SEnC – A Sequential Enif Client

(based on SEnC Version 3.5)

Heinz Spiess *

October 2003

Abstract

This paper describes the program SEnC (**S**equential **En**if **C**lient), which is a simple command line based Enif client which allows to communicate with an instance of the Enif program in order to automate simple repetitive tasks.

Before describing the SEnC program itself, the basic concepts of parameters and configurable objects, which are the foundation on which Enif's configurability is based on, are reviewed, followed by a general introduction of the Enif server mode and the command protocol used for the communication between the Enif server and Enif client programs.

In the second part of the paper the SEnC program is discussed, including detailed description of the calling sequence, the command line options, the variable substitution mechanism and the client commands (i.e. commands which are executed locally within SEnC), which include function definitions and conditional execution of commands.

While we hope that the availability of SEnC will be of direct practical help to those users who want to automate well defined repetitive tasks (such as e.g. the printing or exporting of standard set of graphic output), the paper might also be interesting to all those who need to program their own Enif client and just want to become acquainted with the basic principles of Enif's client/server interactions.

This paper is presented at the 17th Annual International EMME/2 Users' Conference, held in Calgary, Alberta, Canada, on October 22-24, 2003.

*EMME/2 Support Center, Haldenstrasse 16, CH-2558 Aegerten, Switzerland

Introduction

Besides the powerful transportation modeling capabilities offered by the EMME/2 transportation planning package [4, 1], one of its main features is certainly its macro language which allows complex procedures to be automated and standardized.

EMME/2's new graphic interface program Enif [3, 2] provides an even more powerful mechanism for controlling the programs from the outside. This is done using Enif's server mode, which allows external client programs to communicate with Enif via an openly specified TCP/IP protocol. Compared to EMME/2 macros, which are directly modeled after EMME/2's sequential dialogs, the client/server approach is more suitable to Enif's inherently non-sequential, event-driven operating mode. It allows arbitrary client programs to be implemented at the user level. Such a client can either be programmed to directly perform some application specific task, like e.g. implementing an interface to a web server which makes Enif generated data and graphics available on-line. Or, alternatively, it can implement a more general interface which only provides the necessary framework which can then be used, via some sort of scripting feature, to perform all kind of different tasks, without any need to write or modify the client program.

The SEnC program presented in this paper is a simple Enif client which belongs to the second category. Essentially, its goal is to read sequential Enif configuration commands from input files and feeds them to a running Enif server. This allows the SEnC program to be used as some kind of "macro" processor, which can be used to automate simple tasks, such as e.g. generating automatically the same standard set of graphic output for each new scenario that has been assigned, running an "automatic" Enif demonstration, or performing standardized performance benchmarks and/or software validation tests.

An important design goal of SEnC is that the program concentrates exclusively on interfacing with the Enif server protocol. This means that the SEnC program does not depend at all on any particular functionalities of Enif (except for the server's TCP/IP protocol, of course), nor does it make any assumptions on the type of Enif operations the user might want to control with SEnC. This ensures that a particular version of the SEnC program is not restricted to run on a particular release of Enif. So, as new functionalities are added to Enif, they may be readily accessed without the need to modify the SEnC program.

SEnC's "ignorance" of Enif's functionalities implies that all tasks that SEnC will perform must necessarily be defined from scratch in the input files processed by SEnC. In order to avoid having to "reinvent the wheel" each time SEnC is used for a new task, the input commands are usually read in two parts. First "reusable" header files are read in to define useful standard functionalities and bring them into an easily usable form. The file defining the actual task to be carried out is then based on the functions that are defined by the header file and are thus much shorter and easier to understand, even without in-depth knowledge of the internal workings of Enif.

Enif Parameters and Configurable Objects

In order to understand how an Enif client, including the SEnC program described later on, interacts with the Enif server, it is necessary to have some basic knowledge on Enif's general configuration mechanism. But note that the following explanations are limited to provide the bare minimum needed for a basic understanding. A more complete and detailed description of the concepts outlined below can be found in [2].

The fundamental building block of Enif's configurability is the **Parameter**. In a nutshell, a parameter is defined by:

- an identifying name and a plain text description,
- its type (such as Click, Bool, Integer, Float, String, Expression, Selector, Stylus, ...) defining the type of data stored,
- a set of optional flags specifying special properties of the parameter,
- an optional group specification consisting of *send*, *receive* and *update group*.
- one or more indexed values of the specified type.

Enif's **grouping mechanism** is activated whenever the value of a parameter is changed. If the parameter has a non-empty *send group*, a group signal is propagated to other parameters, causing those parameters having the same *receive group* (and a compatible type) to synchronize automatically to this same value. Parameters having the same *update group* will, instead of changing their value, perform an update action, which depends on the type of the parameter and its properties.

