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

Matthias Baumann
Dresden University of Technology
Communications Laboratory

1.0 Introduction

YATS is a small discrete-time simulation tool tailored for investigations of ATM networks. The tool is “grown up” during 1995 / 1996 at the chair for telecommunications, Dresden University of Technology. The development was partly supported by the ACTS project AC 049 EXPERT.
An event scheduler, a symbol manager, and a scanner / parser front end constitute the kernel of the system. Basic network elements like different source types, (de)multiplexers, delay lines, measurement devices and graphical online-displays are provided. The system uses a simple script language for the problem description, and is written itself in C++.
The program expects the name of an input file:

```
   yats inputFile
```

It contains a description of the model configuration and commands to the simulation kernel (like: simulate 1 million time slots) and to the model objects (like: return losses in a multiplexer). Model description, simulation control and result analysis are eased by variables, loops and macros. Objects communicate with other ones and with the simulation kernel by unified methods.
On the web, YATS is available via

```
  http://www.ifn.et.tu-dresden.de/TK/yats/yats.html
```

Many thanks go to the following people who have implemented parts of the software, were involved in debugging or made proposals for improvements:

- Alfonso Santos, TID (Spain)
- Andreas Teresiak, TUD
- Axel Buksnowitz, TUD
- Gunnar Löwe, TUD
- Sven Forner, TUD
- Torsten Müller, TUD
- Wouter Ooghe, SUG (Belgium)

A trace of the version development can be found in the ASCII file yats/HISTORY of the distribution. Here, all differences between versions are listed.

2.0 General Remarks, Example Input File

The input file contains a series of statements which can represent network object declarations and commands. Statements beginning with a network object class identifier are interpreted as an object declaration. The parser creates an appropriate object and calls its initialisation method which then reads the parameters. For classes with outputs, the names of the succeeding objects are given in the statement. This defines the network structure.
Simulation control and result analysis are realized by commands to the defined objects (Sim is the predefined simulation kernel object). All statements beginning with an object identifier are recognized as a command. They are converted into a message to the object which again can read parameters from the input text.
The result analysis strategy is as follows: All objects “understand” simple commands to return the content of counters and other values collected during a simulation run. These values are passed into the kernel. Therefore, commands can be used in mathe-

matical expressions. Since the input language comprises macros, it is possible to define libraries for statistical evaluations. Output to the standard output (normally redirected to a file) is implemented by a print statement which gives full flexibility for output formatting.
Additionally, measurement devices with graphical online displays can be defined. These objects ask network objects for exporting addresses of variables to be displayed, and process and display the values on their own. This ensures that the complexity of network objects is independent of the possibilities for measurement and online display.

```
// example input file

var i, nsrc, load, sent;   // declaration of variables
nsrc = 10; load = 0.95;
// 10 Bernoulli sources:
for i = 1 to nsrc
    GEOquelle src[i]: ED=nsrc/double(load), VCI=i, OUT=mux->I[i];
// a multiplexer with buffer size 20
Multiplexer mux: NINP=nsrc, BUFF=20, OUT=sink;
// the sink:
Senke sink;

// simulate 100000 time slots
Sim->Run SLOTS=100000;

// print results
sent = 0;
for i = 1 to nsrc
    sent = sent + src[i]->Count;
print “cells sent: “, sent, “\n”;
print “CLR in mux: “,mux->Losses(1,nsrc)/double(sent),
      “\n”;

// end of example
```

More complex examples using also macros can be found in Section 10.0 on page 44.

3.0 Syntax of the Input Language

3.1 Statements

Statements are:

- `classID declaration ‘;’`

  Definition of a network object. The syntax of the definition is determined by the network object class implementation.

- `objectID ‘->’ command ‘;’`

  Command to an network object. The syntax of the command is determined by the network object class implementation.

- `‘var’ listOfIDsWithOptionalInit ‘;’`

Definition of variables. Variables can be defined everywhere. They are viewable until the block end. The entries in the list are seperated by commas. An entry can simply comprise an identifier (see Section 3.2 on page 4), or it may have the form '`ID '=' expression`' (initialisation).

- `'global' listOfIDsWithOptionalInit ';'`

  Declaration of global variables. These variables are not local to the current block, but are viewable from everywhere. The syntax otherwise is the same as for normal variables. This can be used to create e.g. global arrays as routing tables from inside of a macro. Due to "name space polution", globals should be used with care. For restrictions, see Section 11.4 on page 51.

- `variableID '=' expression ';'`

  Assignment to a variable.

- `'print' expression ',' ... ',' expression ';'`

  Print the expressions in the list to the standard output. Formatting depends on the expression value:

  integer: value, sign only if negative

  double: exponential representation, 1.10 digits

  string: contents

- `'print' '[' intExpr ',' ... ',' intExpr ']' exprList ';'`

  Some shells, e.g. sh and bash, allow to redirect arbitrary file descriptors (e.g. '`aProgram 3>aFile`'). `print[fd]` generates output to file descriptor `fd`. If `fd` is not writable, e.g. since it has not been redirected by a shell, an error message is launched. The output of `print[fd] ...` is unbuffered (direct usage of write(2)), whereas the normal `print` uses the stdout stream with fflush(stdout) at the end of each `print` statement. `print[fd1,fd2, ...]` generates the output for all file descriptors in the list.

- `'{' statement ... statement '}'`

  Block, the ';' is part of the statement (like in C). When the block is left, then all variables and macros defined inside are deleted. This does not hold for globals, see Section 11.4 on page 51.

- `'macro' ID '(' listOfFormalParameters ')' block`

  Definition of a macro. The formal parameters (if any) are local to the block (block: '{' statements '}'). The parameter names are separated by commas. For examples see file "MACROS" and Section 4.1 on page 8. Note: macro names cannot be overriden when entering a new block (variables can, however). Nevertheless, a macro is local to a block and is deleted when the block is left.

- `macroID '=' expression ';'`

  Assignment of the macro return value (rather like in PASCAL). Inside of the macro body, this assignment can be done arbitrarily often. The macro returns the value from the last assignment. If no such assignment is processed, then the macro cannot be used in expressions (error message otherwise).

- `macroID '(' listOfExpressions ')' ';'`

Macro call. The expressions in the list are the actual parameters, they are seperated by commas. The number of actual parameters is checked against the number of formal parameters.

- `‘system’ stringExpression ‘;’`

  Starts a shell with the command specified by the string.
- `‘exit’ intExpression ‘;’`

  Leave YATS with the given exit status.
- `‘;’`

  No-op.

Control structures are:

- `‘if’ ‘(‘ expression ‘)’ statement ‘else’ statement`

  The else branch is optional.

-
  ```
  ‘for’ variableID ‘=’ intExpression ‘to’ intExpression
      statement
  ```

  PASCAL-like loop: the variable is counted from the first to the second int-value.
- `‘while’ ‘(‘ expression ‘)’ statement`

  The statement is executed until the expression evaluates to zero.
- `‘foreach’ variableID ‘(‘ listOfExpressions ‘)’ statement`

  The statement is executed for each value (values separated by commas) given in the brackets.

-
  ```
  ‘switch’ ‘(‘ intExpression ‘)’ ‘{‘
      ‘case’ listOfIntExpressions ‘:’ statement
      ‘case’ listOfIntExpressions ‘:’ statement
      ‘default’ ‘:’ statement ‘}’
  ```

  For each case branch, a list of values can be specified (seperated by ‘,’). The default branch is optional.

3.2 Identifiers

All identifiers for variables, macros, and for network objects can be formed using indices and aggregation (with ‘.’). An example for an identifier with both integer indices and aggregation is ‘`a[1].b[2]`’. String expressions also can serve as indices, e.g.: `var a[“X”];`

Including references and literals (see Section 3.5 on page 7), the syntax for identifiers can be summarised as follows:

```
id:      stdId      // standard form, also with ‘lit’
|        derefId    // somewhat restricted version with ‘deref’ construct

//  stdId comprises a kind of aggregate identifiers.
stdId:   baseId
|        stdId ‘[‘ intExpr ‘]’      // normal integer index
|        stdId ‘[‘ stringExpr ‘]’  // strings can form indices
```

```
|          stdId ‘.’ baseId           // aggregated ID
baseId:    simpleIdentifier           // a normal word like ‘a1’, beginning with a letter
|          ‘lit’ ‘(‘ stringExpr ‘)’   // stringExpr may contain all except white space

//   A ‘deref’ construct only may stand at the beginning, and it only may be expanded
//   with indices (number of exepected indices follows from the ref statement).
derefId:   ’deref’ ‘(‘ refStringExpr ‘)’
                                      // refStringExpr has been generated by ‘ref’
|          derefId ‘[‘ intExpr ‘]’    // normal index
|          derefId ‘[‘ stringExpr ‘]’// strings can serve as indices
```

3.3 Variables, Expressions, and Built-In Functions

The language is block oriented (block: `‘{‘ statements ‘}’`). This means that variables (except globals) and macros declared inside of a block are deleted when the block is left. On the other hand, variables defined outside of the block and having the same name appear again with their old value when the block is left. Objects, however, are viewable for ever and will never be released.

Global variables are defined using the keyword ‘global’. They are not local to the current block, but are viewable from everywhere. Globals are never deleted. A number of restricitons is bound to global variables, see Section 11.4 on page 51.

Variable types are interger, double and string. Variables are not declared with a certain type, but bear the type resulting from the last assignment. After declaration, they have a special type which ensures that they cannot be used in expressions.

Constants have the following types:

“xyz” -> string,

123 -> integer,

1.23 -> double,

1e-3 -> double.

Type casts
With `‘string’ ‘(‘ expr ‘)’`, `‘int’ ‘(‘ expr ‘)’`, and `‘double’ ‘(‘ expr ‘)’`, types can be translated. The conversion double -> int performs rounding instead of truncation. In case of casting string to int or double, no white spaces are allowed in the string.

Arrays of variables can be defined as follows:

```
var i; // auxiliary variable
for i = 1 to 10
    var x[i]; // defines x[1] to x[10]
```

It is *essential* that the `var` statement is not encapsulated by a block (‘`{`‘ ... ‘`}`’), since variables are only local to a block. Arrays with more dimensions are possible, strings can serve as indices. For an example, see Section 10.1 on page 44.
Note: Actually, an element of an array is a variable on its own, the name comprising the indices, too. Very large arrays therefore should be avoided (speed, memory space).

Operators (priorities like in C) are:

- `‘+’  ‘-’  ‘*’  ‘/’  ‘%’`

  The op ‘%’ is only applicable on integer operands. The op ‘+’ also concatenates strings.
- `‘==’  ‘!=’  ‘<‘  ‘>’  ‘<=’  ‘>=’`

  Comparisons yield the values 0 and 1 (integer). The ops == and != are applicable on strings.
- `‘&&’  ‘||’`

  Logical AND and OR, yield 0 and 1 (integer).
- `‘(‘ expression ‘)’`
- `‘!’  ‘+’  ‘-’`

  Negation (yields 0 and 1), unary + and -.
- `‘int’ ‘(‘ expression ‘)’`

  `‘double’ ‘(‘ expression ‘)’`

  `‘string’ ‘(‘ expression ‘)’`

  Explicite type casts, see above.

Operands in expressions can be:

- `constant`

  The type results from the way of writing, see above.
- `variableID`

  Only initialised variables are allowed (otherwise syntax error message).
- `objectID ‘->’ command`

  Commands to network objects can return a value. The command method of the object decides, whether the command returns a value.
- `macroID ‘(‘ listOfExpressions ‘)’`

  Macros can return a value. A test is performed, whether a return value has been specified.
- `‘env’ ‘(‘ stringExpression ‘)’`

  Reads an environment variable and returns its value as string. In case the variable does not exist, the empty string (““) is returned.

Built-in mathematical functions:

- `‘pow’ ‘(‘ expression1 ‘,’ expression2 ‘)’`

  Returns expression1 to the power of expression2. Integer and double expressions are allowed, the return type is always double.
- `‘exp’ ‘(‘ expression ‘)’`

  Returns *e* to the power of expression. Integer and double expression allowed, the return type is always double.
- `‘log’ ‘(‘ expression ‘)’`

Returns the natural logarithm. Like exp().

- `‘sqrt’ ‘(‘ expression ‘)’`

  Returns the square root. Like exp().

- `‘rand’ ‘(‘ ‘)’`

  Returns an integer random number (equally distributed in 0 ... 32767).

3.4 Include Files, System Interface

Include file:

```
#include “fileName”
```

The rest of the input line is skipped (also the start of a comment!). The directive has to be written immediatly at the beginning of a line. Include files can be nested.

System interface:

- `‘system’ stringExpression ‘;’`

  Starts a shell with the given command.

- `‘exit’ intExpression ‘;’`

  Leave YATS with the given exit status.

- `‘env’ ‘(‘ stringExpression ‘)’`

  Reads an environment variable and returns its value as string. In case the variable does not exist, the empty string (““) is returned. Env() can be applied in normal expressions. The most common application is to read and convert an environment variable specifying a parameter value, e.g (see also Section 10.1 on page 44):

  ```
  buf = int(env(“BUF”)); // generates a syntax error if
                         // BUF not set
  Multiplexer mx: NINP=10, BUFF=buf, OUT=sink;
  ```

3.5 References, Literals

References
The constructs below provide replacements for real pointers to scalars and arrays. They allow to create and use references to variables, macros, and network objects as well as to arrays of these entity classes.

- `‘ref’ ‘(‘ entityID ‘)’`

  Returns a reference to the given single entity (which nevertheless may be part of an array). The generated reference actually is a string encrypting the ID, and additional block information. It should only be assigned to variables, but never be manipulated.

