 3-D variant of node deployment model for transceivers and radio channels (section 6.4.1). It affects the headers of these `Transceiver` methods: `setup` (section 6.5.1), `setLocation`, `getLocation` (section 6.5.2), and the `getRange` method of `RFChannel` (section 6.5.2). An attempt to combine `-3` with `-X` results in an error.

`-R`

This option selects the real-time version of the kernel. By default, the protocol program executes in virtual time, i.e., as a simulator.

`-W`

This option compiles in the code for handling the visualization mode. It is needed, if the program ever wants to run in the virtualization mode (i.e., call `setResync`—section 7.10). Similar to `-R`, `-W` makes it possible to bind mailboxes, journal them, and drive them from journal files (section 13.5). Note that `-W` and `-R` cannot be specified together.

`-J`

This option compiles in the code for journaling (section 13.5) and is only effective together with `-R` or `-W`. The journal-related call arguments of SIDE programs (sections 13.5.1, 19.3) are not available without this option.

`-D`

Deterministic version. With this option, the nondeterminism of scheduling events is switched off (see section 7.4). The option implies `-n`, i.e., perfect clocks.

`-F`

This option selects the no-floating-point version of the resulting executable. It only makes sense for the real mode of SIDE and automatically forces `-R`, `-S`, and `-D`. One feature that does not work with `-D` is the `setEtu`, `setItu` pair (section 3.2).

`-I`

This argument must be followed by a path pointing to a directory which will be searched for `include` files. Multiple arguments of this form are permitted (up to 8), and the order of their occurrence determines the search order. The `include` directories specified this way are searched *after* those declared with `maker` (section 19.1).

`-V`

The program writes the package's version number to the standard output and exits.

In consequence of running `mks`, all protocol files from the current directory are compiled and merged with the kernel files. The program operates similarly to the standard UNIX utility `make`,[55] i.e., it only recompiles the files whose binary versions are not up to date. The resulting executable protocol program is written to the file "`side`," unless the user has changed this default with the '`-o`' argument (see above).

Formally, the kernel files of SIDE should be combined together with the user-supplied protocol files, then compiled, and finally linked into the executable program. Note that it would be quite expensive to keep all the possible binary versions of the standard files: each different configuration of `mks` arguments (except '`-o`' and '`-t`') needs a separate binary version of practically all files. Thus, only a number of a few most recently used versions are kept. They are stored in directory `LIB`; each version has a separate subdirectory there, labeled with a character string obtained from a combination of `mks` arguments. When the user requests creation of a protocol program with a combination of arguments that has no matching subdirectory in `LIB`, `mks` recompiles the kernel files and creates a new `LIB` subdirectory. If the total number of subdirectories exceeds the declared limit (section 19.1), the least recently used subdirectory is removed from `LIB` before the new one is created.

It is safe to run concurrently multiple copies of `mks` (within the domain of a single copy of the package) as long as these copies reference different input/output files. The program uses file locks to ensure the consistency of `LIB` subdirectories. Unfortunately, no file locking is available under CYGWIN.

19.3 Running the program

The binary file created by `mks` is a stand alone, executable program for modeling the behavior of the user-defined network and protocol. A number of arguments can be specified with the `side` call command. They are listed below.

`-r`

This argument should be followed by at least one and at most three nonnegative integer numbers defining the starting values of seeds for the random number generators. The three numbers correspond (in this order) to `SEED_traffic`, `SEED_delay` and `SEED_toss` (section 4.1). If not all three numbers are specified, only the first seed is (or the first two seeds are) set. A seed that is not explicitly initialized is assigned a default value, which is always the same.

[55] In fact, `make` is called at the last stage of processing.

`--`

A double dash terminates the argument list interpreted by the simulator's kernel. Any remaining arguments, if present, can be accessed by the user program via these two global variables:

```
int PCArgc;
const char **PCArgv;
```

interpreted as the standard arguments to `main`, i.e., `PCArgc` tells the number of arguments left, and `PCArgv` is an array of strings consisting of `PCArgc` elements.

