Department of Computer Science

University of Waikato

Hamilton, NewZealand

The Naïve Programmer's Guide to ATM_TN

Philip J. Tree

July 1999

1 Introduction

One of the biggest problems with ATM_TN is the lack of a brief, yet reasonably comprehensive, overall description of the program from the programmer's viewpoint. The lack of this feature means that when the programmer first confronts ATM_TN, with the aim of making modifications to the code, it is very difficult to know where to begin. I hope this brief overview will be of some use to programmers in this situation.

Note: Where the program structure has been significantly modified by the addition of IP and ATM over IP functionality, this has been added as a footnote.

2 File and library structure

ATM_TN is a fairly large program. If you have never encountered large programs before, its sheer size will be overwhelming. Don't try printing the entire source code, for example, as it runs to some 4000 pages!

At the top level these directories are seen: *README, atmtn_s, include, simfunc, simtest, doc, lib, simkit_s* (the names vary slightly between versions of the code, but the idea remains the same).

All ATM_TN is inside the *atmtn_s* directory. The rest is part of SimKit. Within *atmtn_s* there are a number of directories:

Three of these directories *common*, *frame* and *switch* are code libraries, and these have the same sub-directory structure. Each code library directory consists of two sub-directories *build* and *src*. The *build* sub-directory is further subdivided into directories for each architecture to which ATM_TN has been ported (*sgi, sun4* ...). Source code is **always** in the *src* sub-directory.

The library in which a source file resides can **usually** be determined from the prefix in its name:

- Files with the prefixes *nm* ... are located in the *switch* library.
- Files with the prefixes *afr* ... are located in the *frame* library.
- Files with the prefixes *atm* ... are located in the *common* library.

- Other file prefixes usually mean the file will be located in a code library attached to a traffic model.

The most obvious exceptions to this rule are the files *atm–frame.cc* and *atm_message.cc* which are in the *frame* library, and the file *atm–tn–model.cc* which is in the directory *model/src*. Also, note that *.h* files use hyphens as separators in filenames, while *.cc* files usually (but not always) use underscores.

Opening the *traffic* directory reveals an even larger set of directories, most of which are code libraries for the traffic models. Other directories named within *traffic* have the same function as their similarly named counterparts in the *atmtn_s* directory.

The *model* directory has a similar structure to the code libraries, but in addition contains a third sub-directory, *tests*, in which the user data files are placed. The README file in this sub-directory has important information.

The *bin* directory contains architecture sub-directories, within which symlinks to the actual executable will be found. (The actual executable is in *model/build/"arch"*).

The *include* directory contains symlinks to *.h* files

The *lib* directory contains architecture sub-directories, into which Makefile places the *.a* executable libraries.

3 SimKit

ATM_TN uses the Discrete Event Simulator provided by the SimKit API to perform the simulation. The description supplied here will be brief, as the SimKit API is well documented in the SimKit header file *simkit.h*.

The SimKit API consists of three classes *sk_simulation*, *sk_lp* and *sk_event*. A single object of type *sk_simulation* must be created at the start of the simulation, and classes *sk_lp* and *sk_event* are used as the base classes from which simulation process classes and simulation event classes respectively are derived.

SimKit must be installed and compiled before ATM_TN can be compiled. The exact way to compile SimKit may depend on the version used and is explained in the top-level README file.

It will probably involve first modifying the script in file *include/Makefile.common* to supply the pathname for your implementation, and then compiling first *simkit* and then *simfunc*. The Makefile script in the *build/"arch"* subdirectory within each of the directories simkit and simfunc is the top-level script, and is compiled with the three commands:

```
make export
make depend
make build
```

4 Simkit execution phases

This is a summary of the six phases in a SimKit simulation: [1]

Phase I – Program Initialization

> This phase occurs before the line: *afr_kern.initialize(argc, argv);* in *main* is executed. The *sk_simulation* object is constructed in this phase. All of this phase is run sequentially on the host processor.

Phase II – SimKit and Model Global Initialization

> This phase starts with the line: *afr_kern.initialize(argc, argv);* in *main*. The LPs, derived from *sk_lp*, are instantiated. The CSS area of each LP's Global application data structures are built during this phase.