- `‘ref’ ‘(‘ baseID ‘:’ validExampleIndices ‘)’`

  Returns a reference to an array of entities. `BaseID` is the identifier without the indices which shall be added when resolving the reference. `ValidExampleIndices` are used to construct a complete valid identifier (necessary to bind the reference to the “home” block of the entity - provides savety, see Section 11.1 on page 48). The generated reference actually is a string encrypting the ID, and additional block and index information. It should only be assigned to variables, but never be manipulated.

- `‘deref’ ‘(‘ scalarRef ‘)’`
  `‘deref’ ‘(‘ arrayRef ‘)’ indices`

  `Deref()` resolves the given reference. The number of indices supplied is checked against the number declared in the corresponding `ref()` expression.
  If the entity is a macro, parameters are appended as usual, e.g. `deref(mRef)(x)`.
  For network objects, extensions also follow the normal rules, e.g.
  `deref(objRef)[index]->command.`

Example (see also Section 5.0 on page 12 and Section 10.1 on page 44)

```
var x, y[1], y[2], r1, r2;
x = 1; y[2] = 2;
r1 = ref(x);      // reference to x
r2 = ref(y:[1]);  // reference to y[1] and y[2]
print deref(r1) + 2, “\n”;    // yields output 3
print deref(r2)[2] + 2, “\n”; // yields output 4
deref(r1) = 0;     // assignment to x
```

For an exact syntax definition of identifiers comprising ‘ref’ constructs see Section 3.2 on page 4, additional remarks are found in Section 11.1 on page 48.

Literals
Generally, references should be used wherever possible, since they provide some degree of savety (see Section 11.1 on page 48). References always can be used, if an entity already defined is referred to. When writing e.g. macros which shall create compound network objects like an entire switch, then problems may arise. The following identifier substitution will be usefull:

- `‘lit’ ‘(‘ stringExpr ‘)’`

The effect is the same as it would be for `deref(stringExpr)`, except that `stringExpr` can be created arbitrarily. Additionally, the restrictions of ‘deref’ (only index expansion) do not apply. Thus, ‘lit’ can be used to flexibly generate new identifiers. For an exact syntax definition of identifiers comprising ‘lit’ constructs see Section 3.2 on page 4. For applications, see Section 4.1 on page 8. Please pay also attention to the further notes in Section 11.2 on page 49.

4.0 Modular Description of Complex Network Structures

4.1 Commented Example

Some kind of hierarchical system description can be achieved using macros, literals, and string manipulation. The gain in clarity of course depends on the complexity of the sub-models. Macros can be used to define sub-models since network objects are not local to the current block. They continue to exist and are never deleted. The following example macro defines a simple end system consisting of a source and a delay line.

`Name`, and `nxt`, are string arguments which are converted into identifiers using `lit()` (see Section 3.5 on page 7).

```
macro endsys(name, load, delay, nxt)
{    CBRquelle lit(name).S: DELTA=int(1.0/load), VCI=1,
             OUT=lit(name).L;
     Leitung lit(name).L: DELAY=delay, OUT=lit(nxt);
}
```

To connect the end systems to a multiplexer, some string manipulation can be used (`string(i)` converts `i` into a string which can be concatenated with the rest):

```
macro idx(bas, i)  // this macro is defined in “MACROS”
{    idx = bas + ”[“ + string(i) + ”]”;
}

var  i;
for i = 1 to 10
     endsys(idx(“src”, i), 0.05, i, idx(“mux->I“, i));
Multiplexer mux: NINP=10, BUFF=100, OUT=snk;
Senke snk;
Sim->Run SLOTS=500;
print src[1].S->Count, “\n”;
```

This first version has the disadvantage of less readable code due to the idx() macro calls. The class `MacroShell` provides a better interface for the macro. It allows to define whole sub-models (implemented by macros) like normal network objects. For larger sub-models it might be worthwhile to write such an interface which provides typed arguments and also allows for optional arguments with default values (see Section 4.2 on page 10). It is also possible to ‘translate’ commands directed to a macro shell object into macro calls, see Section 11.3.1 on page 49.

```
// The definition of macro endsys() remains the same.
// Interface:
MacroShell EndSys:
     // Arguments 1 & 4 are identifiers (keyword lit),
     // argument 2 is double, argument 3 is integer.
     // The given keywords are expected (none for arg 1).
     ARGS = (lit@1: LOAD=double@2, DELAY=int@3,
             OUT=lit@4),
     // arguments 1 & 4 are casted to strings
     MACRO = endsys(string(@1), @2, @3, string(@4));

var  i;
for i = 1 to 10
     EndSys src[i]: LOAD=0.05, DELAY=i, OUT=mux->I[i];
Multiplexer mux: NINP=10, BUFF=100, OUT=snk;
Senke snk;
Sim->Run SLOTS=500;
print src[1].S->Count, “\n”;
```

4.2 Macro Shells

Macros are useful to define sub-models in a modular model description, but the code applying these macros becomes less readable, see Section 4.1 on page 8. The object class `MacroShell` provides a better interface to the macro implementation, allowing a ‘read-and-feel’ like for direct network object declarations. Keywords, typed and optional parameters with default values are supported. New identifiers can be written down as identifiers (without ‘”’ signs), they can be converted to strings by the interface object.

```
‘MacroShell’ shellID ’:’
                ‘ARGS’ ‘=’ ‘(‘ argList ’)’ ‘,’
                ‘MACRO’ ‘=’ macID ‘(‘ parList’)’
                { ‘,’ commandSpecs }
                { ‘,’ ‘print’ { ‘=’ intExpr } } ‘;’
```

The argument list `argList` specifies keywords, argument types, and default values for optional arguments. It is built as list of entries, the entries being separated by the same delimiters which are expected lateron during instantiation. As delimiters, currently only ‘:’ and ‘,’ are available. The end of the list is marked by the closing bracket ‘)’. A list entry has the following syntax:

```
{keyWord‘=’} typeName ‘@’posNo {‘(‘ ‘default’ defVal ‘)’}
```

The entry may begin with an optional keyword (given as raw word), followed by a ‘=’ sign. Next, a type name `typeName` has to follow. It specifies the type of the argument. Possible are ‘int’, ‘double’, ‘string’, and ‘lit’. The type ‘lit’ says that a raw identifier is expected. The ‘@’ required then is complemented by the integer position number which has to increase one by one, starting with 1. Arguments with keyword (only these) can be marked as optional by specifying a default value. This is done in brackets ‘()’ with leading ‘default’ keyword. The expression or identifier `defVal` has to match the type given for the parameter. For an exception (NULL identifier) see Section 11.3.2 on page 51. If during later instantiation an optional parameter is missing (expected keyword not found), then the default value is passed to the macro. The delimiter following the corresponding `argList` entry then is not expected, thus directly continuing with the next argument.
The parameter list `parList` specifies the macro parameters, delimited by commas ‘,’. A macro parameter declaration can comprise a cast operator specifying that the value shall be casted before it is passed to the macro. Arguments declared as ‘string’ can be transformed to ‘lit’ (the ‘”’ signes are stripped off), ‘lit’ can be changed to ‘string’ (simply add the ‘”’ signes to the identifier read). Other casts currently are not foreseen. To summarise, a macro parameter in the list `parList` is defined as follows (`posNo` again has to count upwards, starting with 1).

```
    { { ‘lit’ | ‘string’ } ‘(’ } ‘@’posNo { ‘)’ }
```

The optional part marked with `commandSpecs` allows to generate an object and ‘translate’ commands to this object into macro calls. This is described in Section 11.3.1 on page 49.
If the final ‘print’ is given (optional), then the generated macro call is printed to stdout, before the macro actually is executed. Inserted default values are highlighted by com-

ments. In case the ‘print’ is followed by an integer expression with ‘=’, then logging is performed only if the expression is non-zero.

Example
```
// this macro actually should define our sub-model:
macro sub(nam,a,b)
{    // only echo parameters
     print “sub: nam=”, nam, “ a=”, a, “ b=”, b, “\n”;
}
// the macro shell
MacroShell Sub: ARGS=(
     lit@1:          // first arg: identifier
                     // no keyword, delimiter ‘:’
     D=int@2 (default -1) ,
                     // optional integer arg, keyword ‘D’
                     // default value -1
     LD=double@3),    // mantadory double arg, keyword ‘LD’
     MACRO=sub(      // call the macro sub
     string(@1),      // convert identifier to string
     @2, @3),        // pass arguments 2 and 3 directly
     print;          // log the generated macro call
// apply the shell
var i;
for i = 1 to 2
     Sub s[i]: D=i, LD=i;
for i = 3 to 4
     Sub s[i]: LD=i;// default value -1 for D
```

Another example can be found in Section 4.1 on page 8.

4.3 Input Name Aliasing and Dummy Objects

Sometimes non-matching input names of different network objects lead to description problems. Then it is possible first to include dummy nodes (see Section 7.12.3 on page 40). Secondly all network object classes derived from the generic class `ino` provide the following command:

```
objName->AliasInput(
     aliasInputNameAsString -> origInputNameAsString);
mux->AliasInput(“I[12]” -> “I[10]”);
     // I[10] is reachable via the name I[12], too
```

For both strings, it is possible to write the word `NoExt`. This specifies the input without extension (input name equals object name).

```
mux->AliasInput(NoExt -> “I[10]”);
     // I[10] is reachable via the pure multiplexer name
snk->AliasInput(“Data” -> NoExt);
     // snk is reachable via snk->Data, too
```

The command AliasInput() does not perform any checks, whether the given original input name exists. Thus, errors are detected lateron during connection setup. Recursive aliasing is not supported.

5.0 Statistical Evaluation, Simulation Control

5.1 Calculation of Confidence Intervals

Confidence intervals can be calculated using the object class ConfidObj. Such an object is defined like a normal network object, but it does not have network object functionality. Commands to add measured values to a dynamic array managed by the object, and to ask for statistics and confidence interval bounds are provided. Older, macro-based versions to calculate confidence intervals can be found in Section 11.5 on page 51.

Declaration

```
ConfidObj conf: LEVEL=0.99;
```

LEVEL: optional: level of confidence. Default: 0.95. For other values see below.

If LEVEL is not given, then also the ‘:’ has to be left out. For LEVEL, the values 0.9, 0.95, 0.975, and 0.99 are supported. The object has to be defined before the first simulation run (`Sim->Run` ...) is performed.

Commands

`conf->Add(double)`
: Adds a value to the internal array, no return value.

`conf->Len`
: Returns number of values collected so far (int).

`conf->Val(int)`
: Returns a specific value (1 ... Len), double.

`conf->Flush`
: Flushs all values (Len := 0), no return value.

`conf->Mean`
: Returns the mean of the values collected so far (double).

`conf->Var`
: Returns the *empirical* variance of the values collected so far, double.

`conf->Lo`
: Returns the lower bound of the current confidence interval, based on the values collected so far (double).

`conf->Up`
: Returns the upper bound of the current confidence interval, based on the values collected so far (double).

`conf->Width`
: Returns the width (upper bound - mean) of the current confidence interval, based on the values collected so far (double).

`conf->Lo(double)`
`conf->Up(double)`
`conf->Width(double)`
: These versions return the values, if the level of confidence equals the value of the additional parameters.

5.2 Batch Means Procedure

This macro is part of the macro file “MACROS” in src/examples.

BatchMeans(level, prec, maxcorr, nbat, batsiz, refObs, refLog)
Implements the algorithm of the so-called batch means procedure. For given level of confidence and relative target width of the confidence interval (according to Student’s t-distribution), observations are performed until the conditions are met. To obtain observation results and report on procedure progress, two user-provided macros are applied (references `refObs` and `refLog`). Once `BatchMeans()` has been called, it entirely controls the simulation progress, repeatedly calling the two specified macros. It returns when the simulation goal has been reached. Macro parameters are:

`level`: level of confidence (values: see macro `Confid()`)

`prec`: relative target width of confidence interval (`(upBound-mean)/mean`).

`maxcorr`: upper bound of correlation measures. Additionally to the confidence interval, correlations between collected batch values are considered. Measurements are continued as long as the coefficient of correlation between batch values exceeds this bound.

`nbat`: number of batches to be used

`batsiz`: initial number of observations per batch. This number is doubled with each iteration step.

`refObs`: reference to a macro which performs an observation. The macro has to prepare (reset counters etc.) and perform a measurement. The value of interest is expected as macro return value. No arguments are passed to this macro.

`refLog`: reference to a macro which is called to report on procedure progress. It is invoked (upon completion of an iteration) as follows:
`deref(refLog)(cnt, mean, width, c1, c2, flag)`

> `cnt`: number of observations already made
>
> `mean`: current estimate of mean value
>
> `width`: current absolute width of confidence interval (`upBound-mean`)
>
> `c1`, `c2`: current correlation measures
>
> `flag`: set to zero, if target conditions are met. One otherwise.

6.0 Commands to the Simulation Kernel

- `Sim->Run SLOTS=slots {, DOTS=dots};`

  Simulates the network for SLOTS time slots. If DOTS is given and larger than zero, then a dot is printed to standard output approximately after every DOTS-th time slot. Approximately means, that the dots are only printed together with each time slot with (SimulationTime modulo TIME_LEN == 0). TIME_LEN is a constant defined in the source file “defs.h”, it defines the static length of the calendar queue used by the scheduler. By default TIME_LEN equals 1000, it might be useful to change it to a prime number.

- `Sim->ResetTime;`

Reset simulation time. The activation time of all events registered at the kernel is diminished by the current time. This command is used to avoid an overflow of the simulation clock in case of very long runs (for simple models, this can happen after two hours!).

- `Sim->SetRand(seed);`

  The central r.n. generator is set to the seed value. This is only usefull before defining any objects, since some objects call the r.n. generator e.g. for the initial event registration. By default, the generator is initialized with the system time.

- `Sim->EchoInput;`

  The input text (include files excluded) is printed to standard output, every line is marked with a hash mark (‘#’).

