YATS – Yet Another Tiny Simulator
Programmer’s Manual for Version 0.3

Matthias Baumann
Dresden University of Technology
Communications Laboratory

1.0 Basic Principles

1.1 Creation of Network Objects

The input file is analyzed by the parser front end. When it recognizes a statement beginning with a network object class identifier, then it creates an object of the desired class. This is done by calling an intermediate routine provided by the class (see Section 5.1 on page 9). The object constructor normally is empty. Upon object creation the parser calls its init()-method. Here, the definition statement can be analyzed with the parser utility routines (see Section 9.0 on page 21). Inputs can be declared by calling the input()-methods of the generic class ino (see Section 7.1 on page 17). Output names can be parsed and registered by using the output()-methods provided by the classes in1out and inxout (see Section 7.2 on page 17). When init() returns control, the parser registers the object at the symbol manager, and expects the final ‘;’ of the definition statement.

1.2 Connection of Network Objects

After creation, the objects exist without any connections to each others, they are only registered at the symbol manager. Before running the first simulation (first statement Sim->Run), the connection process is invoked by the kernel. All objects registered are informed via the connect()-method (see Section 5.2 on page 10). The following then is normally done by the generic base classes in1out and inxout, so you never need to programm it when using one of them: The object looks for the succeeding object(s) obtained by the output()-calls during init(). It calls the symbol manager to get the object address and then asks the object itself to obtain the input number associated to the given input name (handle()-method, see Section 5.2 on page 10). This input key number lateron accompanies every data object passed to the successor via the rec()-method (see Section 5.3 on page 11). For the object data structure members filled during the connection process, see Section 7.6 on page 19.

1.3 Communication between Network Objects

1.3.1 Data Transfer

The standard way for data transfer is calling the rec()-method (see Section 5.3 on page 11) of the next object:

```
    retval = SucObjectPtr->rec(DataObjPtr, InputKey);
```

The method returns an information, whether the sending object now should stop to send. This allows the receiver to control the sender (see Start-Stop protocol, Section 6.4 on page 15). In the most “pure” ATM object classes, however, this kind of backpressure is not implemented since not according to reality.

Please note two very important conventions (see Section 5.3 on page 11):

- Rec() can only be used during the early phase of a time slot. You do not know your succeeding object, and most more complex network objects rely on receiving only in the early phase.

- Per time slot, you must not send more than one data item. Classes like delay lines and multiplexers simply cause a core dump when violating this rule.

The type of data transfered by rec() is unspecified, it only must be derived from the basic data object class “data”. The receiving object has to verify the data type when relying on a specific one. Please use the typecheck()-methods (see Section 7.5 on page 19) for this purpose. They do not perform an absolute check, but they also accept that the actual data type is derived from the desired one. Since they use an integer table lookup, they are very fast.

1.3.2 Export of Variables for Reading

Objects can export addresses of their internal variables to other objects like measurement devices or to the simulation kernel. Measurement devices simply ask for the address of a variable to display, and perform scanning and display on their own. For the simulation kernel (or more exactly the parser), exported variables automatically are available to read them in the input file, i.e. they can be used as commands in expressions. Although the parameter structure of the export()-method itself is rather complicated (see Section 5.6 on page 12), convinience methods provided by the class ino allow to export scalars and one-dimensional arryas (both integer or double) by writing one line of source code (see Section 7.4 on page 18).

1.3.3 “Special” Communication

For all cases, where possibilities or restrictions of the first two methods prohibit their application, you can use the universal method special() (see Section 5.7 on page 13). This method can be used for arbitrary data types (upon propper definition) and bidirectional data transfer. Currently it is used e.g. for r.n. transformation table export to source objects (see class Distribution), establishment of ABR connections (see ABR classes), and for writing the routing table of a demultiplexer.

1.4 Event Scheduler

In principle, the event scheduler only is designed to manage time-dependent operations within the same object. It is e.g. not possible to use the scheduler directly for delayed delivery of a data item to another object. In such cases, it is always necessary that the sending object sets a timer with the wished delay, and sends the data item itself by calling the rec()-method of the successor. This may appear as inconvinient, but in practice only very view object classes have to use this technique, or it is something to do anyway at the instance of sending.
To solve problems caused by the undefined processing order of simultaneous events, time slots are divided in an early and a late phase. Since data are only sent during the early phase, it is possible to implement e.g. the round-robin strategy in a multiplexer in the late phase.
The scheduler provides routines for single activation upon expiry of a specified time, and for activation during each time slot. The former kind of registration can be deleted as long as the event has not yet been activated.
For detailed information how to start and stop timers, see Section 8.0 on page 20.
Please pay also attention to the problems which may arise with the event scheduler (see Section 3.2 on page 7).

2.0 A Network Object Class: Commented C++ Code

The following example can be found as source code in the file src/user/vciCount.c. The example defines a network object class which only forwards cells on a given virtual channel. All other cells are terminated, i.e. the object acts as sink. Additionally, a “clock” is implemented (only in order to demonstrate scheduler usage). In the simulator input file, an object is declared e.g. with

```
VCICount xyz: VCI=13, TICK=10, OUT=sink;
```

The Object then counts cells on VCI 13, and forwards these cells to sink. The input name of the object is ‘xyz’, too. So we can have e.g. a source `CBRquelle src: ... OUT=xyz`. The counter for passed cells can be read with `x = xyz->TheCounter`. The clock is incremented with each TICK-th simulation time step, and the clock value can be read with `x = xyz->Clock`.

We start our description of the C++ code file with the class definition. The new network object class is derived from the generic base class ‘in1out’ which is described in more detail in Section 7.0 on page 16. The most important services of this base class are the definition of inputs and outputs, and the complete handling of the connection process.

```
#include “in1out.h”  // declaration of class in1out,
                     // also includes all other system headers
class vciCnt: public in1out {
typedef in1out baseclass; // often useful for recursion through
                          // the class hierarchy
public:
    vciCnt(): clockEvent(this, 0) {} // constructor
```

Events – clockEvent is defined further below – need to be initialized by the constructor for safety reasons. The first argument is the pointer to our object, the second is a key value allowing to distinguish between different events. Actually, we do not need this: we only have one event. Instead of defining clockEvent, we also could have reused an event structure which already is provided and initialized by the base class (its name is std_evt, see Section 7.3 on page 18).

```
    void init();         // evaluates the definition statement
    rec_typ REC(data *, int); // receives and forwards data.
               // REC is a macro normally expanding to ‘rec’.
               // REC is overloaded in debugging mode.
    void early(event *);      // is called by the scheduler
    int export(exp_typ *);    // allows read access to myCounter

    event clockEvent;   // an event structure for our clock tick
    int   myVCI;        // the VCI on which to count cells
    unsigned myCounter; // the counter itself
    tim_typ clockTick;  // tick, tim_typ is the type for times
    unsigned myClock;   // is incremented with each tick
};  // end of class definition
```