Phase III – Logical Process Initialization

> This phase starts with the line: *afr_kern.start_simulation();* in *main*. Each LP's *initialize()* function is called exactly once. This processing is run concurrently on all processors. Simulation time does not advance during this phase and no events are received. No LPs may be created during this phase, but new events may be created.
>
> This phase is used to send out seed events to start the simulation. These seed events are not received by their destination LPs until all LPs have been initialized and phase IV has begun.

Phase IV – Simulation Execution

> Once all LPs have been initialized, the SimKit kernel begins dispatching events to their destination LPs. When an event arrives at an LP, the LP member function *process(...)* is called. This phase ends when one of these conditions is met:

[1] A more extensive description can be found in the SimKit header file *simkit.h*

1. there are no more events to process (normal termination).
2. the simulation end time has been reached (normal termination).
3. a user reported error occurs (abnormal termination).
4. a SimKit error occurs (abnormal termination).

Phase V – Logical Process Termination

When phase IV ends normally, this phase starts. Each LP's *terminate* function is called. This processing is run concurrently on all processors. Simulation time does not advance during this phase and no events are received. The programmer is restricted from sending events and creating LPs. This phase is used to perform LP specific termination functions, such as reporting LP specific statistics.

Phase VI – Simulation Clean-up

When phase V ends normally, the function *afr_kern.start_simulation()* returns. Execution is once again sequential on the host processor. This phase is often used to tally statistics and output final reports.

5 Compilation and Makefile structure

ATM_TN uses a multi-level Makefile structure that will appear daunting to anyone who did not achieve an 'A' in Makefile 101. Furthermore, as the way Makefile scripts are interpreted varies between the different Unix's, it is entirely possible that the programmer may need to make modifications to some of the Makefiles to get ATM_TN to compile.

The top-level ATM_TN README file (*atmtn_s/*README) has most of the instructions required to compile ATM_TN. Note that SimKit **must** be compiled first. If you have never encountered *make depend* previously, read the *makedepend man* page on your system very thoroughly.

There are Makefile scripts throughout the directories. Most of them are extremely boring reading.

The Makefile script in *build/"arch"* is the top-level script. It is from this directory that ATM_TN is compiled, using the three commands:

```
make export_all
make depend_all
make build_all
```

There is a *Makefile* script in every *"library"/build/"arch"* directory. These are the Makefiles that *makedepend* writes for you. Everything below the line: *# DO NOT DELETE THIS LINE – make depend depends on it.* is written by *makedepend*. Don't touch!

There is a *Makefile.common* script in every *"library"/src* directory. These are Makefile scripts for system independent modules, and have a common framework plus a few rules special to the particular directory. It is stated there should be no need to modify these scripts when porting between architectures.

There is a *Makefile.common* script in the *src* and *traffic/src* directory. These are the Makefile scripts that export *.h* and *.a* files to the directory above them. As with other Makefile.common scripts, it is stated there should be no need to modify these scripts when porting between architectures. Not quite true! I had to make a subtle change to *traffic/src/Makefile.common* to port from *sun4* to *linux* [2].

The *Makefile.common* script in the *include* directory is where you enter the pathname for the implementation you are about to compile. You will need to modify this script.

The *Makefile."arch"* script in the *include/"arch"* directory has script specific to the architecture on which you are compiling ATM_TN. This is the script you modify when porting to other Unix's.

6 Using print statements within *ATM_TN*

The usual *printf* and *cout* statements do not work well (if at all) within ATM_TN. The print functions ATM_TN provides for the programmer are well documented in file *afr-utils.h*, and I strongly recommend you use only these functions.

I also recommend that you take careful note of the obfuscatory feature caused by the macro AFR_IMPL_ERROR being a compound statement. The statement: *if (xyz)* AFR_IMPL_ERROR*(...);* will work, but almost certainly not as you intended. Instead, use the statement: *if (xyz)* {AFR_IMPL_ERROR*(...);*} which should perform as intended.

[2] In *traffic/src/Makefile.common*, in the lines following *SUBTREES:* the name SUBSYSTEMS must be changed to another name eg. TSUBSYSTEMS. This is because the scoping rules within the *make* scripting language are different.