7.0 Available Network Object Classes

7.1 Sources

7.1.1 CBR Source

Declaration

```
CBRquelle src: DELTA=5, VCI=1, OUT=line;
```

DELTA: cell distance (integer)
VCI: VC number of generated cells
OUT: input name of the succeeding object

Exported variables

```
src->Count
```

Returns the number of sent cells.

Commands

```
src->ResCount
```

Resets the cell counter.

```
src->Restart
```

Sets a new cell phase (random).

Output data type

Cell

7.1.2 Bernoulli Source

Declaration

```
GEOquelle src: ED=4.5, VCI=1, OUT=line;
```

ED: mean of the geometrically distributed cell distance (double)
VCI: VC number of generated cells
OUT: input name of the succeeding object

Exported variables

```
src->Count
```

number of sent cells.

Commands

```
src->ResCount
```

Resets the cell counter.

Output data type
Cell

7.1.3 Source with Arbitrarily Distributed Cell Distances

Declaration
```
DistSrc src: DIST=dist, VCI=1, OUT=line;
```
DIST: name of the distribution object providing the distribution
VCI: VC number of generated cells
OUT: input name of the succeeding object

Function
The source uses the distribution table of a distribution object which has to be declared in advance (Distribution object: Section 7.2 on page 17).

Exported variables
```
src->Count
```
number of sent cells.

Commands
```
src->ResCount
```
Resets the cell counter.

Output data type
Cell

7.1.4 ON/OFF Source with Geometrically Distributed Phase Durations

Declaration
```
BSquelle src: EX=10, ES=100, DELTA=10, VCI=1, OUT=line;
```
EX: mean number of cells per burst (double)
ES: mean duration of the silence phase (in time slots, double)
DELTA: cell distance in the ON-state (in time slots, integer)
VCI: VC number of generated cells
OUT: input name of the succeeding object

Exported variables
```
src->Count
```
number of sent cells.

Commands
```
src->ResCount
```
Resets the cell counter.

Output data type
Cell

7.1.5 MMBP Source (ON/OFF)

Declaration
```
MMBPquelle src: EB=100, ES=100, ED=10, VCI=1, OUT=line;
```
EB: mean duration of the ON-state (in time slots, double)
ES: mean duration of the OFF-state (in time slots, double)
ED: mean of the geometrically distributed cell distance (double)
VCI: VC number of generated cells
OUT: input name of the succeeding object

Exported variables

`src->Count`

number of sent cells.

Commands

`src->ResCount`

Resets the cell counter.

Output data type

Cell

7.1.6 GMDP Source

Declaration

```
GMDPquelle src: NSTAT=2, DELTA=(2,3), EX=(10,20),
    TRANS=(0,1,1,0), VCI=1, OUT=line;
```

or:

```
GMDPquelle src: NSTAT=2, DELTA=(2,3), DIST=(d1,d2),
    TRANS=(0,1,1,0), VCI=1, OUT=line;
```

NSTAT: number of states (integer)
DELTA: cell distances (integer, for zero cell rate see below)
EX: mean numbers of cells per state - geom. distributed (double, for zero cell rate see below)
DIST: objects providing distributions of cells per state (for zero cell rate see below)
TRANS: matrix of transition probabilities, row by row (double)
VCI: VC number of generated cells
OUT: input name of the succeeding object

Function

The object class provides a GMDP source with geometrically (if EX is given), or with arbitrarily (DIST is given) distributed sojourn times. In the latter case, the distributions are imported from Distribution objects (Section 7.2 on page 17) wich have to be defined in advance.
To define states with zero cell rate, define a DELTA value of zero. The EX entry then specifies the mean duration of this silence state, the DIST entry gives the distribution of the duration.

Exported variables

`src->Count`

number of sent cells.

Commands

`src->ResCount`

Resets the cell counter.

Output data type

Cell

7.1.7 GmdpStop: a Source Following the Start-Stop Protocol

The source recognizes start and stop messages (see Section 9.3 on page 44) sent by the succeeding object. Therefore, it possesses an input src->Start. Declaration and commands do not differ from GMDPquelle. Up to now, the only corresponding object class is ShapCtrl (see Section 7.7.3 on page 26). Both are used for the test of new classes.
Oct 5, 1996: Now, also the ABR source is a candidate for GmdpStop.

7.1.8 Source Reading IATs from a Trace File

Declaration

```
Filsrc src: FILE=”trace.bin”, REPEAT=10, START=20,
    WAIT=100000, VCI=1, OUT=line;
```

FILE: name of the file containing the inter arrival times (see below)
REPEAT: number of trace repetitions. Can be omitted, then the file is read once.
START: start with the START-th entry in the trace file
WAIT: wait WAIT time slots before sending first cell. Can be omitted.
VCI: VC number of generated cells
OUT: input name of the succeeding object

Function

The object reads the inter arrival times from the given trace file. The stored IAT values are in binary format (unsigned integer). In case of entries with IAT zero, the source fails. The parameters REPEAT, START, and WAIT are optional.

Exported variables

```
src->Count
```

number of sent cells.

Commands

```
src->ResCount
```

Resets the cell counter.

Output data type

Cell

7.1.9 Source Sending a Directly Given Cell Sequence

Declaration

```
ListSrc: src: N=3, DELTA=(10,10000,5), VCI=1, OUT=line;
```

N: number of cells, int
DELTA: inter departure times (first cell sent at the first DELTA), int
VCI: VCI number of cells generated, int
OUT: where to send cells

Exported variables

```
src->Count
```

number of sent cells.

Commands

```
src->ResCount
```

Resets the cell counter.

Output data type

Cell

7.2 Definition of Distributions

A distribution object generates a r.n. transformation table from a given distribution. This table can be imported by other objects - see e.g. DistSrc (Section 7.1.3 on page 15) and GMDPquelle (Section 7.1.6 on page 16). The distribution must not have a nonzero probability for the r.v. value zero. There is a couple of versions to specify the distribution:

```
Distribution dist: FILE=”xyz”;
```

In the given ASCII file, each line contains the r.v. value (integer, greater than zero, ascending order) and the associated probability (double). The values are delimited by white space. R.v. values with probability zero can be omitted. A '#' marks the rest of a line as comment.

```
Distribution dist: TABLE=(1, 0.5), (2, 0.5);
```

The probabilities are directly given in the input text.

```
Distribution dist: DISTRIB=(x = x_min to x_max,
             1.0 / (1 + x_max - x_min));
```

Calculation of the distribution from a formula. The variable x has to be defined in advance, the formula can be an arbitrary double expression. In case of very "long" distributions, the repeated interpretative evaluation can become slow.

```
Distribution dist: GEOMETRIC(E=4.5);
```

Gemetrical distribution with the given mean. The distribution is shifted by one, i.e. P(X=0)=0.

```
Distribution dist: BINOMIAL(N=20, P=0.5);
```

Binomial distribution with parameters n and p. The distribution is shifted by one, i.e. P(X=0)=0.

7.3 Multiplexer

7.3.1 Standard Multiplexer (Arrival First)

Declaration

```
Multiplexer mux: NINP=10, BUFF=50, MAXVCI=50, OUT=sink;
```

NINP: number of inputs (integer)
BUFF: buffer size (cells, integer)
MAXVCI: optional: max. VC number (default: NINP)
OUT: input name of the succeeding object

Function

Multiplexer with server strategy Arrival First, inputs are served in random order. Losses are counted per input and per VC. The per-VC registration is only performed for cell-like data items and in the VCI range 0 ... MAXVCI.
The input names are nameOfObject->I[inputNumber], where inputNumber ranges from 1 to NINP.

Remark - Bug

The implementation is fast but the sojourn time in the mutliplexer is exactly one time slot too long. The network object class `MuxAF` (Section 7.3.3 on page 19) implements the correct sojourn time.

Exported variables

`mux->Loss(i)` or
`mux->LossInp(i)`
loss at input i.
`mux->LossVCI(i)`
loss at VCI i.
`mux->LossTot`
total loss.
`mux->QLen`
current queue length (actually: system occupation).

Commands

`mux->Losses(i, k)` or
`mux->LossesInp(i, k)`
Returns sum of losses on inputs i to k.
`mux->LossesVCI(i, k)`
Returns sum of losses on VCs i to k.
`mux->ResLoss`
Resets all loss counters.
Input data type
Data

7.3.2 Multiplexer with Lower Output Rate (Departure First)

Declaration
```
MuxDF mux: NINP=10, BUFF=100, MAXVCI=100, ACTIVE=4,
    OUT=line;
```
NINP: number of inputs (int)
BUFF: number of buffer places (int)
MAXVCI: optional: max. VC number (default: NINP)
ACTIVE: optional: serve output queue only every ACTIVE-th time slot
OUT: where to send cells
Function
Multiplexer with Departure First strategy. Inputs are served in random order. The output queue is served only during each ACTIVE-th time slot in case ACTIVE is given.
Remark – Lax Model
The model is simple: a queue comprises both the real queue places and the server place. A deomon at multiplexer output scans the queue during each ACTIVE-th time slot, and sends a data item if available. Thus, some data may stay in the “server” for less then ACTIVE time slots. For more sophisticated models, see multiplexers in Section 7.4 on page 22.
Exported Variables and Commands
like for Multiplexer (see Section 7.3.1 on page 18).

7.3.3 Multiplexer with Lower Output Rate (Arrival First)

```
MuxAF mux: NINP=10, BUFF=100, MAXVCI=100, ACTIVE=4,
    OUT=line;
```
Declaration, function, and commands as for MuxDF. The server strategy is Arrival First.
Remark – Lax Model
The model is simple: a queue comprises both the real queue places and the server place. A deomon at multiplexer output scans the queue during each ACTIVE-th time slot, and sends a data item if available. Thus, some data may stay in the “server” for less then ACTIVE time slots. For more sophisticated models, see multiplexers in Section 7.4 on page 22.

7.3.4 Multiplexer with Arbitrarily Distributed Serving Time

Declaration
```
MuxDist mux: NINP=10, BUFF=50, MAXVCI=100,
   DIST=serv_dist, OUT=sink;
```

NINP: number of inputs (integer)
BUFF: buffer size (cells, integer)
MAXVCI: optional: largest VC number for loss count, default: NINP
DIST: name of the distribution object defining service time distribution
OUT: input name of the succeeding object

Function

Multiplexer with Departure First strategy. Inputs are served in random order. The time spent in the server at queue output is distributed according to the specified distribution (Distribution object: Section 7.2 on page 17).

Exported Variables and Commands

like for Multiplexer (see Section 7.3.1 on page 18).

7.3.5 Multiplexer with Early Packet Discard Scheme

Declaration

```
MuxEPD mux: NINP=<int>,  BUFF=<int>, MAXVCI=<int>,
    THRESH=<int>, OUT=<object input>;
```

NINP: number of inputs
BUFF: buffer size in cells
MAXVCI: optional: maximum VC number, default: NINP
THRESH: buffer occupation at which to begin to discard bursts
OUT: where to send cells

Function

The object implements an early packed discard strategy: for all connections sending data type AAL5Cell (see Section 9.1 on page 43) on VCI 0 to MAXVCI, burst durations are kept track off. With each beginning of a burst, the current buffer occupation is compared to THRESH. If this mark is reached, then all cells of the burst are dropped. Bursts which began successfully but nevertheless lost a cell (due to background traffic or too high threshold), are also dropped til the end. Connections with out-of-range VCI and non-AAL5Cell data items are multiplexed following the normal way (using the same common buffer).

Remark

The last cell of a dropped frame is *not* passed. If THRESH has been set appropriately, then this should not be a problem: frames are either forwarded or dropped completely. Otherwise it could be a problem. Therefore it is planed to add a flag which turns on forwarding of last cell (the default should remain as it is).

Exported Variables and Commands

like for Multiplexer (see Section 7.3.1 on page 18).

7.3.6 Multiplexer with Weighted Fair Queueing Strategy

Declaration

```
MuxWFQ mux: NINP=<int>, MAXVCI=<int>, OUT=<object input>;
```

NINP: number of inputs
MAXVCI: optional: largest VC number possible. default: NINP
OUT: where to send cells

Function

The weighted fair queueing algorithm according to J.W. Roberts is implemented. The connection specific parameters buffer size and inverse of the mean cell rate are set by the command `mux->SetPar(...)`, see below. An incomming cell causes an error message in case its VCI has not yet been initialized with `SetPar()`.

WFQ Algorithm

Per VC, an input queue is maintained. A sort queue at the output manages the order in which to serve cells from the VC queues. It holds the front cell from each VC queue (provided there is one). The decision about serving order is made as follows. A global variable “Spacing Time” – it is common for all VCs – always contains the “Virtual Time” of the last cell which has left the multiplexer. This “Virtual Time” has been assigned to the cell when it was entering the sort queue (the preceeding cell of this VC just had been served, or the VC recently had not been present in the sort queue). The “Virtual Time” is always set to the current “Spacing Time” plus the inverse mean cell rate of the new cell’s connection registered at the multiplexer. The sort queue serves cells in the order corresponding to their “Virtual Times”.

The effect of the algorithm is that a cell entering the sort queue can overtake other cells, if its inverse mean cell rate is low enough compared to the other connections already waiting in the queue.

In the current implementation, the sort queue does not really hold cells, but only refers to the corresponding VC queues. A cell is dequeued from a VC queue in the instance it has to be sent, and the “Virtual Time” is assigned to the whole associated VC queue. Thus, possibly one additional buffer place might be needed in each VC queue (compared to the original algorithm).

Exported Variables (additional to Multiplexer, see Section 7.3.1 on page 18)

mux->QLenVCI(vc)

current queue length of this VC

Commands (additional to Multiplexer, see Section 7.3.1 on page 18)

mux->SetPar(vci, invCellRate, BufSiz)

The command sets the connection-specific WFQ parameters for VC vci. The invers cells rate has to be integer (mean cell spacing in time slots), as well as the buffer size in cells.

