A Simulator Function Library for the SWIFT LAN Manager Prototype

Armando J. Pinho, Fernando M. S. Ramos

Abstract - **This paper gives a description of the Simulator part of the SWIFT LAN Manager prototype, developed at INESC, as a contribution to the EURET/SWIFT Project. The editor tool used to manipulate simulator event files and the encoding used to represent actions are also described.**

Resumo - **Este artigo descreve parte do trabalho desenvolvido no INESC respeitante ao protótipo de um LAN Manager para o projecto EURET/SWIFT, projecto co-financiado pela Comissão da União Europeia. Em particular é descrito o núcleo do simulador assim como o editor dos ficheiros de simulação e o respectivo código usado para a representação de acções.**

I. INTRODUCTION

The aim of the EURET/SWIFT (Specifications for Controller Working Positions in Future Air Traffic Control) project is to provide a proposal of detailed specification of the Controller Working Position (CWP) to be used in the development of future Air Traffic Control systems.

One important aspect of the definition of CWPs is the communications environment, and in particular the definition of the LAN Manager architecture and facilities that should be implemented. For the purpose of clarifying the concepts of LAN Management in SWIFT it was specified and developed a prototype of LAN Manager.

The SWIFT LAN Manager prototype includes two basic functional parts: the Human Machine Interface (HMI) and the Simulator. The HMI provides all the interaction with the user, including the support of user inputs and representation of outputs, and the execution of the kernel machine of the prototype. The Simulator part provides the HMI, at its request, with time controlled simulation events previously recorded in a simulator event file. Also, an Editor is provided as the tool to manipulate the simulator event files. The Editor, although working dissociated (i.e., off-line) from the Simulator and HMI, is considered as belonging to the Simulator part.

This paper gives a brief description of the design and implementation of the Simulator part of the prototype (including the Editor) which was developed at INESC, Portugal. The HMI part, that will not be described in this paper, was developed by ESG, Germany, that is a private company partner of INESC in this SWIFT work package.

All the programming was done using the C language, under the UNIX operating system environment. Although the programs were intended to run on a HP 9000 machine, the code (at least the one used to implement the Simulator part) is easily portable to other architectures (in fact it was developed mostly under the LINUX version of UNIX, running on a 486 machine).

In the following sections we will give an overview of the most important aspects of the Simulator including the respective event file Editor, focusing on the implementation, file format and encoding of actions.

II. THE SIMULATOR KERNEL

The purpose of the simulator kernel that we describe here is the generation of events affecting a global data structure that encodes, in every moment, all the state of the current simulation. This data structure is also the connection between the simulator kernel and the HMI front-end. Briefly, the interation between the HMI front-end and the simulator kernel is as follows. Periodically (tipically every second), the front-end calls a function (the simulator kernel) that, if there is any action to be performed, it will be reflected in the global data structure. Therefore, the coupling between the HMI and the simulator kernel was kept quite low in order to avoid, as much as possible, the inter-dependency of these two modules. This was an important factor since the two modules were developed at distinct institutions, the HMI at ESG in Germany and the simulator kernel at INESC in Portugal.

The events can be of several types including structural and functional changes of the environment, and alarms. Nevertheless, they are treated all the same way by the simulator, i.e. as changes on the global data structure (these changes on the global data structure will be referred from now on as actions).

From the HMI point of view there is only a way to access the simulator kernel which is by calling the sim_periodic_timer() function. This function will determine if there is any action to be performed and will execute it if this is the case. The return value of this routine informs the HMI if the call succeeded, if the

simulation ended or if the simulator was not able to access the specified simulator event file.

Analyzing with more detail the simulator kernel we identify an internal timer that is responsible for the correct deliverance of the time programmed events. As we will show shortly, each event has a time tag that represents the time of occurence of that event during simulation. The data structure that encodes an event is simply:

```
typedef struct {
    int     nActions;
    int     offsetTime;
    int     absoluteTime;
    char    **actions;
  } SimEvent;
```

The parameters offsetTime and absoluteTime hold, respectively, the number of seconds that have to tic since the last event occured, and the number of seconds since the begin of the simulation (obviously this is redundant information but makes things easier to handle). The paramenters nActions and actions denote, respectively, the number of actions to be performed when that event occurs, and the actions encoded as strings (below we will explain the encoding used to represent actions as strings).