While the class definition normally should be placed in a header file, the rest has to reside in the normal C++ code file. Two macro calls establish the connection to the kernel:

```
    CONSTRUCTOR(VciCnt, vciCnt);
    USERCLASS("VCICount", VciCnt);
```

CONSTRUCTOR is a macro wich generates an interface routine (first argument, here VciCnt()) to be called by the parser when it wants to create a new object. The routine creates an object of class vciCnt (second argument) and returns the pointer to the parser.
The line containing USERCLASS(...) is copied by make config (see Section 3.2 on page 7) into the source code file "src/kernel/class.c". It declares the interface routine VciCnt() in the kernel, and says it has to be called, if a statement begins with "VCI-Count". During compilation of our C++ file, USERCLASS is an empty macro. Hence it does not generate code here.
If the parser now detects a statement beginning with "VCICount", then it calls the interface routine VciCnt(). Immediately afterwards it calls the method init() of the returned new object. At this instant, the input reading pointer still points to the beginning of the statement, i.e. to the class identifier.

```
    void vciCnt::init()
    {
        skip(CLASS);           // we skip the class ID
        name = read_id(NULL);// NULL: no keyword in front expected
```

Name is a member of the root class named '`root`'. If we do not set 'name' then it continues to point to "<name: unknown>" (initialized by the constructor of the base class `root`). Read_id() is a parser utility routine. It reads an identifier and checks that this ID is not yet used.

```
        skip(':');                // we require a ':' to follow
        myVCI = read_int("VCI"); // read the VCI
        skip(',');             // require a ','
```

We initialize myVCI by calling read_int() which again is a parser utility routine. It expects the given key word "VCI", followed by a '=', and an arbitrary expression evaluating to an integer value. The value is returned. If something goes wrong, the error message is automatically generated by the parser, and read_int() does not return (YATS is terminated).

```
        clockTick = read_int("TICK"); // read the clock tick
        if (clockTick < 1)
            syntax0("invalid TICK");
        alarme( &clockEvent, clockTick);
        skip(',');
```

The clock tick may not be zero. Besides that it would not be useful, it is not allowed to ask the scheduler for an activation after zero time steps. For performance reasons, this is, however, not tested by the scheduler interface routines. Syntax0() is an error message routine which prints the message and terminates YATS. Our error message is preceeded by the current line of the input text, the line number, and the current input file name. Additional parameters can be passed like for the printf() function family (see Section 13.0 on page 25).
Alarme() is a scheduler routine. The call requests that the early() method of our object shall be activated in the first half of the time step (`SimTime`+clockTick). The simulator clock `SimTime` is zero during init(). As first argument, we pass the event which we

have initialized with the constructor. This event structure is directly used by the scheduler, so it does not need to create and initialize an own data structure.
Remark: The price for the fairly fast operation of the scheduler are some subtle application rules, please pay attention for the details given in Section 4.0 on page 8 and Section 8.0 on page 20. The next two code lines define input and output of our object.

```
        output(“OUT”);
        stdinp();
```

Ouptut() is a method of class in1out. It reads the word OUT from the input text, followed by a ‘=’. Then an input name of another network object is expected and read. This name is remembered in an internal data structure of class in1out. During the connection process handled by the base classes, the object members suc (pointer to the successor) and shand (input key number of the successor), which we will use lateron, are initialized. For objects with more than one output, the base class inxout has to be used, it provides the necessary methods for multiple outputs (see Section 7.0 on page 16).
Stdinp() again is a method of the base class. The call indicates that our object has an input whichs name equals the object name. The input key number is 0. There are other methods to declare inputs with name extensions and other key values (see Section 7.0 on page 16).

```
        myCounter = 0; // this also could be part of the
                       // constructor vciCnt::vciCnt()
        myClock = 0;
    }   // end of vciCnt::init()
```

When init() returns, the parser expects the final ‘;’ of the definition statement, registers the new object at the symbol table (using the member ‘name’), and continues processing the input text. The first simulation statement Sim->Run ... invokes the connection process which is handled by our base classes. When the simulation then is running, we - hopefully - will receive data objects. A preceeding object calls our rec() method for that purpose. In order to allow debugging (see Section 4.1 on page 8), the macro `REC` is used instead of the real name `rec` for the method definition. In standard mode, `REC` expands to `rec`.

```
    rec_typ vciCnt::REC(data *pd, int) // REC is a macro normally
                                       // expanding to ‘rec’
    {
```

The pointer pd contains the pointer to the data object, the second argument is the input key number. Since we only have one input, we do not need to evaluate it. The pointer pd points to a general data object. If we want to access the VCI then we first have to ensure that the data object actually is of class cell or of a derived class (only cells have a VCI). This is done by typecheck(). The method provided by the base class checks the data type and generates an error message if the class does not fit (termination of YATS). In case we had more than one input, we would have had to use typecheck_i() (see Section 7.5 on page 19).
Now we are sure that pd points to a cell (or derived class), thus we can cast the pointer.

```
        typecheck(pd, CellType); // CellType is a key number
                                 // for cell objects
        cell *pc = (cell *) pd;
        if (pc->vci == myVCI) // without pc:
                              // if(((cell*)pd)->vci==myVCI)
        {   if ( ++myCounter == 0)
                errm1s(“%s: overflow of counter”, name);
            return suc->rec(pd, shand);
               // Do *not* use REC when calling a rec() method.
               // Otherwise, a syntax error appears in debug mode.
        }
```

Errm1s() resembles the syntax() function family, but it does not print any information related to the input text (see Section 13.0 on page 25). It remains to forward the data object to the successor. The pointer ‘suc’, and the input key number ‘shand’ have been initialized by in1out. The return value of the rec() method indicates whether a sender of data may continue transmission. We simply pass this return value to our preceeding object. If the cell does not carry the right VCI, it is deleted, and we return ContSend. This means that the preceeding object may continue to send in the next time step.

```
        else
        {   delete pd;
            return ContSend;
        }
    }   // end of vciCnt::rec()
```