Input data type

Cell

7.3.7 Multiplexer with Input Buffers

Declaration

```
MuxInpBuf mx:   NINP=10, BUF=100, BSTART=90,
      {BYTEFACT=0.3 | SERVICE=1}, RELAX=3,
      OUTCTRL=(i: src[i]->Start), OUT=snk;
```

NINP: int, number of inputs

BUF: int, buffer size per input, counting in data objects.

BSTART: optional, int. Turns on start-stop protocol at inputs. Buffer size at which to start again a stopped sender. Default: no start-stop at inputs.

BYTEFACT: double, number of simulation time steps needed to serve one byte. Specify either BYTEFACT or SERVICE. See below.

SERVICE: int, number of time steps needed to serve one data object. See below.

RELAX: optional, int. Length of a relaxation period introduced between after each finished service (in simulation time steps). Default: 0.

OUTCTRL: optional, only if BSTART has been given. Control inputs for start-stop protocol. Names can be specified using the same constructs as for

output names of demultiplexers (see Section 7.5 on page 24).
OUT: where to connect multiplexer output (network object inupt name).

Function
A multiplexer with following features:

- Input buffers (same size per input). The input buffer size counts in data objects.
- Start-stop protocol at output (see Section 9.3 on page 44). The corresponding control input is mx->Start.
- Optional start-stop protocol (Section 9.3 on page 44) at inputs. If ‘BSTART’ has been given, then this is turned on, and ‘OUTCTRL’ with the specification of the control inputs of the preceding objects is expected.
- Service time constant or depending on data object size. If ‘BYTEFACT’ has been given, then the service of a data objects takes (BYTEFACT*LenOfDataObject) simulation time steps (rounded to integer, at least 1). If ‘SERVICE’ is specified, then each service takes this number of time steps.
- Optional relaxation period between two services. Turned on with ‘RELAX’.

Exported Variables
`mx->Loss(i)`
Number of data objects lost at input i (int).
`mx->QLen(i)`
Current queue length at input i in data objects (int).
Command
`mx->ResLosses`
Reset all loss counters. No return value.

7.4 Multiplexers with Pure Event-Triggered Scheduling

The following multiplexers are sometimes a bit slower than their counterparts which apply a combination of event- and time-triggered scheduling (Section 7.3 on page 18). The combined approach has advantages if the output line operates at full speed. But if e.g. one output ATM time slot corresponds to 10 simulation time steps, then the pure event-triggered scheduling becomes faster. This also holds in case of very low traffic loads.

Asynchronous Operation
The multiplexer consists of an input queue (capacity: BUFF data items), and a server. Data items are fed into the server whenever the device is free. They then are forwarded SERVICE time steps later. The server therefore emulates a slower output line with asynchronous cycles.
If a data item reaches an empty multiplexer, then it is immediately placed in the server, from where it is forwarded SERVICE steps later. A data item approaching a non-empty system has to wait in the queue.
Synchronous Operation
The multiplexer consists of an input queue (capacity: BUFF data items), and a server. Data items are fed into and taken from the server only at time steps with (SimTime modulo SERVICE = 0). The server therefore emulates a slower output line with synchronous cycles.

If a data item reaches an empty multiplexer with the first time step of an output cycle, then it is immediately placed in the server, from where it is forwarded SERVICE steps later. If a data item reaches an empty multiplexer, but not with beginning of the first step of an output cycle, then it has to wait for service until start of next cycle (while remaining in the queue). Then it is put into the server from where it is forwarded after another SERVICE time steps. A data item approaching a non-empty system has to wait in the queue.

Departure First

DF relates to events taking place during the same simulation time step. If the server completes a service, then first the next data item is taken from the queue (if any), and then new arrivals are queued.

Arrival First

AF relates to events taking place during the same simulation time step. If the server becomes available to begin a service, then first new arrivals are queued, and then the next data item (if any) is transfered from the queue to the server.

When combined with synchronous output operation, this also holds for the case where the data item is transferred from the queue to the server with beginning of a new output cycle, since the item did not arrive with cycle start: the transfer occurs after having queued the current arrivals.

7.4.1 MuxAsyncDF: Asynchronous Output, Departure First

Declaration

As for MuxDF (see Section 7.3.2 on page 19):

```
MuxAsyncDF mux: NINP=10, BUFF=100, MAXVCI=100, SERVICE=4,
    OUT=line;
```

Function

Asynchronous Output, Departure First (specification: see Section 7.4 on page 22)

Exported Variables and Commands

As for Multiplexer (see Section 7.3.1 on page 18).

7.4.2 MuxAsyncAF: Asynchronous Output, Arrival First

Declaration

As for MuxDF (see Section 7.3.2 on page 16):

```
MuxAsyncAF mux: NINP=10, BUFF=100, MAXVCI=100, SERVICE=4,
    OUT=line;
```

Function

Asynchronous Output, Arrival First (specification: see Section 7.4 on page 22)

Exported Variables and Commands

As for Multiplexer (see Section 7.3.1 on page 15).

7.4.3 MuxSyncDF: Synchronous Output, Departure First

Declaration

As for MuxDF (see Section 7.3.2 on page 16):

```
MuxSyncDF mux: NINP=10, BUFF=100, MAXVCI=100, SERVICE=4,
    OUT=line;
```

Function

Synchronous Output, Departure First (specification: see Section 7.4 on page 22)

Exported Variables and Commands
As for Multiplexer (see Section 7.3.1 on page 15).

7.4.4 MuxSyncAF: Synchronous Output, Arrival First

Declaration
As for MuxDF (see Section 7.3.2 on page 16):
```
MuxSyncAF mux: NINP=10, BUFF=100, MAXVCI=100, SERVICE=4,
    OUT=line;
```
Function
Synchronous Output, Arrival First (specification: see Section 7.4 on page 22)
Exported Variables and Commands
As for Multiplexer (see Section 7.3.1 on page 15).

7.5 Demultiplexer

Declaration
```
Demultiplexer demux: MAXVCI=100, NOUT=3,
    OUT=sink[1], sink[2], sink[3];
```
or:
```
Demultiplexer ... OUT=(i: sink[i]);
or:
Demultiplexer ... OUT=(i=1 to 2: sx[i]->Start, 3: snk[i]);
```
MAXVCI: dimension of the routing table, VCI range from 0 to MAXVCI
NOUT: number of outputs (integer)
OUT: input names of succeeding objects (three possibilities: see below)

Function
Incomming cells are forwarded to the output which corresponds to the VCIs of the cells. The routing table can be written with
```
    Signal demux (vciOld, vciNew, outpNum);
          // see Signal object.
```
There are three possibilities to specify the names of the succeeding objects: a complete list, or - if possible - the generation by a template. In the second case above (i: ...), the variable has to be defined in advance and is counted from 1 to NOUT. In the third case (i=...) a list of ranges can be specified. A range is given in the form “i to k”, the first range has to start with 1, the last has to end with NOUT. A range with one member can be written short (only the one number), all ranges in the list are seperated by commas. If a cell with unassigned VCI is received, an error message is generated.
No commands
Input data type
Cell

7.6 Sinks / Delay Line

7.6.1 Sink

Declaration
```
Senke snk;
```
Exported variables

`snk->Count`

  number of arrived cells.

<u>Commands</u>

`snk->ResCount`

  Resets the counter.

<u>Input data type</u>

  Data

7.6.2 Sink Writing IATs to a Trace File

<u>Declaration</u>

`SinkTrace snk: FILE=”iat.dat”;`

All functions as Senke (see Section 7.6.1 on page 24), additionally the inter arrival times are written to the given file (ASCII format, one IAT per line).

7.6.3 Delay Line

<u>Declaration</u>

`Leitung line: DELAY=100, OUT=meas;`

  DELAY: delay of the line (integer, in time slots)
  OUT: input name of the succeeding object

<u>No commands</u>

<u>Input data type</u>

  Data

7.7 Shaping / Policing

7.7.1 Peak Rate Shaper with Integer Cell Spacing

<u>Declaration</u>

`Shaper shap: DELTA=10, BUFF=100, OUT=line;`

  DELTA: cell spacing enforced by the shaper (time slots, integer)
  BUFF: shaper buffer size (cells, integer)
  OUT: input name of the succeeding object

<u>Remark</u>

In case of BUFF=0, a “hard” spacing is performed.

<u>Exported Variables</u>

`shap->QLen`

  current queue length

`shap->Count`

  number of lost cells.

<u>Command</u>

`shap->ResCount`

  Resets the loss counter.

<u>Input data type</u>

  Data

7.7.2 Peak Rate Shaper with Arbitrary Cell Spacing

Declaration

```
Shaper2 shap: DELTA=3.5, BUFF=100, OUT=line;
```

DELTA: effective cell spacing enforced by the shaper (time slots, double)
BUFF: shaper buffer size (cells, integer)
OUT: input name of the succeeding object

Function

The shaper works with 2 spacing values. The ratio of both yields the effective cell rate. In case of BUFF=0, a “hard” spacing is performed.

Exported Variables, Command, Data Types

like for Shaper (Section 7.7.1 on page 25)

7.7.3 Shaper Using the Start-Stop Protocol

Declaration

```
ShapCtrl shap: DELTA=10, BUFF=100, BSTART=10,
        OUTCTRL=src->Start, OUTDATA=line;
```

DELTA: cell spacing enforced by the shaper (time slots, integer)
BUFF: shaper buffer size (cells, integer)
BSTART: buffer occupation, at which the sender is waked up again
OUTCTRL: control input name of the preceeding object
OUTDATA: input name of the succeeding object

Function

This class is similar to the Shaper class. The object tries, however, to control its preceeding object by the Start-Stop protocol (see Section 9.3 on page 44). The data sender is stopped, if the buffer is full. It is started again at a buffer occupation of BSTART. Up to now, the only corresponding object class is GmdpStop (see under Sources). Both are used for the test of new classes.

Exported Variables, Command, Data Types

like for Shaper (Section 7.7.1 on page 25)

7.7.4 Leaky Bucket Policing Function

Declaration

```
LeakyBucket lb: INC=10, DEC=5, SIZE=20, VCI=1, OUT=line;
```

INC: bucket increment with each incomming cell (integer)
DEC: bucket decrement with each time slot (integer)
SIZE: bucket size (integer)
VCI: if given (can be omitted), then only this VC is subject to policing
OUT: input name of the succeeding object

Function

With each time slot, the bucket size is decremented by DEC (until zero). Upon receipt of a cell, it is tested whether the increment would result in an overflow. If yes, then the cell is discarded (no bucket increment). If no, then the increment is performed, and the cell is passed. In case no VCI is given, all cells are subject to policing.
A histogram about bucket sizes seen by arriving “good” cells is maintained. It can be displayed and used in commands.

Remark

The algortihm does *not* coincide with the continuous time algorithm given by ATM Forum TM 4.0. There, the cell passes, and the increment is performed when the bucket contents is not greater than the limit *before* incrementing.

Exported Variables

`lb->Count`
  number of lost cells.

`lb->LbSize`
  current bucket size.

`lb->LbStat(i)`
  number of arriving cells which have seen bucket size i. The size is the size before increment.

Commands

`lb->ResCount`
  Resets the counter.

`lb->ResLbStat`
  Resets LbStat counters.

Input data type
  Data - if no VCI given
  Cell - otherwise.

7.8 AAL 5 Connections

7.8.1 AAL 5 Sender

Declaration

```
AAL5Send aals: VCI=1, BUF=100, BSTART=20,
   OUTDATA=line, OUTCTRL=src;
AAL5Send aals: COPYCID, BUF=100, BSTART=20,
   OUTDATA=line, OUTCTRL=src;
AAL5Send aals: MAXCID=100, BUF=100, BSTART=20,
   OUTDATA=line, OUTCTRL=src;
```

  VCI: VC number of cells generated (int)
  COPYCID given: use the connection ID of the current frame as VCI
  MAXCID: establish a translation table (connection ID) -> (VCI), table length
  BUF: input buffer size (int)
  BSTART: input buffer occupation at which to wake up stopped data sender (int)
  OUTDATA: where to send cells
  OUTCTRL: control input of preceeding network object

Function

The object implements the AAL5 sender functionality: Incoming data items which have to provide the interface for data item embedding (Section 9.2 on page 44) are transformed into sequences of ATM cells. The ATM cells contain sequence numbers for both cells and AAL SDUs. This allows the receiver to detect cell losses. Thus, only uncorrupted AAL SDUs are extracted from transmitted cells and delivered to the data receiver. If MAXCID has been given, then the translation table can be filled using command SetVCI().
On input and output, the start-stop protocol (see Section 9.3 on page 44) is supported. The data input is `aals->Data`, the control input is `aals->Start`.

Commands

`aals->ResetStat`

Resets all statisitcs counter.

`aals->SetVCI(cid, vci)`

Write translation table: connection ID cid is converted in VCI vci.

Exported Variables

For use in commands and with display devices, the following variables are exported:

`aals->QLen`

current input queue length

`aals->CellCount`

number of cells sent

`aals->SDUCount`

number of AAL SDUs sent

`aals->DelCount`

number of incoming data items dropped since start-stop not recognized

Input Data Type

Data on inputs ->Data and ->Start.