The simulation is controlled by another data structure that is loaded with data from a simulator event file at the start of a simulation:

```
typedef struct {
    FILE     *filePtr;
    char     *fileName;
    char     *creationDate;
    char     *lastChangeDate;
    int      numberOfEvents;
    int      currentEventNumber;
    SimEvent **events;
    int      simStartTime;
  } SimEventFile;
```

Parameters filePtr and fileName contain information related to the simulator event file that is in use, while creationDate and lastChangeDate indicate the date of file creation and last modification. The other parameters are related with the simulation itself: number of events in file (numberOfEvents), indication of the event that is currently being accessed (currentEventNumber), a list of events (events), and the start time of the simulation obtained through the system clock (simStartTime).

When a simulation begins the global data structure is initialized with a default state, which mostly corresponds to null values of the parameters. Therefore, it is expected that the first event (that always occurs at simulation time zero) will fill the parameters accordingly to the desired initial scenario. All the subsequent actions are performed in an incremental fashion, i.e. only the changes are communicated.

The next section is devoted to provide a general overview of the editor tool. It includes also a description of the simulator event file format and of the encoding used to specify actions.

III. THE EDITOR

The main goal of the editor tool is the manipulation of simulation event files, which contain sequences of sets of actions intended to feed the simulator. Each set of actions is considered a *record* or *event* if it shares the same time tag, corresponding to the time of occurrence of that event. Therefore, each event may consist of several actions that will be performed "simultaneously" (this means that when the time of occurrence of the event arrives all the actions associated will be performed as soon as possible, but at a non-specified order).

When a new file is created the global data structure contains the same initialization values as in the case of a simulation. In every moment of editing the global data structure reflects all past actions that are recorded in the file until the previous event. The current event generates only the actions needed to change the state of the previous event to the state of the current. This means that the editing, as the simulation, is also incremental.

In this section we describe three main items related to the editor tool: its implementation, the format of the simulator event files, and the encoding of actions.

A. The implementation of the editor

The editor was implemented using terminal screen oriented menus. The curses package [1] (system V terminal screen handling and optimization package) was the main source of screen handling routines that we used to develop the editor. This package has the advantage to be independent of the specific terminal used to run the programs, since it uses the terminfo [2] terminal capability data base to look for the appropriate escape sequencies.

The main motivation to use a menu based editor was the reasonable complexity of the data structure that has to be handled during editing. This approach offers the advantage of easy move across all the data structure parameters making the changes in an arbitrary sequence. The organization of the menus follows closely the organization of the global data structure. For example, to the sub-structure alarm_event:

```
typedef struct alarm_event_type {
    Boolean     alarm_tag;
    id_type     CWP_id;
    id_type     resource_id;
  Int_16        alarm_no;
  char          alarm_message[MAX_mesg];
  Boolean       alarm_criticality;
  Int_8         alarm_type;
  time_type     alarm_log_time;
} alarm_event_type;
```

corresponds the menu items:

```
Alarm tag
    CWP id
    Resource id
    Alarm number
    Alarm message
    Alarm criticality
    Alarm type
    Alarm log time
```

The movimentation can be done using the cursor keys and the data is entered using fields with auto-validation. Some of the input fields offer a pre-defined list of choices that can be selected in an easy manner.