When the first `clockTick` time steps have been processed, we are called in the first half of the time step (the scheduler divides each time step into two phases, during the first phase already early() is called). We increment the clock and register again. The argument of early() is just the event structure we have used for registration. This is sometimes useful since an event comprises a key value which allows to tell between different timers, if more than one are in usage. Here we only have one timer, so we do not need to look into the key. The important members in the event structure have not been changed, so we can directly reuse the event structure for the next registration.

```
    void vciCnt::early(event *)
    {
        if ( ++myClock == 0)
            errm1s(“%s: overflow of clock”, name);
        alarme( &clockEvent, clockTick); // register again
    }   // end of vciCnt::early()
```

The base class provides a counter named ‘counter’, together with methods allowing to read and reset this counter from the simulator input file (see Section 7.0 on page 16). Since we did not use ‘counter’, these methods have to be provided. In order to not over-complicate this example, only the method for reading is described here. For reseting, see the command() method specified in Section 5.5 on page 11.
Export() is a method used both by the parser and by other network objects (e.g. online displays), if they want to gain read access to a certain variable. The asking party specifies a string, and we have to reply with a pointer to the wished variable. By convention this pointer is only used for reading. Both request and reply information are part of an object of class exp_typ which is argument of export(). The method shall return TRUE, if the operation was successful, FALSE otherwise.

```
    int vciCnt::export(exp_typ *msg)
    {
        return baseclass::export(msg) ||
               intScalar(msg, “TheCounter”, (int *) &myCounter) ||
               intScalar(msg, “Clock”, (int *) &myClock);
    }   // end of  vciCnt::export()
```

First we try our base class (baseclass is the typedef in the class definition very above). If it was successful, then the evaluation of the or-expression is finished, and export() returns TRUE. Otherwise the result is determined by the calls of intScalar(). This is a method of the base class, other versions to export double values and arryas are available (see Section 7.4 on page 18). The arguments of the first call specify that the address of myCounter shall be returned via msg, if the name ‘TheCounter’ has been asked for. If this is not the case - the first intScalar() returned FALSE -, finally ‘Clock’ is tried.

The simplest way to include the new class into YATS is to play the code file (extension has to be .c) into the directory yats/src/user/. Typing ‘`make config`’ in the yats/bin/ directory generates entries for the new class in yats/bin/Makefile, and in the central configuration file yats/src/kernel/class.c. Typing ‘`make`’ afterwards compiles our source code file and the changed yats/src/kernel/class.c (see Section 3.0 on page 7), and links all together. Now the new class should be available for usage.

3.0 How to Add a New Network Object Class

Normally, each object class is coded in its own source code file. There are two ways to add a network class.

3.1 Configuration by Hand

To establish the connection to the simulation kernel by hand, three things have to be done:

1. Definition of an intermediate routine which can be called by the parser for object creation. This is easily done by including the macro
   `CONSTRUCTOR(NameOfIntermRout, InternalClassName);`
   into the source code file (see Section 5.1 on page 9).
2. Declaration of the new class in the source code file “class.c”. Here, the name of the intermediate routine is associated to the class name to be used in the simulation input file.
3. Include the name of the object code file into the list in the Makefile, and call make depend.

3.2 Automatic Configuration

All code files for user specific network object classes are collected in the directory
        src/user/
To install a network object class, simply put copies of or links to the corresponding .c and .h files into the src/user directory. Typing
        % make config
in the bin/ directory scans src/user/ for .c files and generates appropriate entries in the Makefile and in src/kernel/class.c. Your .c file has to comprise two things:

1. Definition of the intermediate routine which can be called by the parser for object creation. This is done as described in Section 3.1 on page 7.
2. The macro USERCLASS(NameAsString, NameOfIntermRout). NameAsString is the name of the object class to be used in the simulator input file. Make config searches the .c files in src/user for lines containing USERCLASS, and copies this lines into src/kernel/class.c.

For an example, see the file src/usr/example.c. To install this example, rename src/usr to src/user, and type ‘make config’ and ‘make’ in the bin/ directory.

4.0 Frequent Errors and Debugging

4.1 Usage of the Rec() Method

Remainder (see Section 5.3 on page 11):

- Rec() can only be used during the early phase of a time slot. You do not know your succeeding object, and most more complex network objects rely on receiving only in the early phase.
- Per time slot, you must not send more than one data item. Classes like delay lines and multiplexers simply cause a core dump when violating this rule.

A first way to debug violations of this two rules is to uncomment the definition of the pre-processor variable `RECEIVE_DEBUG` at the beginning of the file src/kernel/defs.h. This redefines all calls of rec() methods to calls of a check routine provided by the base class `root`. The base class performs the tests and generates error messages if necessary. Then the original rec() method is called. This requires that the rec() methods of all network object classes are declared (in the class definition) and defined (method body) with the macro name `REC(...)` instead of rec(...). The `RECEIVE_DEBUG` mode cannot be used together with classes which do not use the `REC()` macro, since this leads to incorrect C++ code. Thus, syntax errors during compilation with `RECEIVE_DEBUG` show the places where the change to `REC()` has been forgotten. The debugging mode slows the simulator down by typically 50 per cent.
Important: Calls of the rec() methods *must not* be changed. Otherwise, a syntax error is generated in debugging mode (Hence, these mistakes are easily found, too).

Note: The RECEIVE_DEBUG mode can generate wrong (i.e. too many) error messages, if the simulation time is reset between two runs (command Sim->ResetTime).

As a second way, you can use the network object class TypeCheck (see User’s Manual). This object can be included into each output line of your new object. It checks, that both rules are met, and produces appropriate error messages.

4.2 Event Handling

Remainder (see Section 8.0 on page 20):

- An event structure must not be used twice at the same time. It can be reused only if the corresponding timer has expired (the object has been activated), or if the timer has been stopped with unalarme() or unalarml().
- The time difference specified in alarme() / alarml() calls may not be lower than one. There are two exceptions of this rule: If the simulation is not running (e.g. during the init() method or when processing a command except Sim->Run), the time difference may be zero. Zero time difference also is legal during the early slot phase when setting a timer for the late phase of the same slot (alarml(..., 0)).

To find violations of this rules, you can define the C++ preprocessor variable `EVENT_DEBUG` in the source code file "defs.h" (uncomment the line at the beginning of the file). Afterwards you have to recompile. Now, tests are performed for all alarme(), alarml(), eache(), and eachl() calls. The additional code will slow down the simulator by approximately 10 to 15 per cent.