Output Data Types

OUTDATA: AAL5Cell
OUTCTRL: Data

7.8.2 AAL 5 Receiver

Declaration

`AAL5Rec aalr: OUT=tcpr;`

OUT: where to send reconstructed data items

Function

Data items segmented by the corresponding AAL 5 sender are reassembled. Cell and SDU losses are detected using sequence numbers: The incomming frame is embedded into the first cell of the burst generated by the sender. Additionally, the first cell carries the SDU sequence number. All cells contain a cell sequence number, therefore the receiver can detect cell loss and flushes its input cell queue when detecting cell loss. The last cell is marked with PT = 1, and a field of this cell repeats the cell sequence number of the first cell of the burst. The receiver checks whether the first cell in the input queue bears this sequence number. If yes, then the frame has been transmitted succesfully and it can be extracted from the first cell. Otherwise, all cells in the queue are dropped. Thanks to the SDU sequence number, the number of lost SDU also can be detected.

Commands

`aalr->ResetStat`

Resets all statisitcs counter.

Exported Variables

For use in commands and with display devices, the following variables are exported:

`aalr->QLen`

current input queue length

`aalr->CellCount`

number of cells received

`aalr->SDUCount`

number of AAL SDUs successfully transmitted

`aalr->CellLoss`

number of cells lost

`aalr->SDULoss`
number of AAL SDUs lost
`aalr->DelayMean`
Mean SDU transfer delay (measured from sending first until receiving last cell of SDU)

Input Data Type

AAL5Cell

Output Data Type

The output data type is the type which has been encapsulated by the corresponding AAL 5 sender.

7.8.3 AAL 5 Receiver with Concurrent Reassembly

Declaration

```
AAL5RecMult aal: MAXVCI=100, OUT=tcp;
```

MAXVCI: int, largest VCI number
OUT: where to send packets

Function

The function is the same as for `AAL5Rec` (Section 7.8.2 on page 28), but multiple frames can be reassembled concurrently. This eases the definition of AAL end systems over which more than one higher-layer connection is set up. Otherwise, the streams had to be demultiplexed, and for each connection an AAL5Rec object would be necessary.

Commands

aal->ResetStat
Resets all statistics counters.

Exported Variables

For use in commands and with display devices, the following arrays are exported. The parameter `vc` is the ATM connection (1 ... MAXVCI).

`aal->QLen(vc)`
current cell queue length
`aal->CellLoss(vc)`
number of cells lost
`aal->CellCount(vc)`
number of cells received
`aal->SDULoss(vc)`
number of SDUs lost
`aal->SDUCount(vc)`
number of SDUs succesfully transmitted
`aal->DelayMean(vc)`
mean SDU delay (from sending first cell until receiving the last)

7.9 CTD and IAT Measurements

7.9.1 Measurement Device

Declaration

```
Meas2 ms: MAXCTD=100, MAXIAT=100, VCI=1, OUT=line;
```

MAXCTD: largest cell transfer delay which can be measured
MAXIAT: largest cell inter arrival time which can be measured
VCI: if given (can be omitted), only cells on this VC are recognized

OUT: input name of the succeeding object, can be omitted

Function

The object generates statistics over cell transfer delays and inter arrival times. If no OUT is given, it acts as sink. The VCI can also be omitted.

Remark

The device fails at Sim->ResetTime.

Exported Variables

`ms->Count`

number of arrived cells.

`ms->CTD(time)`

cell counter for this transfer delay.

`ms->CTDover`

number of CTD values larger than CTDMAX.

`ms->IAT(time)`

counter for this inter arrival time.

`ms->IATover`

number of IAT values larger than IATMAX.

Commands

`ms->ResCount`

Resets the counter.

`ms->ResDists`

Resets all counters.

The following commands are provided for convenience, they could be realized in the input file or by macros, too. They only evaluate the stored distributions (values up to MAXCTD and MAXIAT).

```
ms->MeanIAT
ms->MeanCTD
```

Return mean values of the recorded distributions.

```
ms->MinIAT
ms->MinCTD
ms->MaxIAT
ms->MaxCTD
```

Return the minimum and maximum IAT and CTD values with counters larger than zero (only inside of the specified MAX borders).

Input data type

Data - if no VCI given

Cell - otherwise.

7.9.2 More Complex Measurement Device for Cells and Frames

Declaration (full version):

```
Meas3 ms: CTD=(min,max), CTDDIV=10, IAT=(min,max),
       IATDIV=10, VCI=(min,max), ERANGE, OUT=sink;
```

Function

All parameters are optional, but one parameter has to be given at least. If OUT is omitted, the the object acts as sink. The device expects cells, if ‘VCI’ is given. Frames can be measured, if the word ‘VCI’ is replaced by ‘CONNID’.

If the keyword ‘CTD’ is given, then mean values and extreme values (min/max) of the transfer delays are measured (either measured from the source of the data object or from the last setting of the time stamps, see Section 7.9.3 on page 32). In case '=(min,max)' is added to 'CTD', then a complete distribution between min and max is collected. The specification of 'CTDDIV=`div`' (optional, only if complete distribution collected) says that all measured values are diveded by `div`, before they are counted in the distribution. Note, however, that the mean and extreme values (commands MeanCTD, MinCTD, MaxCTD) are not influenced by this down-scaling. The parameters 'IAT' and 'IATDIV' follow the same rules. The 'VCI' or 'CONNID=(min,max)' specifies the range of channel IDs over which the measurements are performed. If neither of both is given, then the range is (0,0). If 'ERANGE' is given (optional), then an error message is generated if a data object with out-of-range ID arrives. Otherwise, only the global counter 'Count' is incremented.

Commands

`ms->ResStats`
    Resets all CTD and IAT statistics and all counters, no return value.

`ms->ResCount`
    Resets the global counter `Count`, no return value.

Exported Variables

For use in commands and with display devices, the following arrays are exported.

`ms->Count`
    overall number of arrivals (int).

`ms->Counts(i)`
    number of arrivals for this index (int)

`ms->CTD(i, tim)`
    counter for cell transfer delay time `tim`, scaled by CTDDIV (int)

`ms->IAT(i, tim)`
    counter for inter arrival time `tim`, scaled by IATDIV (int)

`ms->CTDover(i)`
    overflow counter for CTD distribution (int)

`ms->CTDunder(i)`
    underflow counter for CTD distribution (int)

`ms->IATover(i)`
    overflow counter for IAT distribution (int)

`ms->IATunder(i)`
    underflow counter for IAT distribution (int)

`ms->MeanCTD(i)`
    current cumulative mean CTD value (double). Is updated 'on-the-fly' with each arrival. Note the difference to the command of Meas2 (Section 7.9.1 on page 29).

`ms->MaxCTD(i)`
    largest encountered CTD (int). Same notes as for MeanCTD.

`ms->MinCTD(i)`
    smallest encountered CTD (int). Same noteas for MeanCTD.

`ms->MeanIAT(i)`
    current cumulative mean IAT value (double). Same Note as for MeanCTD.

`ms->MaxIAT(i)`
    largest encountered IAT (int). Same note as for MeanCTD.

`ms->MinIAT(i)`
    smallest encountered IAT (int). Same note as for MeanCTD.

Input Data Type
Cell - if VCI given
Frame - if CONNID given
Data - if neither of both given

7.9.3 Updating Time Stamps

Declaration
```
TimeStamp ts: VCI=1, OUT=line;
```
Function
Updates time stamps in all data items, or only on the given VC (VCI can be omitted).
Input data type
Data - if no VCI given
Cell - otherwise.

7.10 ABR - ATM-Forum TM 4.0

7.10.1 ABR Source

Declaration
```
AbrSrc src:BUFF=100, BSTART=50, LINKCR=2.2e5, MCR=200,
     PCR=1000, FRTT=1000, ROUTE=(2, abrmx, abrsnk),
     AUTOCONN, OUTCTRL=src->Start, OUTDATA=line;
```
BUFF: input buffer size (int)
BSTART: buffer occupation at which to start again the data source (int)
LINKCR: optional: output link cell rate (cells per second, double), default: 353207.55
MCR: minimum cell rate (cells per second, double)
PCR: peak cell rate (cells per second, double)
FRTT: fixed round tripp time (micro seconds, double)
ROUTE: ABR routing members, see below
AUTOCONN: optional: if given, connection establishment with first incomming cell
OUTCTRL: control input of the data source (start-stop protocol)
OUTDATA: ABR cell output

Function
The class AbrSrc implements the reference source behaviour of ATM-Forum TM 4.0. All parameters not specified are set to the default values given in TM 4.0. The source possesses four inputs:

- input `src`: data input
- input `src->BRMC`: input for backward RM cells
- input `src->Start`: an incomming cell causes connection establishment
- input `src->Stop`: an incomming cell causes connection release

If the output link rate LINKCR is not given, then it is set to the equivalent of 149.76 Mbps. The ROUTE contains first the number of ABR members and then the members itself. These objects are contacted (in the given order) at first cell arrival - in case AUTOCONN is given -, or at request for connection establishment on input src->Start. In both cases, a connection with the VCI of the incomming cell is established. Since

FRTT in the given version can not be determined during connection setup, it has to be given explicitly.
The ABR source needs a data source implementing the start-stop protocol (e.g. GmdpStop, see Section 9.3 on page 44). Otherwise, error messages are generated upon input buffer overflow.
The exact internal behaviour of the source (timer actions) are documented in the source code file abrsrc.c.
Exported Variables
The following variables are exported for use with commands or measurement devices:
- `src->CountData`: number of data cells sent, int
- `src->CountRMCI`: number of in-rate RM cells sent, int
- `src->CountRMCO`: number of out-of-rate RM cells sent, int
- `src->QLen`: current input queue length, int
- `src->IAT`: current IAT, int (changes permanently due to non-interger cell spacing)
- `src->ACR`: current ACR, double

Data types
At `src, src->Start,` and `src->Stop,` cells are expected. At `src->BRMC,` RM cells are expected.

7.10.2 ABR Multiplexer

Declaration
```
AbrMux mux: NINP=10, MAXVCI=100, BUFFCBR=200,
    BUFFABR=10000, BUFFRMCO=1000, HI_THRESH=7000,
    LO_THRESH=5000, TBE=2000, AI=30, CBRI=100, ZOL=12.5,
    LINKCR=2.2e5, TARGUTIL=0.9, DYNFAIRSHARE, BINMODE,
    OUTBRMC=dmx, OUTDATA=line;
```
NINP: number of inputs, int
MAXVCI: largest VCI number possible, int
BUFFCBR: buffer size for non-ABR traffic, int
BUFFABR: overall buffer for ABR traffic, int
BUFFRMCO: optional: max. number of out-of-rate cells to be stored, int (default: BUFFABR)
HI_THRESH: optional: turn congestion indication on, int (default: BUFFABR)
LO_THRESH: optional: turn congestion indication off, int (default: HI_THRESH)
TBE: transient buffer exposure for connection negotiation, int
AI: measurement interval for ABR traffic (time slots, int)
CBRI: optional: measurement interval for non-ABR traffic (time slots, int), default: AI
ZOL: optional: z-value in case no ABR bandwidth available. default: 1000
LINKCR: optional: output link cell rate (cells per second, double), default: 353207.55
TARGUTIL: target link utilisation (double)
DYNFAIRSHARE: optional: use number of active connections for calculations (see below)
BINMODE: optional: no ER feedback is written into RM cells (“pure” CI bit mode)
OUTBRMC: where to send backward RM cells
OUTDATA: multiplexer output

Function

The AbrMux is an ABR multiplexer with ERICA and binary (CI bit) feed back and backward RM cell processing. ABR and non-ABR traffic use different buffers. All ABR traffic (virtually) shares one buffer, but the amount of out-of-rate RM cells can be limited. The multiplexer serves cells in the following order:

1. non-ABR cells
2. in-rate RM cells
3. out-of-rate RM cells
4. ABR data cells

Thus, RM cells overtake ABR data cells. The binary feedback watermarks are related only to the ABR buffer. During congestion indication, the CI bit is set in forward and backward direction.

For the integration of non-ABR, the current non-ABR load is measured (interval: CBRI time slots, default: AI). The fair and VC share values of the ERICA algorithm then are computed as follows. For the fair share, the difference of target rate (LINKCR times TARGUTIL) and current non-ABR rate is divided by the number of ABR connections. It is set to zero in case no ABR bandwidth available. The z-value for the VC share is the ratio of current ABR input rate (measured over AI time slots) and the difference between target and non-ABR input rate. Z is set to 1000 in case no ABR bandwith available, or to ZOL if specified in the definition statement. The current rate of a connection is taken from the CCR of forward RM cells, the current ER of a connection is included in both forward and backward RM cells.

If DYNFAIRSHARE is given, then the fair share computation is based on the number of connections which have sent during the last AI interval (no averaging with previous values is performed). Without DYNFAIRSHARE, the fair share results from the number of established connections (regardless wether actually sending or not).

In BINMODE, no ER feedback is filled into RM cells. Therefore, the multiplexer behaves like a device with only binary feedback.

During connection setup (initialized by the ABR source), only a check of the sum of the current MCRs against the target rate is performed. The current non-ABR load is not taken into account. For computing the initial cell rate, also the ERICA algorithm is used: CCR is set to MCR in this case.

The multiplexer has NINP + 1 inputs:

- `mux->I[i] ... mux->I[NINP]`: forward data inputs
- `mux->BRMC`: input for backward RM cells

<u>Exported Variables</u>

An object exports the following variables for use with commands or measurement devices:

- `mux->QLenNABR`: current non-ABR queue length, int
- `mux->QLenABR`: current ABR queue length (all kinds of ABR cells), int
- `mux->QLenABRData`: current ABR data cell queue length, int
- `mux->QLenRMCI`: current in-rate RM cell queue length, int
- `mux->QLenRMCO`: current out-of-rate RM cell queue length, int
- `mux->LossData(vci_no)`: number of cells lost on this VC (non-ABR or ABR data), int
- `mux->LossRMCI(vci_no)`: number of in-rate RM cells lost on this VC, int
- `mux->LossRMCO(vci_no)`: number of out-of-rate RM cells lost on this VC, int
- `mux->CRNABR`: current non-ABR input cell rate, double
- `mux->CRABR`: current ABR input cell rate (all kinds of cells), double
- `mux->Z`: current z-value (ERICA algorithm), double

<u>Data types</u>

The inputs `mux->I[x]` expect Cell and RMCell , on `mux->BRMC` RMCell is expected.