7 The *main* function

ATM_TN has a layer of obfuscation to designed to confuse, so don't expect to find *main* in file *nm_main.cc*. It isn't there. It is located in file *atm–frame.cc* (*frame* library).

The *main* function can be followed sequentially (with a debugger, for example) until the line: *afr_kern.start_simulation();* is reached. Beyond this point a debugger is of limited use (except for getting stack traces of Segmentation Faults and Bus Errors), as the logical order for the program is under the control of the Discrete Event Simulation kernel within SimKit.

Once the simulation commences it proceeds until it either faults, runs out of events to process, or reaches the simulation end time.

8 Reading (lexing) the input data type files

The code that does this is found in the *frame* library, with file *afr–interface.h* the key file.

Data files are read, parsed and lexed in this order, first the *.sim* file, then the type files (."xx"*t*). Instance data files (."xx"*i*) are not parsed or lexed at this stage.

Unfortunately there are two methods of parsing and lexing data:

1. Uses an instance of class *afr_gpars_t* (file *afr_gpars_t.cc*) to parse the data, and an instance of class *afr_lexer* (file *afr_lexer.cc*) to lex it.

   This method creates a variable *lex* to refer to an instance of the class *afr_lexer*.

2. Creates an instance of class *afr_lex_record* (file *afr_lex_record.cc*), then use the functions provided by this class to both parse and lex the data.

   This method uses a function *lex(...)*, a member function of class *afr_lex_record*.

Both methods ultimately place the data in record tables of type *afr_lex_record*.

The easiest way to discover which method applies to a particular piece of code is to observe the context in which the term *lex* is used.

9 Creating the component construction tables

This is where the interesting stuff starts. The line: *init_comp_cons_tbl();* in *main* calls this function to create an **array** of instances of class *afr_create_info* by calling the *init(...)* function (file *afr_create_info.cc*) for each member of the array.

Each instance of *afr_create_info* (file *afr–interface.h*) has two data members. The first member is a pointer to a string with the name of the component, the second member is a function pointer to the creation function for that component.

Note that for these functions to perform correctly it is important to update the code in file *atm–tn–model.cc* when any new components are defined. These must be correctly entered in the same place in **both** arrays in this file, and the constant CONSTBL_ENTRIES must also be incremented.

10 Parsing the instance files

The line: *if (create_instances()) ...* in *main* calls this function to parse the instance data files. It, in turn, calls the function *parse_instance_file(...)* once for each instance file.

As this function lexes the file (the name *parse_instance_file* is a bit of a misnomer), line by line, into the appropriate *afr_lex_record*, a model component of the specified type is constructed.

The first instance file lexed is the Link-Port instance file (*.lpi*). Function *init_ports()* then connects individual links to the ports at each end of the link. This occurs before other instance data files (*."xx"i*) are lexed.

11 Creating the model components

The process by which the model components (network nodes, links, traffic endpoints, etc.) are created is sufficiently complex to be disconcerting to the uninitiated. Even after following the code in action with a debugger, there remains an aura of mystery that it actually works at all. There is something of a magician's touch here.

The four stage instantiation process begins at the line: *cfcn = find_creator(...)*

in function *parse_instance_file(...)*, located in file *atm–frame.cc*.

1. The function *find_creator(...)* (file *atm–frame.cc*) uses the name of the model to search the component construction table and returns a *pointer* to the actual creation function and assigns it to the variable *cfcn*. The pointer has type *afr_create_comp*, which is an alias (*typedef*) for *afr_base*. All component types derive from *afr_base*, so this works.

   The line: *comp = cfcn();* then calls the specific creation function for the component type. Creation functions for network nodes are located in file *nm_create.cc*, and for other components, in files *..._create_ ...* in the relevant library.

   The creation function calls the class constructor. The class constructor takes no arguments, therefore the object is created with default values in all its parameters (or garbage if no default values are specified). The constructor code in turn calls class constructors for each LPs this node will require.

2. The *init(...)* function is called by the next line: *if (comp–>init(type, &inst))*. This function takes two arguments, both pointers. One points to the record that contains the data lexed from the type file *.swt*, the other to the record that contains the data lexed from the instance file *.swi*. This function reads data from both these files and uses this data to set the component parameters to their correct initial values.