4.3 Locating Erroneous Network Objects

When developing more complex network object modules, core dumps unfortunately are used to happen. To locate the causing network module in a straightforward manner, you can turn on logging of object activation (calls of early()/late()). In the source code file "kernel/sim.c", there is a preprocessor variable EVENT_LOG. It is normally commented. Defining this variable causes the scheduler to print a log message for each activation occuring at a simulation time not smaller than EVENT_LOG. The message has the form

```
  currTime: objName->{early|late}{Evt|Tim}(begin ... end)
```

`Early` and `late` tell early and late time slot phase, `Evt` and `Tim` distinguish between event and time triggered activation (see Section 8.0 on page 20). The message is printed to standard output – except the final `end)` – before entering the object method. The tail of the message appears upon method completion.

With a debugger, you gain access to the erroneous object e.g. as follows. Define a global pointer variable (type as the class you want to debug) in the source code file of the object class. In the init() method, you initialize the global pointer, if the object name is the right one:

```
  if (strcmp(name, "xyz") == 0)  theDebugPtr = this;
```

Upon processing the declaration statement in the simulator input file, you can access the object members via `theDebugPtr`.

5.0 Network Object Classes and Their Methods

The root class for network objects is "root". This class declares the methods described below as virtual functions containing an error message.

5.1 Object Creation

The parser does not call directly the constructor of a class. Instead he invokes the intermediate routine declared in usr_classes() (source code file "class.c"). This routine is part of the source code file of the object class and creates the object. The registration of

the object at the symbol manager is done by the parser after calling the new object's init()-method. To define the intermediate routine, you can use the macro
`CONSTRUCTOR(NameOfIntermRout, InternalClassName);`
In case the object class shall not really create objects, the intermediate routine has to be programmed “by hand” and has to return a NULL pointer (normally, the new object's address, see e.g. signal.c).

In most cases, the constructor of a class can stay empty. At least, the constructor should not read from the input file, since the constructor is also executed when creating objects of derived classes. The definition statement is processed in the init()-method.

`void newClass::init(void);`

When called by the parser, the current input symbol is yet the class name. The name can be taken from `tval.nam`, but must not be freed. The symbol is skipped with `skip(CLASS)`. Reading the object name and evaluating of parameters is done via parser utility routines (see Section 9.0 on page 21). To ease derivation of other classes, the method addpars() (see next) is called at the place where derived classes should read their additional parameters. Classes with an own init()-method define addpars() as empty (or inherit the empty method from root).

`void newClass::addpars(void);`

A class derived from another one can reuse the init()-method of its base class. Reading of own parameters is done by overloading the addpars()-method. Addpars() should behave as follows:

```
void xyz::addpars(void)
{   // first call addpars() of the base class
    // baseclass is an appropriate typedef
    baseclass::addpars();
    // then read own parameters:
    xyzpar = read_int(“XYZPAR”);
    skip(‘,’);
}
```

By doing so, more than one derived class can reuse the init()-method. The class most “closed” to the base class first reads its parameters.

5.2 Connection Establishment

The methods for connection establishment mostly can be inherited from the generic classes `in1out` and `inxout`.

`void newClass::connect(void);`

This is a broadcast message from the kernel to cause connection establishment. All objects are defined, and the first `Sim->Run` statement has been reached. With
`root *find_obj(char *name);`
the succeeding objects can be found. Continuation see handle()-method.

`int newClass::handle(char *inpName, root *callgObject);`

The preceeding objects sends this message to obtain the input key number of the given input. The method should ensure that an input is not connected to more than one other object.

5.3 Receiving Data

```
rec_typ newClass::rec(data *dataPtr, int inputKey);
```

InputKey is the value returned by handle() during connection setup.
An object is allowed to call rec() of its successor
** only once a time slot*, and
** only during the early slot phase*.
Since the exact type of the data item is unknown, it has to be checked prior to access to data item members. See Section 6.2 on page 14.
Rec_typ is an enumeration type defined in the source code file “special.h”. The rec() return value can be evaluated by objects which want to be controlled by the succeeding object. The convention is, however, that a data item sent always remains at the receiver (or is lost there). The receiver only can protect against buffer overflow, when returning the stop value of rec_typ early enough. The Start-Stop protocol (see Section 6.4 on page 15) uses this mechanism.
Objects transparently passing data items (measurement devices, demultiplexer ...) should return the value given by the succeeding object to its precceeding one. Thus, these objects can be included into handshake lines.

REMARKS
In order to allow debugging of the usage of rec() (see Section 4.1 on page 8), the macro name `REC` has to be used for method declaration (in the class definition) and method definition (method body). Calls of rec() methods, however, *must not* use the `REC` macro. Otherwise, C++ syntax errors are generated when compiling in debug mode.

5.4 Activation by the Event Scheduler

The event scheduler divides each time slot into an early and a late phase. Dependent on the registration, one of the following methods is called for activation.

```
void newClass::early(event *eventPtr);
void newClass::late(event *eventPtr);
```

The event pointer is the address of the event structure used for registration. The event member eventPtr->key can be used to distinguish between different timers. The event structure may be reused immediately to register again. See also Section 8.0 on page 20.

5.5 Processing Commands

```
int newClass::command(char *cmdName, tok_typ *retVal);
```

A command shall be processed by the object. The command name is given in cmdName. If the command is known, it is carried out. Command arguments can be taken from the input file via parser utility routines (see Section 9.0 on page 21). Upon successfull completion, TRUE is returned by command(). In case the command is unknown, FALSE is returned.

The structure tok_typ contains the two members tok and val, where the latter is a union of an integer value, a double value, and a char pointer. In tok, the return type of the command is passed back, the val union is used to store the value (if any). The following values are possible for retVal->tok:

- NILVAR: The command does not return a value, and therefore can not be used in expressions
- IVAL: The return type is integer. The value is stored in retVal->val.i.
- DVAL: The return type is double. Value stored in retVal->val.d.
- SVAL: The return type is string. The pointer given in retVal->val.s has to specify a dynamic copy, since it is deleted in the kernel if necessary.

ATTENTION: do not return IVAR, SVAR, or DVAR.

Commands can be inherited as follows:
```
int xyz::command(char *s, tok_typ *v)
{   // first try base class
    if (baseclass::command(s, v))
        return TRUE;
    // then try it yourself and return TRUE or FALSE
}
```

5.6 Exporting Variable Addresses

To ease the rather complicated operations described next, the generic class `ino` provides convenience functions which allow to export a variable with one line of source code (see Section 7.4 on page 18).

```
int newClass::export(exp_typ *msg);
```

Either the parser or another object look for the address of a certain variable. The parser does so before trying the command()-method in order to evaluate the value of a command-like expression in the input file. Measurement objects as another example want to display the variable asked for.
The type exp_typ is defined in the source code file “special.h”. Msg->varname contains the name of the variable wanted. Msg->ninds specifies the number of indices given by a user, msg->indices[] specifies the indices itselfs. In case variable name, as well as number and ranges of indices are o.k., the field msg->addrtype has to be filled with the type of address (integer / double?, scalar / array?). The address itself then is written into the right filed of the address union in *msg. The method returns TRUE. In case the variable is an array, additionally the following fields have to be set. Msg->dimensions[] has to be initialized with the number of entries per dimension, and msg->displacements[] specifies for each dimension the displacement

between logical x value and physical index in the array pointer.
Export() returns FALSE if the variable is unknown or an error occured. Inheritance is implemented applying the same scheme as shown for command().