7.10.3 ABR Sink

<u>Declaration</u>

```
AbrSink sink: BUFF=1000, TARGBUFF=500, HI_THRESH=800,
     LO_THRESH=600, AI=100, LINKCR=2.2e5, TARGUTIL=0.9,
     OUTBRMC=line, OUTDATA=aal5;
```

BUFF: output buffer size (int), *!! ATTENTION !!* see below.
TARGBUFF: optional: buffer occupation at which to decrease ER, default: BUFF/2
HI_THRESH: optional: turn congestion indication on, int (default: BUFF)
LO_THRESH: optional: turn congestion indication off, int (default: HI_THRESH)
AI: optional: measurement interval for output rate measurement, default: 30
LINKCR: optional: output link cell rate (cells per second, double), default: 353207.55
TARGUTIL: optional: ER reduction factor, see below. Default: 1.0
OUTBRMC: where to send backward RM cells
OUTDATA: where to send data cells

<u>Function</u>

AbrSrc implements an ABR sink which also can act as virtual sink. The sink function is very simple: RM cells are reflected towards the SES (DIR bit set, BN bit cleared). Data cells are passed to the next object.

!! ATTENTION !!:

Also if the next object never stops the ABR sink, the BUFF value limits the TBE value negotiated during connection setup. So set BUFF high enough in any case.

<u>Backpressure:</u>

The binary feedback is performed as in the ABR multiplexer described. If the current output buffer occupation reaches TARGBUFF (due to stop send caused by the data sink or next ABR control loop), ER is reduced to TARGUTIL times the current output link rate. The latter one is measured over AI time slots. For the start-stop protocol (see Section 9.3 on page 44), the source has an input src->Start.

<u>Exported Variables</u>

An object exports the following variables for use with commands or measurement devices:

- `sink->Qlen`: current output queue length, int
- `sink->Loss`: number of lost cells (output buffer), int
- `sink->CountData`: number of ABR data cells passed, int
- `sink->CountRMCI`: number of in-rate RM cells reflected, int
- `sink->CountRMCO`: number of out-of-rate RM cells reflected, int
- `sink->CROut`: current output cell rate, double

<u>Data Types</u>

Input `sink` expects Cell and RMCell, at `sink->Start` Data is accepted.

7.11 TCP Connections

7.11.1 TCP Sender

<u>Declaration</u>

```
TCPIPsend tcps: BUF=<int>, BSTART=<int>, MTU=<int>,
    TS=<0/1>, NAGLE=<0/1>, FRETR=<0/1>, BITRATE=<double>,
    PROCTIM=<double>, TICK=<double>, RTOMIN=<double>,
    OQWM=<int>, LOGRETR=<0/1>, REC=<object>,
    OUTDATA=<object input>, OUTCTRL=<object input>;
```

BUF: input buffer size (bytes)
BSTART: optional: buffer occupation where to wake up sender, default: BUF-4096
MTU: optional: maximum output frame size (bytes), default: 9180 bytes
TS: optional: time stamp option (RFC1323) on/off, default: on
NAGLE: optional: Nagle’s algorithm (RFC1122) on/off, default: on
FRETR: optional: fast retransmission and recovery on/off, default: on
BITRATE: optional: ATM layer bit rate (Mbit/s), default: 149.76 Mbit/s
PROCTIM: optional: processing time per sent packet (msec), default: 0.3msec
TICK: optional: length of TCP clock tick (msec), default: 500msec
RTOMIN: optional: minimum retransm. timeout (msec), default: 1500msec
OQWM: optional: maximum output queue length (packets), default: 2 packets
LOGRETR: optinal: print retransmission messages on/off, default: off
REC: name of the peer TCP receiver object
OUTDATA: where to send TCP frames
OUTCTRL: control input of the preceeding network object

Function

The object implements the TCP sender functionality. The IP part only is reflected by 20 bytes IP overhead in each packet generated. On the input `tcps->Data`, frames containing a length indicator are expected. On this input, a network object following the start-stop protocol (see Section 9.3 on page 44) is expected: it is stopped in case the input buffer is full. The frames are segmented, and appropriate TCP/IP frames are generated. The TCP sender also can be stopped by its succeeding network object (start-stop protocol, the start input is `tcps->Start`). The input expecting the acknowledgement packets is `tcps->Ack`.
The following algorithms are implemented by the object class:

- Slow start and congestion avoidance
- Silly window syndrome (SWS) avoidance according to RFC 1122, section 4.2.3.4. The implementation differs slightly: condition (3) in this section is not subject to Nagle’s algorithm, i.e. a segment of half of the receiver’s buffer size is sent always.
- Nagle’s algorithm according to RFC 1122, section 4.2.3.4. See remarks to SWS.
- Karn’s algorithm (no RTT measurement during retransmissions)
- Fast retransmission and recovery
- RTT measurement with time stamp option according to RFC 1323

After connection setup, both TCP sender and receiver know the identity of the peer object. The identifiers are written in each frame sent (data and ACK frames). Thus checks can be performed, whether an arriving frame stems from the peer object or whether it has arrived due to erroneous routing (problem in large network configurations). Error messages are launched in this case.

Remarks

The object DOES NOT YET IMPLEMENT the zero windo probe: if an acknowledgement packet reopening the receiver window is lost, then the connection stops for ever. To model the socket interface (input `tcps->Data`) in a useful way, the sending object already has to split available data into pieces of e.g. 4096 bytes. This is necessary since the start-stop protocol does not (yet) allow a partial reject of a received data item.

Exported Variables
An object exports the following variables for use with commands or measurement devices:
`tcps->NXT`: data sequence number next to send (bytes)
`tcps->UNA`: first unacknowledged data sequence number (bytes)
`tcps->RTT`: estimated round trip time in ticks
`tcps->RTTreal`: “real” (measured in the simulator) round trip time in time slots
`tcps->RTOcalc`: retransmission timeout value most recently calculated from RTT
`tcps->WND`: receiver window size currently seen (bytes)
`tcps->WND_MIN, tcps->WND_MAX`: min an max `WND`
`tcps->CWND`: current congestion widow size (bytes)
`tcps->INPQ, tcps->INPQ_LEN`: current input queue length (bytes)
`tcps->INPQ_MAX_LEN`: max. `INPQ_LEN`.
`tcps->PRCQ_LEN`: current processing queue length (queue models `PROCTIM`)
`tcps->PRCQ_MAX_LEN`: max. `PRCQ_LEN`
`tcps->REXMTO`: number of retransmission timeouts
`tcps->RECEIVED_BYTES`: number of bytes received from the application
`tcps->XMITTED_BYTES`: total number of bytes sent (with headers and retransm.)
`tcps->XMITTED_USER_BYTES`: the same, but without headers
`tcps->XMITTED_SEGMENTS`: overall number of packets sent
`tcps->REXMITTED_BYTES`: total number of retransmitted bytes (headers included)
`tcps->REXMITTED_SEGMENTS`: number of packets retransmitted
`tcps->RECEIVED_ACKS`: number of received ack packets
`tcps->REX_PERC`: current percentage of retransmissions
Commands
The command `tcps->ResetStat` resets the following variables:
```
INPQ_MAX_LEN, PRCQ_MAX_LEN, REXMTO, RECEIVED_BYTES,
XMITTED_BYTES, XMITTED_USER_BYTES, XMITTED_SEGMENTS,
REXMITTED_BYTES, REXMITTED_SEGMENTS, RECEIVED_ACKS,
REX_PERC.
```
Data types
Input tcps->Data: Frame
Input tcps->Start: Data
Input tcps->Ack: TCPAcknowledge
Output OUTDATA: TCPIPFrame
Output OUTCTRL: Data

7.11.2 TCP Receiver

Declaration
```
TCPIPrec tcpr: WND=<int>, PROCTIM=<double>,
    ACKDEL=<double>, IACKDEL=<double>,
    OUTDATA=<object input>, OUTACK=<object input>;
```
WND: receiver buffer size, including resequencing queue (bytes)
PROCTIM: optional: processing time per received packet (msec), default: 0.3msec
ACKDEL: optional: delay of “delayed” acknowledgmts (msec), default: 200 msec
IACKDEL: optional: delay of “immediate” acks (msec), default: PROCTIM msecs
OUTDATA: where to send received data
OUTACK: where to send acknowledgement frames
Function

The object implements the TCP receiver functionality. TCP/IP packets are expected at the input `tcpr->Data`. The object can be stopped by the succeeding network object which stops eventually the window update process of the receiver (start-stop protocol, input `tcpr->Start`). During connection setup, the object is informed by the peer sender about ATM bit rate (to translate seconds into slots), the time stamp option, and the maximum segment size. The sender, in turn, is provided with the maximum window size. In case turned on by the peer sender, the time stamp option according to RFC 1323 is supported.
A delayed acknowledgement is registered with every packet received in-sequence. Immediate ACKs are sent:

- if an out-of-sequence packet arrives,
- if the receiver window closes to zero,
- if a zero window probe is received, and
- if the right window edge shifts by at least one maximum-sized segment or half of the total receiver buffer size.

After connection setup, both TCP sender and receiver know the identity of the peer object. The identifiers are written in each frame sent (data and ACK frames). Thus checks can be performed, whether an arriving frame stems from the peer object or whether it has arrived due to erroneous routing (problem in large network configurations). Error messages are launched in this case.

Remark
RFC 1122, section 4.2.3.2 recommends to send an ACK for at least each second full-sized segment. This currently is only realised indirectly via the immediate ACK launched by the subsequent window update. In case the receiver has been stopped by the successor, this algortihm will fail (no window update will occur as long as the receiver remains stopped).
Exported Variables
An object exports the following variables for use with commands or measurement devices:
`tcpr->THROUGHPUT`: mean throughtput since first packet sent by sender (bit/s)
`tcpr->SDU_DELAY`: mean TCP packet delay (slots)
`tcpr->RESQ_LEN`: current length of the resequencing queue (packets)
`tcpr->RESQ_MAX_LEN`: max. `RESQ_LEN`
`tcpr->PRCQ_LEN`: current length of the processing queue (packets)
`tcpr->PRCQ_MAX_LEN`: max. `PRCQ_LEN`
`tcpr->ACK_CNT`: number of ACKs sent
`tcpr->WND`: current window size (bytes, not yet necessarily advertised)
`tcpr->ARRIVED_SEGMENTS`: total number of arrived packets
`tcpr->ARRIVED_BYTES`: total number of arrived bytes (without headers)
`tcpr->ARRIVED_VALID_BYTES`:
bytes arrived in the advertised window (between nxt and nxt+wnd)
`tcpr->USER_PACKETS`: number of packets sent to the successor
`tcpr->USER_BYTES`: number of bytes forwarded to the successor
`tcpr->OUT_OF_ORDER_SEGMENTS`: number of packets received out-of-order
`tcpr->OUT_OF_ORDER_BYTES`: bytes received out-of-order (without headers)
Commands
The command `tcpr->ResetStat` resets the following variables:
`ARRIVED_SEGMENTS, ARRIVED_BYTES, ARRIVED_VALID_BYTES,`
`USER_PACKETS, USER_BYTES, PRCQ_MAX_LEN, RESQ_MAX_LEN,`

`OUT_OF_ORDER_SEGMENTS, OUT_OF_ORDER_BYTES, ACK_CNT, THROUGHPUT, SDU_DELAY`

Data types

Input tcpr->Data: TCPIPFrame
Input tcpr->Start: Data
Output OUTDATA: Frame
Output OUTACK: TCPAcknowledge

7.11.3 TCP Application: Constant Frame Distances

Declaration

```
CBRFrame src: DELTA=<int>, LEN=<int>, StartTime=<int>,
    EndTime=<int>, BYTES=<int>, CONNID=<int>,
    OUT=<object input>;
```

DELTA: distance between two generated frames (time slots)
LEN: length of one frame (bytes)
StartTime: optional, when to start to send (see below), default: DELTA
EndTime: optional: when to stop to send (time slot), default: (virtually) infinite
BYTES: with wich byte stop to send, default: (virtually) infinite
CONNID: optional: layer 4 connection ID of the generated frames, default: 0
OUT: where to send frames generated

Function

This object class can be used to model bulk data transfer for TCP (see Section 7.11.1 on page 35). Every DELTA-th time slot, a frame of length LEN is sent. The object recognizes the start-stop protocol (Section 9.3 on page 44), the control input for this purpose is `src->CTRL`. If StartTime is given and greater than zero, the object begins to send at this instant. In case StartTime is set to zero, then the object begins to send between time slot 0 and 500’000 (random choice, equally distributed).

Output data type

Frame

7.11.4 TCP Application: Arbitrary Frame Distances

The object class `Data2Frame` converts each incoming data item (e.g. cell) into a frame with constant length. Hence, it is possible to use the variety of cell sources for the generation of frames.

Declaration

```
Data2Frame d2f: FLEN=20, CONNID=10, OUT=tss;
```

`FLEN`: length of generated frames (int)
`CONNID`: optional: layer-4 identifier, default: 0
`OUT`: where to send frames

In order to connect this object to a TCP sender or another object relying on the Start-Stop protocol, an interface object has to be placed between frame source and TCP input. This interface `TermStartStop` has not been integrated into `Data2Frame` since it is useful also for other purposes. When the predecessor continues to send, if the sucessor has stopped the interface, then simply the input buffer overflows. But the interface behaves according to the Start-Stop protocol at its output. The object has a control input `tss->Start`.