`-d`

When this argument is used, the program will suspend its execution immediately after `Root` completes the initialization phase, and wait for a connection from DSD.

`-t`

This argument pertains to user-level tracing (section 16.1) and specifies the time interval during which the tracing should be on and/or the set of station to which it should be confined. The string following `-t` (after a space) should have one of the following forms:

start
start`-`*stop*
start`/`*stations*
start`-`*stop*`/`*stations*
`/`*stations*

where *start* specifies the first moment of modeled time when the tracing should become active, and *stop* indicates the first moment when the tracing should stop. If those numbers look like integers (i.e., they only contain digits), they are interpreted in *ITU*s. If a decimal point appears within *any* of the two numbers, then *both* of them are assumed to be in *ETU*s. A single number describes the starting time, with the ending time being undefined, i.e., equal to infinity. No numbers (like in the last variant in the above list) means no time restriction (*start* is zero, *stop* is infinity).

The optional list of station identifiers (section 6.1.2) separated by commas (not including blanks) appearing at the end of the argument (preceded by a slash) restricts the tracing to the specified set of station. This effectively means that a trace message will only be printed if `TheStation->getId ()` matches one of the indicated station `Id`.

Here are a few examples:

```
-t 1.0-2.0
-t 3-3.5/2
-t 1000000
-t /3,7,123
```

Note that in the second case the first number will be interpreted in *ETU*s, despite the fact that it has no decimal point, because the second number does have it.

One more handy feature is the option to terminate the string following `-t` with the letter `s` (`q` works as well), which has the effect of redefining the virtual time limit of the model (section 15) to the end boundary of the tracing interval. For example, with this sequence:

```
-t 12.33-47.99/0,1s
```

you tell the program to make the tracing effective from time 12.33 until 47.99 (expressed in *ETU*s) and trace two stations 0 and 1. At the end of the tracing interval, the program will terminate. This feature is useful for quickly producing narrow traces surrounding the place of an error.

The interpretation of the argument string of `-t` is postponed until the network has been built, i.e., until the program leaves the first state of `Root` (section 7.9). As we explain in section 16.1, this is required because the interpretation of a time bound expressed in *ETU*s can only be meaningful after the *ETU* has been defined by the program (section 3.2), which typically happens in the first state of `Root`. In particular, if the program issues a call to `setLimit` in the first state of `Root`, to define the virtual time limit for the run, that call will be overridden by the `s` flag of `-t`, but only if the end bound in the specification is finite. Otherwise, the `s` flag will be ignored. If the flag has been effective, i.e., once the time limit has been reset to the end bound of the tracing interval, any `setLimit` request from the program can only reduce the time limit (any attempts to extend will be ignored).

By default, if `-t` is absent, or if it is not followed by any specification, the user-level tracing is on for all time and for all stations. This simply means that all `trace` commands appearing in the program are always effective. Note, however, that an empty `-t` specification is different from is total absence, as it nullifies any `settraceTime` or `settraceStation` invocations issued by the program in the first state of `Root` (section 16.1).

`-T`

This argument is similar to `-t`, except that it refers to simulator-level tracing (section 16.2). It is only available if the simulator has been compiled with

the `-g` option of `mks` (section 19.2). Its format is exactly as for `-t`, with one extra option represented by `+`, which can appear at the end of the specification string (preceding or following the optional `s`). The time interval determines the range of simulator-level tracing, which consists in dumping to the output file detailed information about the states executed by the protocol processes (section 16.2). Here are a few examples:

```
-T 667888388880
-T /12+
-T 47.5-49.5/8,6
-T 0-999999999/1,3,5,7s+
```

The `+` option (in the second and last cases) selects the so-called "full tracing" (section 16.2), whereby the state information is accompanied by dumps of activities in all links and radio channels accessible via ports and transceivers of the current station.

In contrast to `-t`, the lack of `-T` means no tracing. A single time value selects the starting time with the ending time being unbounded. The appearance of `-T` not followed by any specification string is equivalent to '`-T 0`', i.e., all stations being traced all the time.