3. The third stage occurs in network components and attaches network nodes to other network nodes via the link components. This stage begins with the line: *if (errflag == 0 && make_connections()) ...* in *main* and calls the *connect(...)* function in each component.

4. Final initialization, if required, takes place after the network topology has been created, and occurs when the *"classname"_init()* function is called. With network nodes, this function requires the Relative station ID of the instance as an argument. Not all components have this final initialization phase.

12 Creating the network topology

Functions that are called during the network topology [3] construction stage are located in files with names beginning *nm_swinit_ ....*

[3] Two separate topologies are constructed:

1. The "REAL" topology consisting of ATM end-nodes, IP end-nodes, converters, ATM switches and IP routers.

The construction of the network topology commences with the line: *if (errflag == 0 && afr_nm_init()) ...* in *main*. If the file *atm.dat* exists, *afr_nm_init()* uses the information in this file to set the value of a number of global constants, otherwise it sets these constants to default values. It then calls function *makeNetwork()*, and this function in turn calls a sequence of other functions to actually do the hard work.

The last function to be called by *makeNetwork()* is *storeTables()*. During the construction of the network topology, a number of global tables are created to hold topological information. However, before simulation commences, the relevant sections of these tables are distributed to local tables in the components. The global table memory spaces are then deallocated.

Network topology functions frequently refer to three *integer* values that must not be confused by the programmer. They are:

Network ID:

This is an array index to the location of the node information in *afr_swi_tbl*. It is assigned when the node data is lexed into this table from the *.swi* data file, and so it is determined solely by the order in which the data was entered into that file by the user.

Station ID:

This is a unique integer from 0 to $n-1$ that is assigned to each node, where n is the total number of nodes in the network topology. Station ID is assigned in strict order first to end-nodes then to switches [4].

Relative station ID: (Unfortunately also called **Station ID** in some functions!)

This integer is unique only within the scope of the type of node.

For example: In a simulation with n end-nodes and m switches, the end-nodes will be assigned relative Id's 0 to $n-1$; the switches will be assigned relative Id's 0 to $m-1$.

As with most of ATM_TN, a layer of obfuscation has been built in to the network topology code. The function *networkIdToStationId(int id)* converts **from** station Id's **to** network Id's, and the function *stationIdToNetworkId(int stnId)* converts **from** network Id's **to** station Id's.

2. The "VIRTUAL" topology consisting of ATM end-nodes, ATM switches and IP network instances, where each IP network instance is an "island" of IP separated from other IP "islands" by ATM.

[4]The strict order by which Station Id's are assigned is: ATM end-nodes, IP end-nodes, converters, IP routers, ATM switches, IP network instances.

Now, re-read the previous sentence. No, this is not a typing error on my part; these two functions really do perform this way round!! Check the code for yourself if you don't believe me!

13 Types of components

All components derive from the base class *afr_base* located in file *afr–interface.h*. This class supplies a number of functions common to all components. Components may be either passive or active.

Passive components have no LPs associated with them. Passive components include:

- links (class *afr_link*) in file *afr–interface.h*. [5].
- routing table entries (class *afr_route_entry*) in file *afr–route.h*).
- permanent virtual connections (classes derived from *afr_vc_base* in file *afr–vcs.h*).

Every component that is active must own at least one LP (the *main* LP, file *nm_main.cc*). All active components derive from the derived base class *afr_comm_base* which provides additional functionality for components that communicate with other components. Active components include:

- traffic source/sink classes found in the respective traffic models.
- end-node and switch classes [6] (file *nm–simple.h*), all derived from class *nm_node* (file *nm–node.h*).

14 Tables

Tables fall into two categories:

1. Tables with data that may be modified while the SimKit simulation phases are in progress. These must be owned by an LP, with access to data in the table restricted to that LP.

[5] *afr_link* becomes a base class from which *afr_link_IP* and *afr_link_ATM* derive
[6] Also converter, IP end-node and router classes

2. Tables with data that is entered during the setup phases and then used in read-only mode. These are usually owned by a component. Access can safely be global if this is required.

Some tables use templates that are supplied in file *nm–templates.h*. Note there are two *.cc* template files *nm–templates.cc* and *nm_templates.cc*. Spot the difference!