5.7 Miscellaneous

`char *newClass::special(specmsg *msg, char *caller);`

This is the method for “all cases”. For every kind of usage, a class containing the needed members has to be derived from class specmsg defined in “special.h”. Specmsg comprises a field type to tell different message types from each others. An object implementing the special()-method always first has to check this field. For every application, an extra value has to be included into the enumeration specmsg_typ (“special.h”). Special() should return the NULL pointer in case all is o.k. If an error occured, an error description should be returned. If this is not possible or too complicated, an error message can be launched by the object itself. For this purpose, the name of the calling object is given as second argument.
Find application examples in “demux.c” (writing the routing table) or in “distrib.c” (exporting the r.n. transformation table).

`void newClass::restim(void);`

Message from the kernel that the simulation clock is to reset (the command read: Sim->ResetTime). This method only has to be implemented by classes which store times internally (see e.g. “line.c” and “leakyb.c”). Events which have been registered at the scheduler, are corrected automatically. The method is predefined by class root as no-op (no error message).

6.0 Data Object Classes, Start-Stop Protocol

6.1 Definition of Data Object Classes

The rec()-method is defined for the class data which is the basic class for all data objects. All data object classes and their derivation relations are defined in the source code file “data.h”. The derivation of data classes and a fast mechanism for run-time data type checking allow that

- objects processing a certain data type can also process all data types derived from the intended one,
- error messages are generated, however, if incomming data items do not contain the necessary members (i.e. they are not derived from the intended data class).

In parallel, destructors of all data classes are virtual. Thus, the right destructor is found for an object regardless of the type of pointer used in the delete statement. The combination of run-time type checking and late binding of destructors costs - even for very simple models - not more than 10 % speed.
Unfortunately, derivation relations can not (yet) be examined automatically. If a new data class is added, then therefore a couple of things has to be done in “data.h” and “data.c”:

1. Add a new value to the enum dat_typ.

2. The following macros have to be part of each class definition:
   - BASECLASS(base_class_name): declaration of the base class
   - CLASS_KEY(dat_typ_value): specification of the dat_typ value associated to the new class
   - NEW_DELETE(number_of_objects_allocated_together): defines inline new and delete operators. They decrease the simulation time by approx. 30% and initialize the ‘type’ object members.
   - CLONE(class_name): defines the method clone() which produces a copy of a data object of unknown type (see Section 6.3 on page 14).
3. Define the static class member (data *new_class::pool) in “data.c”, it has been declared by NEW_DELETE().
4. Register the new class in data_classes() (very below in “data.h”). The macro DATA_CLASS(internal_class_name, external_name) uses the information of previous BASECLASS() and CLASS_KEY() statements to examine the derivation relationships.
5. Do not forget to define the method new_class::pdu_len() which returns the real-world size of a data object.

ATTENTION
If a data object is not created via the new operator, it cannot be sent to other network objects. Additionally, the type field of the data object is not initialized automatically, since this normally should not be done by the constructor (otherwise it would be changed several times during calling all constructors of the object and its bas classes). For objects created via new, the type field is initialised once by the memory allocation routine, and then never again.

6.2 Run-Time Data Type Checking

The run-time data type check is performed by the methods typecheck(), typecheck_i(), and typequery() which are provided by the generic class ino (see Section 7.5 on page 19). During initialisation of the simulator, an integer table reflecting the derivation relations is computed. The type check methods then perform a simple lookup in this table.

6.3 Embedding of Data Objects

Certain network objects have to encapsulate data objects, e.g. to segment them, to transport them in own data objects, and to reconstruct them at the receiver end of a peer-to-peer connection. To organize this process transparently, the following interface has been defined for class data:

```
class data {
     data *embedded; // ‘hook’ for other data objects
     data()
     {   embedded = NULL; }
     virtual ~data()
         // delete possibly embedded data objects
     {   if (embedded) delete embedded; }
     virtual size_t pdu_len() { return xyz; }
         // return the “real-world” size of the object
};
```

A data object which shall be transported via another object class is mounted into the `embeded` pointer of one of the lower-layer data objects (e.g. the last cell of the stream of AAL 5 cells representing a frame). If the critical lower-layer data item is lost and therefore deleted in the network, then the higher-layer object is deleted automatically (`embedded` is non-NULL, so a `delete` is performed). Initially, the `embedded` pointer is set to NULL by the constructor of class `data`. The method `pdu_len()` has to return the “real-world” size of the object in bytes. This allows a lower-layer network object to figure out, how many lower-layer data objects have to be sent.

When a receiving network object expects a mounted data object, it should test `embedded` on non-NULL before it passes the pointer to a successor.
Important:
It is essential that the `embedded` pointer is reset to NULL, if the transported data object is taken from the lower-layer data object. Otherwise, the higher-layer object is deleted twice: at the receiver of the object and together with the lateron deleted lower-layer object.

```
// data *pd contains the lower-layer object
if (pd->embedded)
{    // extract and forward higher-layer object
     suc->rec(pd->embedded, shand);
     pd->embedded = NULL;// !! important !!
}
else // error message: we expected sth embedded
```

For an example, see src/tcpip/aal5send.c and src/tcpip/aal5rec.c.
If a data item has to be duplicated for some reason, the method clone() should help.

```
data *p2;            // p1 comes from somewhere
p2 = p1->clone();    // p1 returns a new object with a copy
                     // of itself. We don’t need to know
                     // the class.
```

6.4 Start-Stop Protocol

To realize a loss free transmission of data objects, e.g. between objects implementing higher layer protocols, the Start-Stop protocol has been defined.

- In case the rec()-method of the data receiver returns StopSend, the data sender must not continue to send.
- Data items sent during the stop state of the data sender are dropped by the data receiver, and rec() again returns StopSend.
- The stopped data sender is started again by sending him a data item on the control input.