Declaration
```
TermStartStop tss: BUF=100, OUT=tcpipsend->Data;
```
`BUF`: input buffer size in frames (int)
`OUT`: where to send frames

Commands
`tss->Count` is the number of lost data items. `tss->ResCount` resets this counter.

7.12 Miscellaneous

7.12.1 Signalling source

This is more a command, since no object is generated. The routing table of the specified object is written.

Declaration
```
Signal demux (oldVCI, newVCI, outpNo), ...;
```
Example:
```
Signal demux (1,2,1), (7,8,2);
```
Cells with VCI 1 are passed to output 1, the VCI is changed to 2.
Cells with VCI 7 are passed to output 2, the VCI is changed to 8.
The number of signalling messages specified with one Signal statement is arbitrarily. Since the signalling is performed instantly, the destination object has to be defined in advance.

7.12.2 Data type and timing checks

For purposes of debugging:
```
TypeCheck tc: TYPE=RMCell, VCI=1, OUT=demux;
```
TYPE: data item type
VCI: if given (can be omitted), only data items on this VCI are checked
OUT: input name of the succeeding object

Function
The object ensures that all passing data items are of the specified class or of a class which is derived from the given one. Violating data items cause an error message. Furthermore, data items are accepted only in the early slot phase, and receiving more than one item per time slot causes an error message.

No commands

7.12.3 Dummy Connection Object

When using e.g. loops to describe networks structures, then sometimes some names do not fit at all into the rule which is given for the whole loop. To “translate” names, this dummy object is usefull. It really does nothing than forwarding an incomming data item to its output.
```
DummyObj dmy: OUT=line;
```

7.12.4 A Class Implementing Non-Local Variables

Variables are local to the enclosing block. It is therefore not possible to create auxiliary variables with macros. Since objects are not local to the block of definition, a special object class may solve some problems.

```
     ValArray arr: LEN=10;
```

This defines an object containing an array of double values. The values can be written with `arr->Set(index, value)`, where index is an integer between 0 and LEN-1. The values can be read with `arr->Get(index)`.

8.0 Graphical Online Displays

8.1 Interactive Control Window

Declaration

```
Control: POS=(50,50), DELTA=100, SLEEP=100000,
       FILE=”xx.pos”, CORR=(3,20);
```

POS: screen position of the control window (integer)
DELTA: number of time slots between to activations (integer)
SLEEP: time enforced between 2 activations (micro seconds, integer)
FILE: optional: file where positions and dimensions of graphical objects are stored
CORR: optional: correction of window shift caused by window manager frames

Function

The object generates a window entitled “YATS” which displays the current simulation time, and the current state (running / stopped). The initial state is “stopped”. Mouse clicks on the window area toggle between the states. Window update and test on mouse events are performed with every DELTA-th time slot. If SLEEP is given (it can be omitted), then a pause is generated after each window update, if the time expired since the last update is smaller than the specified one.

If the optional FILE is given, then all changes of window dimensions and postions are stored in this file. These stored values lateron override the specifications of the normal input text.

ATTENTION: The position file only effects objects which are defined after the Control window definition.

The CORR parameter (default: (5,30) according to Solaris 2.5 DCE manager) corrects the window shift caused by the window manager’s window frames. If window positions move between two calls of YATS, then look into the position file and observe changes in the window positions. You then can specify the appropriate CORR values.

Remarks

The object is activated during the early slot phase, and it registers at the event scheduler like every other object. Therefore, it is unspecified which other events destined for the current time slot have been activated already - and which not. Graphical online displays, however, normally register for the late slot phase. Thus, they may display an old value. By choosing appropriate sample distances, it can be ensured that they are synchronized with each other when stopping the simulator.

8.2 Sliding Time History of a Value

Declaration

```
Meter meter: VAL=src->Count, TITLE=”Cell Rate in src”,
     WIN=(100,100,400,200), MODE=DiffMode, NVALS=200,
     MAXVAL=100, DELTA=100, UPDATE=10;
```

VAL: object and variable name to be displayed
TITLE: window title, can be omitted (then the object name is used)
WIN: (xPos, yPos, width, height) (integer)
MODE: DiffMode or AbsMode (see below)
NVALS: number of displayed values (history length) (integer)
MAXVAL: values are normalized to MAXVAL
DELTA: number of time slots between two samples (integer)
UPDATE: the window is updated only with every UPDATE-th sample (can be omitted)

Function

The object specified by VAL is asked for the address of the given variable. Its contents then is read during every DELTA-th time slot. The time history comprises NVALS samples, the full window height corresponds to MAXVAL. The mode AbsMode displays the samples itself, DiffMode displays the differences between subsequent samples. If UPDATE is given (can be omitted), then the window is updated only with every UPDATE-th sample.
On a mouse click on the window area, a file name to store the current history content (ASCII format) is asked for.

Remarks

The object registers at the event scheduler for the late slot phase. Therefore, values modified during the late phase can not be displayed exactly with a resolution of one time slot. To enable the display of variables, the data source has to implement the export() method (see object methods).

8.3 Histogram of a Distribution

Declaration

```
Histogram histo: VALS=meas->IAT, TITLE=”IAT Distribution”,
     WIN=(100,100,400,200), MAXFREQ=0.2, DELTA=100;
```

VALS: object and array name to be displayed
TITLE: window title, can be omitted (then the object name is used)
WIN: (xPos, yPos, width, height) (integer)
MAXFREQ: normalisation, corresponds to the window height (double, can be omitted)
DELTA: number of time slots between two samples (integer)

Function

The object specified by VALS is asked for address and dimension of the given array. The contents of the array is interpreted as a distribution (the values are divided by the sum of them). Sampling and display are performed during each DELTA-th time slot. If MAXFREQ is not specified, then the display normalisation is done automatically (the window height corresponds to the largest frequency times 1.25).
On a mouse click on the window area, a file name to store the current distribution content (ASCII format) is asked for.

Remarks

See remarks for the Meter object.

8.4 Production of a Distribution from Samples of a Value

Declaration

```
Histo2 histo: VAL=mux->QLen, TITLE=”Queue Length Distrib”,
     WIN=(100,100,400,200), NVALS=52, MAXFREQ=0.2,
     DELTA=1, UPDATE=1000;
```

VAL: value from which to produce the distribution
TITLE: window title, can be omitted (then the object name is used)
WIN: (xPos, yPos, width, height) (integer)
NVALS: length of the distribution. Value range: 0 ... NVALS-1
MAXFREQ: normalisation, corresponds to the window height (double, can be omitted)
DELTA: number of time slots between two samples (integer, can be omitted)
UPDATE: the window is updated only with every UPDATE-th sample (can be omitted)

Function

The object specified by VAL is asked for the address of the given variable. Its contents then is read during every DELTA-th time slot. The produced distribution comprises NVALS values. If UPDATE is given (can be omitted), then the window is updated only with every UPDATE-th sample. If MAXFREQ is not specified, then the display normalisation is done automatically (the window height corresponds to the largest frequency times 1.25).
On a mouse click on the window area, a file name to store the current distribution content (ASCII format) is asked for.

Exported Variables

```
histo->Dist(i)
```

frequency counter of value i.

Commands

```
histo->ResDist
```

Resets all counters

Remarks

See remarks for the Meter object.

9.0 Data Object Classes

9.1 Used Data Object Classes, Derivation Relations

Different network object classes communicate via different data object classes. Currently, the following data classes are defined:

1. Data
   Base class. Only contains a time stamp (generation time). Contains a hook to embed other data objects.
2. Cell
   ATM cell, derived from Data. Contains a VCI number.
3. RMCell
   ABR ressource management cell, derived from Cell. Contains RM data.

4. AAL5Cell
   Cell with payload, carrying cell and AAL SDU sequence numbers. Derived from Cell. Normally is used to embed other data objetcs.
5. Frame
   Data frame only containing a length indicator (used for TCP, Section 7.11 on page 35). Derived from Data.
6. TCPIPFrame
   PDU of TCP connections (Section 7.11 on page 35). Derived from Frame.
7. TCPAcknowledge
   PDU of TCP, carrying acknowledgement information. Derived from Frame.

The check, whether the data item received by a network object is the same as expected or derived from it, is performed at run-time. The mechanism is implemented as integer table lookup and costs not more than 5 % speed, but gives maximum flexibility: Network objects processing e.g. class Data do not complain about receiving Cell. In contrast, a Data item incomming at the backward RM cell input of an ABR source causes an error message.

9.2 Data Object Embedding

The specification of the “Data” object class defines a mechanism to embedd data objects into other objects. This allows to implement protocol layers which are transparent for the next higher layer. The first example will be the implementation of AAL5 connections which can “tunnel” arbitrary data objects, provided they implement the methods for embedding.

9.3 Start-Stop Protocol

Sometimes a loss-free transmission of data objects between network objects is necessary, e.g. for the communication between higher protocol layers. The data receiver should be able to stop and start the data sender depending on its internal state or even the state of the succeeding network object. For this purpose, the start-stop protocol has been defined. As a side-effect of delivering the data item, the sender is informed whether the receiver can accept more data. If not, then the sender has to stop. It is informed by the receiver via an extra input - often called Start - that more data can be sent. The protocol currently is used by ABR objects (Section 7.10 on page 32), the source GmdpStop (Section 7.1.7 on page 16), AAL 5 objects (Section 7.8 on page 27), and the shaping device ShapCtrl (Section 7.7.3 on page 26). TCP/IP will be next candidate.

10.0 Examples

10.1 Complete Simulation Series with Confidence Intervals

In the following, an example producing a curve of the cell loss probability in a multiplexer over the buffer length is given. Confidence intervals for the measurements are calculated, and the produced output file can be visualized directly by gnuplot.

The parameters of a simulation model can be changed by deriving values from environment variables. These can be set by appropriate shell scripts. The sript below sets the variable BLEN to 10, 20, ..., 100 and calls YATS for each value.

```
#!/bin/sh
BLEN=10
while [ $BLEN -le 100 ]
do
    export BLEN
    yats y.in 1>>y.out 2>>y.err
    BLEN=`expr $BLEN + 10`
done
# end of shell script
```

BLEN is recognized in the following simulator input file “y.in”. It performs a warm-up and 10 runs for each value. A confidence interval (95 %) is computed, and results are written to standard output which has been redirected by the shell script.

```
#include “MACROS”        // load macro library
var i, nruns, nsrc, blen, slots;

blen = int(env(“BLEN”)); // read environment variable for
                         // buffer length, convert to int
slots = int(5e6);        // length of one simulation run
nruns = 10;              // perform 10 simulation runs
for i = 1 to nruns
    var ploss[i];        // an array to store results of runs

/*******************************************************************/
/*
*   model description
*/
nsrc = 60;               // number of sources
for i = 1 to nsrc        // nsrc Ethernet-like sources
    BSquelle src[i]: EX=32, ES=2720, DELTA=15, VCI=i, OUT=mux->I[i];
Multiplexer mux: NINP=nsrc, BUFF=blen, OUT=sink;
Senke sink;

/*******************************************************************/
/*
*   warm-up, nruns simulation runs
*/
Sim->Run SLOTS=slots;  // one run for warm-up

for i = 1 to nruns
{   var k, sent;
    mux->ResLoss;          // reset loss counters in mux
    for k = 1 to nsrc
        src[k]->ResCount; // reset departure counters

    Sim->Run SLOTS=slots; // simulate the model

    sent = 0.0;           // force sent to be a double value
    for k = 1 to nsrc     // add departure counters
        sent = sent + src[k]->Count;
    ploss[i] = mux->Losses(1, nsrc) / sent; // store the result
}
```

```
/*******************************************************************/
/*
*   calculate confidence interval, output result
*
*   The macro Confid95() is defined in the file “MACROS”.
*   It expects the reference to the array containing values, its
*   length, and the references to the variables where to store results
*/
var mean, lo, up;
Confid95(ref(ploss: [1]), nruns, ref(mean), ref(lo), ref(up));
print blen, “\t”, mean, “\t”, lo, “\t”, up, “\n”;

// end of YATS input file
```

After having run the shell script, gnuplot can vizualize the curve. The following commands have to be given:

```
$ gnuplot
gnuplot> set log y
gnuplot> plot “y.out” with errorbars
```

A simpler possibility to calculate confidence intervalls is described in Section 5.1 on page 12. For the macro version used here, see Section 11.5.1 on page 51.

10.2 ABR Multiplexer, Graphical Online Displays

The following input file defines an ABR multipxer loaded by 200 ABR sources and an optional background source. Between ABR sources and multiplexer, different delay lines are placed both in forward and backward direction. The data sources always want to send with the full rate, they are stopped by the ABR sources via the start-stop protocol. To turn on the background traffic, uncomment the backround source below.