`-o`

If this argument is used, SIDE will write to the output file the description of the network configuration and traffic. This is done by calling

```
System->printTop ();
Client->printDef ();
```

(see sections 17.5.5, 17.5.14) immediately after the initialization phase.

`-c`

This argument affects the interpretation of the message number limit (see section 15.1).

`-u`

When this argument is used, the standard client is disabled and it will not generate any messages, regardless of how the traffic patterns are defined. Note that the client can be disabled permanently when the SIDE program is created (section 19.2).

`-s`

This argument indicates that information about internal (system) events should be included in exposures (section 17.5.1).

`-e`

When this argument is present, it will be illegal to (implicitly) terminate a process by failing to issue at least one wait request from its current state before putting the process to sleep (section 7.4). An occurrence of such a scenario will be treated as an error aborting the simulation.

`-k`

This argument must be followed by exactly two numbers: a nonnegative `double` number less than 1.0 and a small nonnegative integer number. These values will be used as the default clock tolerance parameters, i.e., `deviation` and `quality`, respectively (section 8.3).

`-M`

This argument must be followed by a file name. The file is expected to contain extra window templates needed by the protocol program (see section 18.4), which apply to non-standard exposures defined by the user and/or, possibly, override some of the standard templates of the monitor (section 18.2).

`-D`

This argument, which is only available in the real and visualization modes, and requires journaling to be compiled in (the `-J` option of `mks`—section 19.2) can be used to shift the real time of the program to a specified date/time. `-D` should be followed by a space followed in turn by the time specification. In its complete format, this specification looks as follows:

```
yr/mo/dy,ho:mi:sc
```

where `yy`, `mo`, `dy`, `ho`, `mi`, `sc` stand for the year, month, day, hour, minute, and second, respectively. Each field should consist of exactly two digits. The following abbreviations are permitted:

```
mo/dy,ho:mi:sc
dy,ho:mi:sc
ho:mi:sc
mi:sc
sc
yr/mo/dy
mo/dy
```

with the absent fields defaulting to their current values. One special date/time specification is '`J`' (the entire argument is "`-D J`") which shifts the time to the earliest creation time of a journal (section 13.5.2).

`-J`

This argument, which is only available in the real and visualization modes, with journaling compiled in (the `-J` option of `mks`—section 19.2), must be followed by the specification of a mailbox. It declares a mailbox to be journaled (section 13.5.1).

`-I`

This argument, which is only available in the real and visualization modes with journaling, must be followed by the specification of a mailbox. It declares a mailbox to be driven from the input section of a journal file (section 13.5.2).

`-O`

This argument, which is only available in the real and visualization modes with journaling, must be followed by the specification of a mailbox. It declares a mailbox to be driven from the output section of a journal file (section 13.5.2).

The first argument from the left that does not start with '`-`' and is not part of a compound argument starting with '`-`' is assumed to be the name of the input data file. Similarly, the second parameter with this property is interpreted as the output file name. If the input file name is "`.`" (period) or if no file name is specified at all, SIDE assumes that input data is to be read from the standard input. If no output file is specified or the output file name is "`.`" (period), the results will be written to the standard output. If you use "`+`" instead of the period, the output will be written to standard error. On some systems (like Cygwin) this is a more foolproof way of enforcing immediate flushing of the output data on every write than when writing to standard output.

Examples

```
side -r 11 12 datafile -d outfile
side  . out
side -k 0.000001 3 < data > out1234
```

In the first of the above examples `side` is called with `SEED_traffic` and `SEED_delay` initialized to 11 and 12, respectively. The simulation data is read from file `datafile`, the results are written to `outfile`. Before the protocol execution is started, the program will suspend itself awaiting connection from the display program. In the second example, the program is executed in real mode with deterministic event scheduling. The input data is read from the standard input and the results are written to file `out`. In the last example, the clock tolerance parameters are set to 0.000001 (`deviation`) and 3 (`quality`). The input data is read from file `data` and the simulation results are written to file `out1234`.