- The data receiver may send the start data item in the same slot when buffer becomes available.
- Upon receiption of the start signal, the data sender has to postpone sending, if data have been send during the current time slot (this is possible!). The throughput is not decreased, if this is ensured by a general delay by one time slot.
- The data sender starts in the state of sending allowed.

This basic version has been extended for future usage together with higher-layer protocols:

- ContSend is defined by -2.
- StopSend equals -1.
- All other values (only >= 0) have the following meaning: We must stop to send, and *logically* a part of the data item has been rejected. This means, that the *physical* data object remains at the receiver, but the receiver has only accepted the amount of associated data given by the return value. This may include that the receiver accepted only a part of the data, or even nothing (recRetVal == 0). In case all data have been accepted, but we shall stop to send, the receiver *must* return StopSend as in the basic protocol.

Network objects which only can handle the two basic values ContSend and StopSend have to check the return value of the rec() call. A utility method is provided by class ino: `void ino::chkStartStop(rec_typ recRetVal);`
It evaluates the given value and generates an error message in case neither ContSend nor StopSend has been returned.

The basic protocol currently is implemented by GmdpStop, ShapCtrl, AbrSrc, and AbrSink. Next candidates are the anticipated AAL5 and the TCP/IP implementations.

7.0 Generic Network Object Classes

There are three generic classes which hide details of connection establishment more or less completely.

- `class ino: public root`
  Network object with arbitrarily many (also zero) inputs. Is not used directly, but is the base class of `in1out` and `inxout`
- `class in1out: public ino`
  Network object with inputs and at most one output.
- `class inxout: public ino`
  Network object with inputs and arbitrarily many outputs.

They provide the following services:

- management of inputs and outputs, parsing of output names
- methods for connection establishment (connect() and handle())
- export()-method for reading a universal counter
- convenience functions to apply in export()

- command()-method for resetting the counter
- methods for run-time data type checking
- provision and initialisation of an event structure

7.1 Definition of Inputs

The three following methods can be called from init() or addpars(), the can be used arbitrarily often, and mixed together with calls to define outputs. If none of the input()-methods is called, then the network object has no inputs.

`void ino::stdinp(void);`

> Definition of an input whichs name equals the network object's name. The input key number lateron returned by handle() is 0.

`void ino::input(char *ext, int key);`

> An input with the name `'objName'->'*ext'`. The input key number is key. In case `ext == NULL`, the input name equals the object name.

`void ino::inputs(char *ext, int ninp, int displ);`

> Define a set of ninp inputs. The inputs have the names `'objName'->'*ext'[no]`, where no is ranging from 1 to ninp. The input keys are (no + displ). If e.g. the first input shall have the key 0, then displ is -1.

Handle() lateron generates error messages, when input names different from the defined ones are requested. Multiple connections also cause error messages, inputs staying unconnected are not reported (they do not cause errors).

7.2 Definition of Outputs

The methods can be called from init() or addpars(), they can be mixed with input()-calls. They can be called arbitrarily often (exception: in1out::output() only once).

`void in1out::output(char *keyw);`

> Definition of an output. The output name is read from the input text. In case keyw is not the NULL pointer, the keyword followed by a '=' sign has to preceed the output name. After connection establishment by connect(), the object element `root *suc` is initialized with the pointer to the next network object, `int shand` contains the input key number associated to the wished input. In case the output()-method is not called, the network object has no output, and suc remains NULL.

`void inxout::output(char *keyw, int idx);`

> Like in1out::output(). Additionally, the index which the successor shall have in the arrays root sucs[] and int shands[], is specified in idx. To pass lateron data on output idx: `sucs[idx]->rec(pdata, shands[idx])`.

`void inxout::outputs(char *keyw, int nout, int displ);`

Define a set of outputs. If keyw != NULL, the keyword and a ‘=’ are expected as introduction. The output indices in sucs[] and shands[] are (no + displ), where no ranges from 1 to nout (example: to generate indices 0 ... (nout-1), specify displ=-1). The output names are expected according to one of three possibilities:

```
‘(‘ variable ‘:’ nameTemplateContainingVariable ‘)’
```

The variable has to be declared in advance and is counted from 1 to nout to obtain the names.

```
‘(‘ variable ‘=’ lo ‘to’ up ‘:’ Template ‘,’ ... ‘)’
```

The variable has to be declared in advance, and a number of ranges with different name templates can be specified. Ranges can be abbreviated. See User’s Manual, section about Demultiplexer.

```
output1 ‘,’ ... ‘,’ outputNout
```

The names are listed completely.

7.3 Command() Method, Event Structure

```
unsigned ino::counter;
int ino::command(char *, tok_typ *);
```

The class ino contains a counter which is initialized with zero by the constructor. With the command objName->ResCount, the counter can be reset (implemented by command()). Reading counter is provided by the export()-method (see next).

```
even ino::std_evt;
```

This is a predefined event which is part of `ino`. The event key number is 0.

7.4 Export() Method, Convenience Functions

```
int ino::export(exp_typ *);
```

The address of the Variable objName->Count is exported, no indices are allowed. The address type is exp_typ::IntScalar (see “special.h”).

The following methods allow an uncomplicated export of integer and double scalars and one- and two-dimensional arrays.
For application examples, see e.g. mux::export() in the source code file “mux.c”.

```
int ino::intScalar(exp_typ *msg, char *nam, int *ptr);
int ino::doubleScalar(exp_typ *msg, char *nam,
                      double *ptr);
```

In case the variable name given in the msg structure equals nam and no indices have been specified, the address ptr is filled into *msg. The address type then is filled with exp_typ::IntScalar or exp_typ::DoubleScalar, and the method returns TRUE. If the conditions are not met, FALSE is returned.

```
int ino::intArray1(exp_typ *msg, char *nam,
                   int *ptr, int dim, int displ);
int ino::doubleArray1(exp_typ *msg, char *nam,
                   double *ptr, int dim, int displ);
```

Like intScalar() / doubleScalar(), but additionally the length of the array is noted in the message. The value displ specifies the differenc between logical value and index in the array. Shall an array e.g. possess the logical indices 1 ... dim (in the simulation input file or on a measurent display), then displ has to be set to 1. The address types filled into the message are exp_typ::IntArray1 or exp_typ::DoubleArray1.

```
int ino::intArray2(exp_typ *msg, char *nam,
                   int **ptr, int dim1, int displ1,
                   int dim2, int displ2);
int ino::doubleArray2(exp_typ *msg, char *nam,
                   double **ptr, int dim1, int displ1,
                   int dim2, int displ2);
```

These two methods allow to export two-dimensional arrays of integer and double values. For both dimensions, `dim` and `displ` have to be given (same meaning as for the one-dimensional versions). For an application, see file ‘src/misc/meas3.c’.