B. The simulator event file format

The simulator event files are ASCII files, starting with a header that is twofold. It is used to distinguish simulator event files from other files and also it provides information about the creation time and last change time of the file. Here is an example of the header of a simulator event file:

```
**************************************
      LAN Manager simulator event file
    Creation date: Thu Jul 21 12:34:11 1994
    Last change  : Thu Jul 21 12:38:14 1994
**************************************
```

Each event begins with a "Begin Event" string and ends with a "End Event" string. Following the "Begin Event" indication, two numbers (on separate lines) represent respectively the offset time (time in seconds since the previous event) and the absolute time (the time in seconds from the begin of the simulation) of the event. All the remaining information until the "End Event" mark is formed of encoded actions (one per line). Next is a short example of one of these files, containing just two events:

```
**************************************
      LAN Manager simulator event file
    Creation date: Thu Jul 21 12:34:11 1994
    Last change  : Thu Jul 21 12:38:14 1994
**************************************

Begin Event
0
0
0 0 0 0 1
0 0 0 1 1
0 0 0 4 3 8 25
0 0 0 9 0 3445
0 0 0 9 1 0
0 2 0 0 1
0 2 0 1 22
0 2 0 3 0 GATEW
0 2 0 3 10 12
End Event

Begin Event
10
10
1 0 0 1
1 0 1 1
1 0 4 This is an alarm!
1 0 5 1
1 0 7 1 20 7
End Event
```

The meaning of the encoded actions are explained in the next section.

C. The encoding of actions

In order to record on file changes of some of the parameters of the global data structure (performed during editing) and read them back during simulation we developed an encoding scheme that we explain briefly. The basic idea was to use the tree organization of the data structure as the mean to access the end parameters. To clarify the idea let us provide an example. Suppose we have the following data structure definition:

```
typedef struct type1 {
  int       P1;                 /* 0 */
  char      *P2;                /* 1 */
  double    P3;                 /* 2 */
} type1;

typedef struct type2 {
  char      *P4;                /* 0 */
  int       P5;                 /* 1 */
} type2;

typedef struct type3 {
  type1     p1;                 /* 0 */
  type2     p2[MAX_ELEMS];      /* 1 */
} type3;
```

Suppose also that only type1 and type2 are composed types, i.e. P1, P2, ... are leaves of the data structure tree. Note the commented numbers on each parameter. They number the parameters on an arbitrary but pre-determined manner. Therefore, to indicate that P2 has value "Testing string" we can use the following encoding:

0 1 Testing string

where the first number (0) indicates parameter p1 (of the root tree), the second number (1) indicates parameter P2 of the data structure type1, and finally the value of the parameter is represented. In the same way

1 5 1 1254

indicates that the structure field p2[5].P2 contains value 1254.

As can be easily understood, the code generator needed to encode the actions is straight forward to design although somewhat tedious to implement.

The parser used to interpret the encoded actions is also easy to design since there is no lookahead, i.e. at every node it is always known what data type should be expected.

A simulator event file decoder was implemented as an aid tool associated with the editor. The output generated by this program when applied to the simulator event file given as example in the file format section is the following:

```
* RECORD 1 (Abs. time = 0, Offset time = 0)
Elems.CWP[0].CWP_existing -> TRUE
Elems.CWP[0].station_id -> 1
Elems.CWP[0].login_time -> 03:08:25
Elems.CWP[0].CWP_secur.Sec_attempt -> 3445
Elems.CWP[0].CWP_secur.Sec_state -> FALSE
Elems.Net_elem[0].Net_elem_existing -> TRUE
Elems.Net_elem[0].Network_No -> 22
Elems.Net_elem[0].node.node_name -> GATEW
Elems.Net_elem[0].node.congestions -> 12

* RECORD 2 (Abs. time = 10, Offset time = 10)
alarm[0].alarm_tag -> TRUE
alarm[0].CWP_id -> 1
alarm[0].alarm_message -> This is an alarm!
alarm[0].alarm_criticality -> TRUE
alarm[0].alarm_log_time -> 01:20:07
```

REFERENCES

[1] curses(3X), CRT screen handling and optimization package. Hewlett-Packard Company, HP-UX Release 9.0, August 1992.

[2] terminfo(4), terminal capability database. Hewlett-Packard Company, HP-UX Release 9.0, August 1992.