15 Defining and using the LPs

All LP classes derive, directly or indirectly from the SimKit API class *sk_lp*.

Some LP classes, for example *nm_input* and *nm_output*, are created in their entirety by their class constructor. Other classes, for example *nm_switch_network_layer*, go through a two-stage process; the second stage being a *"classname"_init()* function called, after the construction of the network topology, by the *"classname"_init()* function of the component to which the class belongs.

The *initialize()* function takes no arguments, and is called once for each LP during phase III. This function is used to complete initialization, if needed, and to send initial messages, again only if needed. In some LPs, for example, the *main* LP in each component, this function does nothing.

The *process(sk_event *)* function is the key function of each LP and must be defined. It is the function called to process an event during phase IV. It is supplied with the event as an argument, processes that event, and usually ends by creating a new event and using the *send_and_delete(...)* member function of the new event to dispatch the event to its destination.

Note that memory allocated to LPs is assigned with the *void *alloc(size_t size)* function and deallocated with the *void dealloc(void *ptr)* function. However, to make code easier to read, most (but not all) classes that derive from *sk_lp* overload their *new* and *delete* operators to use these functions (file *nm_op_new.cc*). It is a pity this overloading is not done within the SimKit API.

Perhaps the most important aspect of LP classes is that memory space that contains data that can be modified during phase III or IV of the simulation **must be owned by an LP**. This can involve instances of arrays, table classes, and so on. Only the LP that owns this memory may write to it or read from it. Other LPs can access the memory only by sending an event to the LP that owns it. This is key to using Discrete Event Simulation on

multiple processors.

This means that design decisions made regarding the ownership of table class instances are very important.

16 Defining and using the events

All event classes derive from the SimKit API class *sk_event* via the class *atm_event_base*. This class uses a *private integer* data-type to represent the type of the event.

The key function in the event classes is *send_and_delete(sk_lp *dest, sk_time rtime)*. Not only does this function send the newly created event to LP *dest* to arrive at *rtime*, it also handles the deallocation of the memory assigned to the event by the *new* operator, so that the programmer never has to manually delete an *sk_event* object. Note that the *new* operator in the *sk_event* classes has been overloaded.

Before you start using the *sk_event* class do ensure you read the relevant comments in the *simkit.h* file.

So far so good. Now things get messy. Two types of classes derive from *atm_event_base*:

1. The class *nm_event* (file *nm–event.h*) is a general event class, and the class constructor requires an *integer* argument to represent the type of event carried by the instance. The object carried by the event exists as a *public* variable in the class—no *get* and *set* functions are provided—and it must be placed in the event by the programmer.

2. All other event classes (file *atm–events.h*) are designed so that there is a separate class for each type. No *integer* type argument is supplied with the class constructor, as creating an instance of the derived class automatically defines its type. However, constructors for these event classes do require either an object of the required type to be passed as a argument, or, failing that, sufficient information that the constructor can itself construct a valid object of the required type. The object carried by the event then exists as a *private* variable.

Now it may seem, to the programmer, that the difference between these classes is trivial; but, in fact, it creates all kinds of problems, not the least

being that it is easy to forget which type of event you are sending, and even easier to send an empty event (and then wonder why it does nothing).

In particular, if you are using class *nm_event* for your event class, the compiler will not check whether you have the correct event—any *integer* is as good as any other to a compiler. Indeed, with *nm_event*, the compiler cannot even tell you whether the event you created contains anything at all.

So you are warned. Avoid the *nm_event* class unless you are feeling particularly masochistic.

17 Objects carried by the events

Objects carried by the event classes are frequently constructed at one LP and destroyed at another, therefore they must use the LP memory allocation and deallocation functions *void *alloc(size_t size)* and *void dealloc(void *ptr)* (or overload the *new* and *delete* operators to use them). The kinds of objects with their location in ATM_TN code are:

- ATM data cells (class *atm_cell*) in file *atm–cell.h* [7].
- Resource Management cells (class *atm_rm_cell.h*) in file *atm–rm–cell.h*. These are used only if congestion control methods have been specified in the input data files.
- ATM signalling messages.

  There is a separate class for each message type. TSS–end-node messages are derived from base class *atm_msg*, in file *atm–message.h*. All other message classes are described in file *nm–control.h*.