7.5 Run-Time Data Type Checking

Network objects expecting certain data object types from their preceeding objects have to check the actual type of arriving data items. This is because the input file syntax can not check whether objects which incompatible data object types are connected. The following inline-methods (defined in “ino.h”) are provided.

```
void ino::typecheck_i(data *pd, dat_typ typ, int key);
```

Using a table created during simulator initialisation, it is determined whether the incomming data item is of the requested type or of a derived type. If neither of them, then an error message is generated. Key is the input key number which has been passed to the rec()-method. It is used to derive input name and name of preceeding object for an error message.

```
void ino::typecheck(data *pd, dat_typ typ);
```

Simpler version of typecheck_i(). Can be used for network objects only possessing one input. If the method is used for objects with more than one input, then the type check is perfomed accuratly, but an error message will probably contain the wrong input and preceeding object name.

```
int ino::typequery(data *pd, dat_typ typ);
```

This is a “soft” version of typecheck(). The method returns TRUE if the data item has the right type, FALSE otherwise. No error message is generated. Using this method, different data types can be distinguished without additional flags.

7.6 Connection Establishment

These methods are called by the kernel prior to the first simulation run.

```
void in1out::connect(void);
void inxout::connect(void);
```

The information obtained by output()-calls is used to connect to the successors. After connection establishment, the following data object members (of the own object) are initialized:

Class in1out:

```
root *suc; // pointer to the successor
int shand; // input key value to pass together with
           // every data item
```

Class inxout:

Similar, but the values now are arrays. The array length is derived from the largest index whished by an output()-call.

```
root **sucs;
int *shands;
```

8.0 Event Scheduler

The event scheduler divides each time slot into an early and a late phase. Therefore, problems arising from simultaneity can be solved inside of the network objects without making the scheduler large and slow (example: “mux.c”). Network objects can register for single activation upon expiry of a specifeid time (“event triggered”), or they can register for activation during each time slot (“time triggered”). During each slot phase, first the event triggered, and then the time triggered events are activated. The global variable `int TimeType` specifies the current slot phase, values are EARLY and LATE. The global variable `tim_typ SimTime` displays the current time.

To avoid repeated allocations and deallocations of event management structures in the kernel, the event structures are directly associated to the network objects. Fix event entries like the pointer to the network object and the event key (to distinguish different timers set by an object) are initialized once at network object creation. When registering at the scheduler, a network object passes the address of the event structure to the scheduler. The latter one directly uses the structure in the event lists. The consequence is that an event structure must not be reused for starting another timer prior to

1. Event activation by the kernel. The object method called by the kernel (early() or late()), however, can reuse the event immediately for e new registration. Or:
2. Deleting the registration via unalarme() / unalarml().

For the most applications, this will not cause any inconvenience: They use the scheme:

1. Register for activation
2. Return control
3. Activation by Kernel, do something (send a cell ...)
4. Register again for activation (according to an IAT ...)

If more than one timer has to be used (for protocols, for ABR, for combination between event and time triggered processing ...), than the appropriate number of events has to be part of the object. The very last possibility is: create an event with operator new, fill the object and key members, use it for registration, return control; when early() / late is called, then the pointer to the event is passed, so you can delete it.

8.1 Event Triggered Activation

```
void alarme(event *evt, tim_typ delta); // register for
                                        // for early phase
void alarml(event *evt, tim_typ delta); // for late phase
```

The network object specified by evt->obj is activated at the time (SimTime + delta). When calling early() or late() of the object, the event pointer evt is passed back. The time difference delta has to be larger than 0. Otherwise the object will be activated never again. There are two exceptions, where delta may be zero:

- During the early slot phase, it is possible to register for activation in the late phase of the same slot (alarml(..., 0)).
- When no simulation is running, e.g. during initialisation or when perfoming a command(), delta may be zero. This then means that the object will be activated during the first time slot of the next simulation run.

8.2 Time Triggered Activation

```
void eache(event *); // call every early slot phase
void eachl(event *); // late slot phase
```

The network object is activated during each time slot.

8.3 Deleting Event Triggered Activations

An event registered with alarme() / alarml() can be deleted as long as the event has not yet been activated. Since deleting the event requires some search operations in the event lists, these routines are relativly expensive.

```
void unalarme(event *); // when registered with alarme()
void unalarml(event *); // when registered with alarml()
```

9.0 Parser Utility Routines

For an easy evaluation of network object parameters, a set of routines is provided. These routines expect a certain input text and automatically generate error messages, if the desired text can not be found. Upon successful completion, the input file reading pointer is shifted appropriately.
The current input symbol type (i.e. token) always is stored in the global variable
`int token;`
The token types are defined in “defs.h”. The token attributes (name, value ...) are stored in
`tok_typ tval;`

`void skip(int tok);`

The given token type is expected in the input text. It is skipped if found there. Otherwise, a syntax error message is generated. For single characters like ‘:’ or ‘,’, the token type equals the character value. Do not use skip() to skip strings or identifiers, since dynamic copies created by the scanner are possibly not freed.

`char *read_id(char *keyw);`

An identifier is read from the input text. In case keyw != NULL, it has to be introduced by the keyword and a ‘=’. The identifier may contain indices, they are included into the string returned by the routine. It is checked, that the identifier is not yet used for a network object, a variable, or a macro. The string returned is a dynamic copy which can be used directly for storing into the object member `name`.

`char *read_suc(char *keyw);`

Like read_id(), but the identifier can be used already. Additionally, the identifier may contain an extension with ‘->’.

`int read_int(char *keyw);`

An (arbitrarily complex) integer expression is expected. Its value is returned. Keyw: see read_id().

`double read_double(char *keyw);`

An (arbitrarily complex) double expression is expected. Its value is returned. Keyw: see read_id().

`char *read_string(char *keyw);`

A string expression is expected. The return value points to a dynamic copy which may be stored for internal use and should be deleted when not longer needed. Keyw: see read_id().

`char *read_word(char *keyw);`

A raw word (identifier ...) is expected. The return value points to a dynamic copy of the word read. It should be deleted when not needed any longer. Keyw: see read_id()

`int test_word(char *txt);`

It is tested whether the given word is next to read. The word is expected raw, not as string. If the word is found, TRUE is returned, otherwise FALSE. The input text *reading pointer is not shifted*. The routine fails in case of input language key words like if, else, macro ...

`void skip_word(char *txt);`

Like test_word(), but the word is skipped when found (reading pointer is shifted). A syntax error message is generated if the word is not found.