The graphical displays have to be placed on the screen with the first run (initially, most of them overlap). The new positions are stored in the file “abr.pos”, see Control object below.

```
// ABR example for YATS

var   i, nsrc;
nsrc = 200;       // # of ABR sources

/*******************************************************************/
/*
*   model description
*/
for i = 1 to nsrc
{   // data source
    GmdpStop gmdp[i]: NSTAT=2, DELTA=(1,1), EX=(10,50),
             TRANS=(0,1,1,0), VCI=i, OUT=abrsrc[i];
    // ABR source
    AbrSrc abrsrc[i]: BUFF=5, BSTART=2, MCR=0, PCR=100000, FRTT=100,
           ROUTE=(2, abrmux1, abrsink[i]), AUTOCONN,
          OUTCTRL=gmdp[i]->Start, OUTDATA=linefw[i];

    // delay forward
    Leitung linefw[i]: DELAY=i*10, OUT=abrmux1->I[i];
    // delay backward
    Leitung linebw[i]: DELAY=i*10, OUT=abrsrc[i]->BRMC;

    // ABR sink
    AbrSink abrsink[i]: BUFF=1000, OUTBRMC=snkmux->I[i],
                        OUTDATA=datasink[i];
    Senke datasink[i];
}

/*
*   background source and sink:
*   uncomment next two lines to turn on background traffic
*/
// MMBPquelle bgsrc: EB=3000, ES=3000, ED=1.3, VCI=nsrc+1,
//                   OUT=abrmux1->I[nsrc+1];
Senke abrsink[nsrc + 1]; // background sink

/*
*   forward direction
*/
AbrMux abrmux1: NINP=nsrc+1, MAXVCI=nsrc+1, BUFFCBR=100,
                BUFFABR=100000,
//              HI_THRESH=5000, LO_THRESH = 4000,
                TBE=1000, AI=100, TARGUTIL=0.95,
                OUTBRMC=srcdmx1, OUTDATA=msload1;
Meas2 msload1: MAXCTD=100, MAXIAT=100, OUT=demux1;
Demultiplexer demux1: MAXVCI=nsrc+1, NOUT=nsrc+1,
                      OUT=(i: abrsink[i]);
for i = 1 to nsrc+1
    Signal demux1 (i,i,i);

/*
*   backward direction
*/
Multiplexer snkmux: NINP=nsrc, BUFF=100, OUT=abrmux1->BRMC;
Demultiplexer srcdmx1: MAXVCI=nsrc+1, NOUT=nsrc, OUT=(i: linebw[i]);
for i = 1 to nsrc
    Signal srcdmx1 (i, i, i);

/*************************************************************/
/*
*   Graphics: place the windows with first call, positions are stored
*   in the file “abr.pos”
*/
Control: POS=(50,50), DELTA=10000, FILE=”abr.pos”;

Meter mq: VAL=abrmux1->QLenABR, TITLE=”Queue Len in muxabr1”,
        WIN=(200,200,400,200), MODE=AbsMode, NVALS=1000,
        MAXVAL=10000, DELTA=100, UPDATE=100;
Meter mqz: VAL=abrmux1->Z, TITLE=”Z in muxabr1”,
        WIN=(200,200,400,200), MODE=AbsMode, NVALS=1000,
        MAXVAL=5, DELTA=100, UPDATE=100;
Meter mqnabr: VAL=abrmux1->CRNABR, TITLE=”CRNABR in muxabr1”,
        WIN=(200,200,400,200), MODE=AbsMode, NVALS=1000,
        MAXVAL=4e5, DELTA=100, UPDATE=100;
Meter mqabr: VAL=abrmux1->CRABR, TITLE=”CRABR in muxabr1”,
        WIN=(200,200,400,200), MODE=AbsMode, NVALS=1000,
        MAXVAL=4e5, DELTA=100, UPDATE=100;
Meter m1: VAL=abrsrc[1]->ACR, TITLE=”ACR abrsrc[1]”,
        WIN=(700,200,400,200), MODE=AbsMode, NVALS=1000,
        MAXVAL=10000, DELTA=100, UPDATE=100;
Meter m1ir: VAL=abrsrc[1]->CountData, TITLE=”data abrsrc[1]”,
        WIN=(700,200,400,200), MODE=DiffMode, NVALS=100,
        MAXVAL=100, DELTA=1000, UPDATE=10;
Meter m1irm: VAL=abrsrc[1]->CountRMCI, TITLE=”RMCI abrsrc[1]”,
        WIN=(700,200,400,200), MODE=DiffMode, NVALS=100,
        MAXVAL=100, DELTA=1000, UPDATE=10;
Meter m1or: VAL=abrsrc[1]->CountRMCO, TITLE=”RMCO abrsrc[1]”,
        WIN=(700,200,400,200), MODE=DiffMode, NVALS=100,
        MAXVAL=100, DELTA=1000, UPDATE=10;
Meter m2: VAL=abrsrc[nsrc]->ACR,
        TITLE=”ACR abrsrc[“ + string(nsrc) + “]”,
        WIN=(700,500,400,200), MODE=AbsMode, NVALS=1000,
        MAXVAL=10000, DELTA=100, UPDATE=100;

/*******************************************************************/
/*
*   Simulation run of 1 million time slots
*/
Sim->Run SLOTS=int(1e6);

//   ouptut some statistics
print “load in abrmux1: “, msload1->Count / 1e6, “\n”;
var  l; l = 0;
for i = 0 to nsrc + 1
    l = l + abrmux1->LossData(i);
print “loss in abrmux1: “, l, “\n”;

print “departures from abrsrc[1]: “, abrsrc[1]->CountData, “\n”;
print “arrivals in datasink[1]: “, datasink[1]->Count, “\n”;
print “data arrivals in abrsink[1]: “, abrsink[1]->CountData, “\n”;
print “RMCI arrivals in abrsink[1]: “, abrsink[1]->CountRMCI, “\n”;
print “RMCO arrivals in abrsink[1]: “, abrsink[1]->CountRMCO, “\n”;
print “arrivals in datasink[“, nsrc/2, “]: “,
        datasink[nsrc/2]->Count, “\n”;
print “arrivals in datasink[“, nsrc, “]: “,
        datasink[nsrc]->Count, “\n”;

// end of YATS input file
```

11.0 Some Further Notes

11.1 ... Regarding References

1. For variables and macros, a reference is bound to the “home” block of the entity (unique ID for each block instance, e.g. in a loop). The reference therefore becomes invalid in case the block has been already left when resolving the reference which results in an error message. Variables with the same identifier, but used in different blocks, can be differentiated due to the block information contained in a reference generated by ref().
2. Although references eventually are normal strings, they only should be assigned to variables, but never be processed. To avoid unintended mistakes, references twice contain the character ‘\001’ which, of course, is not a protection against abuse.

3. The null reference is “”, e.g.: `if (r != “”) deref(r) = 1;`
4. When recognizing the following rule, then references should be really save. Avoid to declare – in the same block – arrays which only differ in the index ranges. The next example probably will not behave as intended. Instead of causing an error message, `a[10]` will be accessed by the macro. We would get an error message, however, if the second array declaration and the macro call would be placed in a new block.

   ```
   macro mac(x) { deref(x)[10] = 1; }
   var i,r;
   for i = 1 to 5
       var a[i];    // creates a[1]...a[5]
   r = ref(a: [1]); // intention: reference to a[1]...a[5]
   for i = 10 to 15
       var a[i];    // !! this extends the scope of r !!
   mac(r);          // will cause: a[10] = 1
   ```

11.2 ... Regarding Literals

As has been shown in the example of Section 4.1 on page 8, the string argument of `lit()` may contain input name extensions built upon ‘->’. This is very useful but somewhat dirty, therefore a note here.
The possible inclusion of input name extensions is based on two facts. Firstly, lit() does only check that its argument does not contain white space or the special character ‘\001’ which is essential for references. Thus, in principle all fency identifiers can be generated. Secondly, the routine used by network objects when evaluating names of their succeeding objects, is based on the same piece of code also implementing the lax lit() transformation (kernel/id.c::parse_id()).
An important consequence of this ‘by-chance’ possibility is that it unfortunately does *not* work for commands and exported variables. There, the whole thing has to be splitted into two parts, e.g.:

```
Senke sink;
// does NOT work (syntax error):
print lit(“sink->Count”), “\n”;
// works fine:
print lit(“sink”)->lit(“Count”), “\n”;
```

11.3 ... Regarding Macro Shells

11.3.1 Commands to Macro Shell Objects

Together with the instantiation of a macro shell, an object can be created, and commands can be directed later on to the interface object. These commands are mapped into macro calls. This is only possible, if

1. the macro shell has at least one argument, and
2. the first argument is of type ‘lit’ (identifier). The actual value of this argument is taken as name of the generated interface object.

It is then possible, to include a list of command specifications in the macro shell definitions. The specifications are separated by commas, each element beginning with keyword CMD or CMDV. CMD defines a command wich cannot be used in expressions. The corresponding macro does not need to return a value. With CMDV, commands with return value are defined, and the corresponding macro has to return a value (syntax error message, otherwise). The syntax of a command specification is as follows.

```
{ ‘CMD’ | ‘CMDV’ } ‘=’ ‘(‘ cmdName { ‘(‘ cmdArgs ‘)’ } ‘,’
            ‘MACRO’ ‘=’ macroName ‘)’
```

If the command (name `cmdName`) has no parameter, then the part `cmdArgs` (with brackets) has to be skipped. Parameters are specified using the type keywords ‘int’, ‘double’, or ‘string’. Keywords and default values as for the macro shell itself, are *not* supported. The parameter specifications are separated by commas. The macro (`name macroName`) has to expect (*nShell* + *nCmd*) arguments, where *nShell* is the number of arguments of the shell, and *nCmd* is the number of command parameters. The first *nShell* values passed to the macro are the values used for the instantiation of the macro shell. They are followed by the *nCmd* values used in the command statement or expression. Example:

```
macro defX(id)    // definition macro
{    Senke lit(id).s;// do NOT use lit(id): this is used
                     // as name of the interface object
}
macro getX(id, n) // command ‘Get(int)’
{    if (n != 1)  // only to demo sth ...
     {   print[2] id, “: an error.\n”;
         exit 1;
     }
     getX = lit(id).s->Count;
}
macro resX(id)    // command ‘Reset’
{    lit(id).s->ResCount;
}
MacroShell X: ARGS=(lit@1), MACRO=defX(string(@1)),
     CMD=(Reset, MACRO=resX),
             // no parameters, no return value
     CMDV=(Get(int), MACRO=getX),
             // integer parameter, returns a value
     print;

X bla;
bla->Reset;
print “bla->Get(1) = “, bla->Get(1), “\n”;
bla->Get(2); // we should see our error message
```

11.3.2 NULL Value for Identifier Arguments

For the definition of sub-models with optional outputs (like for a measurement device) the following might be useful. Default arguments of type ‘lit’ (identifier) can have the default vale ‘0’. This leads to an empty identifier. The normally applied cast to string then results in the empty string ““ which in turn can be tested by the macro. Example:

```
macro hh(id, oo)
{    // ...
     if (oo == ““)
         Meas2 lit(id): MAXCTD=10, MAXIAT=10;
     else
         Meas2 lit(id): MAXCTD=10, MAXIAT=10, OUT=lit(oo);
}
MacroShell HH: ARGS=(lit@1: OUT=lit@2(default 0)),
     MACRO=hh(string(@1), string(@2)), print;
HH h1: OUT=snk;
Senke snk;
HH h2;   // no succeeding object
```

11.4 ... Regarding Global Variables

The following restrictions apply:

- When a global variable is defined, then it is checked that currently no other variable, macro or object with the same name is defined in any surrounding block.
- Once a global has been defined, the name never can be used again. This also holds for “normal” variables which cannot “overlaod” global variables as they can do with “normal”, local ones.

Globals therefore only should be used if really necessary.

11.5 ... Regarding Confidence Intervals

The library “MACROS” (directory yats/examples) comprises the following macros for statistical evaluation and simulation control. The newer version based on an object class is described in Section 5.1 on page 12, the old, macro-based versions are given here.

11.5.1 Basic Macros

`MeanVar`(), `Confid`(), and `Confid95`() require that the user defines the necessary arrays. More convenient macros are described in Section 11.5.2 on page 52.

MeanVar(refVals, nval, refMean, refVar)
Calculates empirical mean and variance of an array of values. Parameters are:

`refVals`: reference to the array of values

`nval`: number of values. The expected array indices are [1] to [nval].

`refMean`: where to write the empirical mean (reference) `refVar`: where to write the empirical variance (reference)

Confid(level, refVals, nval, refMean, refWidth)
Calculates the confidence interval from an array of values and to a specified level of confidence (using quantiles of Student’s t-distribution). The parameters are

`level`: level of confidence (possible values are 0.9, 0.95, 0.975, and 0.99)

`refVals`: reference to the array of values

`nval`: number of values. The expected array indices are [1] to [nval].

`refMean`: where to write estimate of mean value (reference)

`refWidth`: where to write interval width (reference). The interval extends from `(mean-width)` to `(mean+width)`.

Confid95(refVals, nval, refMean, refLo, refUp)
Resembles `Confid()`, but the level of confidence is fixed to 95 per cent. Instead of the interval width, its lower and upper bounds are returned (references `refLo` and `refUp`).

11.5.2 More Convenient Evaluation of Confidence Intervals

For the following macro class, it is not necessary to define auxiliary arrays which sometimes make the input file rather messy.

ConfObj(name, level, maxNvals);
This defines an auxiliary array object (see Section 7.12.4 on page 41) with the given `name` (string). The level of confidence is passed in `level` (double), `maxNvals` (integer) specifies the maximum number of observations which can be stored in the array.
Example: `ConfObj(“ccc”, 0.95, 20); // object ccc defined`
ConfAdd(refObj, value);
This adds `value` (double) to the object referenced in `refObj`.
Example: `ConfAdd(ref(ccc), x); // x is a measured value`
ConfMean(refObj);
`ConfMean`() returns the mean value (as double) of the values collected so far.
Example: `mean = ConfMean(ref(ccc));`
ConfLo(refObj);
Returns the lower bound of the confidence interval.
ConfUp(refObj);
Returns the upper bound of the confidence interval.
ConfVar(refObj);
Returns the emperical variance of the values collected so far.
ConfBase(refObj, refMean, refWidth);
Calculates mean and half interval width as `Confid`().

Example
```
#include “MACROS”
var     i, nruns;
nruns = 9;
// define the object abc
ConfObj(“abc”, 0.95, nruns);
// do the measurements
```

```
for i = 1 to nruns
     ConfAdd(ref(abc), i); // i is our “measurement”
// print results
print ConfLo(ref(abc)), “\t”, ConfMean(ref(abc)),
         “\t”, ConfUp(ref(abc)), “\n”;
```