Signalling messages carried on event instances constructed using the *nm_event* class are cast to type *void* before being assigned to the event, therefore it is essential that the type of the message is transported somehow. This type is carried in the global enumeration *NM_N_Pkt_Type* described in file *nm–network.h* [8].

Unfortunately, because the *integer* type field in *atm_event_base* uses a different number representation than the *NM_N_Pkt_Type* data-type, it means that a two step process has to be used to establish the type of an arriving

[7] This file is extended to include classes *IP_packet*, *ATMoverIP_packet* and *AAL5_conv_pkt*

[8] The *NM_N_Pkt_Type* enumeration is shifted to *atm–cell.h*

message. The *integer* type field in *atm_event_base* is used to determine the base type of the message. The message is then cast to the correct message base type and the *NM_N_Pkt_Type* field in the message base class used to cast the message to the correct final type. (OK, so this models the actual situation in ATM signalling quite well—but do we need to be that pedantic in our modelling!)

There are at least five types of signalling messages [9], deriving from at least three base classes. (I don't promise that you won't find more, tucked away in obscure corners of ATM_TN.) :

1. TSS–end-node messages.

   These are derived from base class *atm_msg*.

2. Messages between signalling adaptation layer peers.

   These are derived from class *nm_ss_pkt*, itself derived from base class *nm_packet_base*.

3. Messages between ATM switch network layer peers.

   These are derived from class *nm_nn_pkt*, also derived from base class *nm_packet_base*.

4. Messages between signalling adaptation and switch network layers.

   These are derived from class *nm_ns_msg*, itself derived from base class *nm_ipc_msg*.

5. Messages from a network layer to itself (timer messages).

   These are derived from class *nm_nn_msg*, also derived from base class *nm_ipc_msg*.

18 ATM signalling

It is important to realise that each ATM switch has both an adaptation and a network layer, and that, while data cells are switched at the ATM layer, the connection, disconnection and management of connections require the existence of higher layers at each switch.

In this respect, the arrival of a cell at the input ATM layer of a switch with VPI = 0 and VCI = 5 means "send me up your protocol stack". Signalling

[9] A sixth type is added for messages between IP processes: class *nm_IP_msg*

information PDUs may be longer than 48 bytes, therefore a signalling adaptation layer is also required.

The code that handles ATM signalling in ATM_TN is perhaps the most complex and convoluted of all. Rumour has it takes 3 months perusal to understand!

The two LP classes involved with ATM signalling are adaptation layer class *nm_sar_layer* and network layer class *nm_network_layer*. Both classes derive from *sk_lp* via class *nm_control_base*, and share a number of functions including the *process(...)* function. Member functions for class *nm_control_base* are located in file *nm_globobj.cc*, while *nm_sar_layer* member functions are located in files with names beginning *nm_s_ ...* and *nm_network_layer* member functions in files *nm_n_ ....*

The *process(...)* function decides whether the incoming event is destined for the adaptation or network layer and calls the appropriate *perform(...)* function, which does the bulk of the event processing. Note that unlike *process(...)* functions in other LP classes, this function does not conclude by creating a new event and dispatching it another LP.

Outgoing events are generated separately by a number of different *"send"* functions. Some of these are general member functions in *nm_control_base*, others are member functions specific to the layer and are located in files *nm_n_messages.cc* and *nm_s_misc.cc*.

19 Traffic models

The construction and implementation of the traffic models mirrors the construction of the lower layers in ATM_TN, but due to the smaller size of the code that builds each model, there is more chance of understanding what is occurring simply by reading the code.

It is important to realise the TSS (Traffic Source/Sink) component is logically entirely separate from the end-node component to which it is attached, and a nominal link component exists between the two.

This link carries representations of cells and signalling messages.

20 Final note – debugging

I can recommend the use of *electric fence* (available in the *Debian* linux distribution) for debugging. This program replaces calls to *malloc* and *free* with its own functions, and these seg-fault if memory accesses fall outside the allocated memory space. The seg-fault is then traced with an ordinary debugger.

Note that to use *electric fence* you will need to recompile ATM_TN using gcc instead of g++, and your computer will probably require 64M of RAM to compile as electric fence can be rather hungry.