`int scan(void);`

Scan() reads the next input token from the input text. Like skip(), it should be applied carefully (see there).

10.0 Symbol Manager

Since registering a new object is done automatically by the parser, only the search routine is needed.

`root *find_obj(char *nam);`

The object nam is searched. Nam may contain name extensions (appended with ‘->’), e.g. input name specifications. These extensions are ignored by the routine. The return value points to the wished object, it is NULL in case the object is unknown.

11.0 Queues

In the header file “queue.h”, data item queues with limited and unlimited capacity are defined and implemented (all methods are defined inline). The queues are virtually as fast as programmed “by hand” for FIFO access. Although the classes also provide methods for time-sorted queueing and random access, these functions probably are not as optimal with respect to speed. The class name for the limited queue (which also may be switched into an unlimited state) is `queue`, the unlimited version is called `uqueue`. In the following, the class methods are described.

11.1 Initialisation

<u>Class uqueue</u>

`uqueue::uqueue(void);`

Constructor. Sets current length to zero.

<u>Class queue</u>

`queue::queue(int mx = 0);`

Constructor. Sets current length to zero, limit to mx. Default mx: 0. Mx < 0 is corrected to 0 (use unlimit() afterwards).

`int queue::setmax(int mx);`

Changes the queue limit to mx.
Returns TRUE: o.k.
FALSE: current queue length is larger than mx, limit not changed.

`void queue::unlimit(void);`

Deletes the capacity limitation of a queue. If possible, the class uqueue should be used instead.

11.2 Enqueueing

<u>Class uqueue</u>

`void uqueue::enqueue(data *pd);`

Enqueues the given item at the tail of the queue.

`void uqueue::enqHead(data *pd);`

Enqueues the given item at the head of the queue.

`void uqueue::enqTime(data *pd);`

Enques the item in front of the first item with a time member not smaller than pd->time (creates a sorted queue with increasing time stamps). If pd->time is larger than all other times, pd is added at the tail.

`int uqueue::enqPrec(data *pd, data *ref);`

Looks for ref and enqueues pd in front of ref.
Returns TRUE: o.k.
FALSE: ref not found in the queue

`int uqueue::enqSuc(data *pd, data *ref);`

Looks for ref and enqueues *pd behind *ref.
Returns TRUE: o.k.
FALSE: ref not found in the queue

<u>Class queue</u>

All uqueue methods are also available for queue. They first check on overflow and then call uqueue’s method. All methods return int:
FALSE on overflow or unsuccesssfull uqueue method,
TRUE otherwise.

11.3 Dequeueing

Classes uqueue and queue

`data *(u)queue::dequeue(void);`

Dequeues and returns the item at the front. An empty queue returns NULL.

`data *(u)queue::deqTail(void);`

Dequeues and returns the item at the tail. An empty queue returns NULL.

`data *(u)queue::deqTime(tim_typ tim);`

Dequeues and returns the first item with a time member not smaller than tim. Returns NULL if no appropriate item found.

`data *(u)queue::deqThis(data *pd);`

Dequeues the specified item. Returns pd if found in the queue, otherwise NULL.

11.4 Information

Classes uqueue and queue

`int (u)queue::isEmpty(void);`

Returns TRUE if queue is empty, FALSE otherwise.

`int (u)queue::isQueued(data *pd);`

Returns TRUE if *pd is queued, FALSE otherwise.

`int (u)queue::getlen(void);`

Returns the current queue length.

`data *(u)queue::first(void);`

Returns the item which would be returned by (u)queue::deq(), but does not actually dequeue it. Returns NULL in case the queue was empty.

`data *(u)queue::last(void);`

Returns the last item of the queue. It is not dequeued. Returns NULL if the queue was empty.

`data *(u)queue::sucOf(data *pd);`

Returns the item behind the specified one. Returns NULL if pd not found or pd is the last item of the queue.

`data *(u)queue::precOf(data *pd);`

Returns the item in front of the specified one. Returns NULL if pd not found or pd is the first item of the queue.

Class queue

`int queue::isFull(void);`

Returns TRUE if queue is full, FALSE otherwise.

`int queue::getmax(void);`

Returns the current queue limit. In case unlimit() was called in advance, -1 is returned.

11.5 Walk through a Queue

`int (u)queue::resCursor();`

Initializes the internal cursor for subsequent calls of getNext(). Returns TRUE on success, FALSE otherwise (queue empty).

`data *(u)queue::getNext();`

Each call returns the pointer to the next data object queued. Returns NULL, if end of queue reached. The first call returns the front of the queue.
Prior to the first call of getNext(), resCursor() has to be called. If resCursor() wasn't successful, getNext() must not be called. Between resCursor() and all subsequent calls of getNext(), no enqueueing or dequeueing actions are allowed.

Example

```
if (q.resCursor())
{    data*p;
     while ((p = q.getNext()) != NULL)
     {   // do sth with p
     }
}
else // queue empty.
```

12.0 Random Numbers

The random nuber generator has been integrated into the system to be independent of different implementations on different platforms. The implemented generator is, however, the simplest one which is known to be not very good. You can choose the random() routine from Berkley by editing the header file "defs.h". Uncomment the line defining the preprocessor variable USE_MY_RAND. Anyway, it is recommended to use always the name my_rand() when asking for a random number. The routine actually used then can be determined by an inline function in "defs.h".

For the generation of random numbers distributed non-uniformly, a table-based version has been chosen. Besides the network object class Distribution (see User's manual) which can generate transformation tables for arbitrary distributions, a modul providing transformation tables for geometrical distributions has been implemented. A user needing r.n.s registers with

```
int get_geo1_handler(double expectation);
```

The returned key value is used lateron to obtain a r.n.:

```
int geo1_rand(key);
```

It is also possible to ask for the transformation table (length: RAND_MODULO) itself:

```
tim_typ *get_geo1_table(int key);
```

Key again is the value returned by get_geo1_handler(). For applications see e.g. the object class GMDPquelle.

13.0 Error Messages

There are two families of error message routines. Both pass different numbers of string and integer arguments to an embedded fprintf(stderr, ...) call. To obtain an overview which error messages are available, take a look at src/kernel/all.c.

Syntax error messages (with relation to the input text):

```
void syntax_XX(...);
```

According to the current position of the input text reading pointer, the current input text line, the line number, the file name and a sign pointing to the current input token are written to stderr. The embedded error message is printed, YATS is left with exit status 1.

Plain error messages (without input text relation)

```
void errm_XX(...);
```

Like syntax_XX(), but no information related to the input text is given.