The parameters are used to provide configurability to Enif's functionalities. Each of Enif's basic functionality (e.g. plot, mapper, list, configurable attribute, . . . , or also the Enif TCP/IP server) constitutes a **configurable object**. Each configurable objects has it's own *set of parameters*, or, in other words, each parameter is *owned* by a configurable object.

The configurable objects are organized as an **object tree**, which implies that a configurable object has a *parent object* and can create its own *child objects*.

The parameter grouping mechanism described above is not restrained to the set of parameters of a single configurable object, but it can also be used to synchronize parameters belonging to different configurable objects. This is done by propagating the group signals along the configurable object tree. In order to control the propagation of the group signal, each configurable object has its own *group filters* (implemented themselves as string parameters) which define which groups can enter and leave the object. The figure below shows a typical Enif object tree, as it is displayed by Enif in configuration mode when selecting the option "Show configurable object tree":

enif

Configurable object tree

Name	Type	Incoming groups	Outgoing groups
Enif root object	Root	all groups	all groups
preferences	Preferences	all groups	all groups
General parameters	Configurable parameter list	all groups	all groups
Enif instance 0	Enif	no groups	global groups
View control	View control	global groups	global groups
Network control	Network control	global groups	global groups
Network 1	Network	all groups	all groups
Legend plane	Legend plane	all groups	all groups
Network plane	Network plane	all groups	all groups
Plot control	Plot control	all groups	all groups
Plot description	Plot description	global groups	global groups
Bare network	Plot	global groups	global groups
Background	Background	all groups	all groups
IncludeFile	Inclusion	all groups	all groups
Link base	Link base	all groups	all groups
PrintLegend	Inclusion	all groups	all groups
PrintLegend	Legend	all groups	all groups
application specific attributes	Configurable attribute list	global groups	global groups
tcp server	Enif TCP server	all groups	all groups

Update Done

Having understood the above mechanism, it is easy to see that storing or loading an Enif configuration (e.g. a plot or a list) can be done simply by writing or reading a file which contains the list of the corresponding configurable objects, and for each object the specification of its set of parameters. Given that the configurable object itself defines name, description and type for each of its parameters, the bulk of the configuration files (such as `*.e2p` plot configurations or `*.e2l` list configurations) consists in commands specifying the parameter's values and group specifications. In their simplest form, the groups of a parameter are specified with commands of the type

parameter* : *send[/receive[/update]]

and parameter values are specified with commands like

parameter = value

for non indexed parameters (i.e. value at index 0) or

parameter*[*index*] = *value

for setting the value for a specified index.

As explained later, the above parameter specification command are not only used in Enif's configuration files, but in exactly the same way they also constitute the basic Enif server mode commands for changing the values and characteristics of Enif parameters.

Some Useful Enif System Parameters

In Enif, the configurable objects are divided into *system objects* and *user objects*. User objects correspond to those that are loaded by the user from corresponding configuration files, whereas all configurable objects that are generated and maintained directly by the Enif program are referred to as system objects.

Parameters associated with system objects are called *system parameters*. Their name always starts with a dollar sign and their group specification (which, of course, is system defined as well) usually corresponds to the parameter name.

Many of the basic tasks that one would like to automate, are controlled by system parameters, such as e.g.

- changing the network scenario,
- loading a plot configuration,
- changing the current view,
- printing the current plot,
- exporting the current plot to an image file,

Thus, it is useful to look at some of available system parameters that are of particular practical importance. In the following, we briefly explain those system parameters which allow performing the above tasks and which will be used in the examples later on:

`$LoadScenario` (Type Integer, write-only, owned by Network control)

This write-only parameter allows manipulating the network stack. Writing an integer value into index 0 will load ("push") the corresponding scenario as an additional network onto the network stack. Writing an integer value into index $i > 0$ will replace the network at position i in the network stack by the corresponding scenario. After the operation has been executed, the values will always be reset to zero, in order to be ready for the next operation.

`$ScenarioNumber` (Type Integer, read-only, owned by Network)
`$ScenarioTitle` (Type String, read-only, owned by Network)

These two system parameters contain the scenario number and the scenario title of the network which is currently loaded at position 1 of the network stack.

`$LoadPlotConfiguration` (Type String, write-only, owned by Plot control)

If a file name is written into this write-only system parameter, the file is opened and the corresponding plot configuration is loaded and displayed.

`$CurrentView` (Type Box, read-write, owned by View control)

This parameter always contains the coordinates of the current view. Since it is a read-write parameter (i.e. both its send and receive group is defined as `$CurrentView`), it can be used for retrieving the coordinates of the current view (by receiving its value) as well as imposing a new view (by sending the new coordinates to `$CurrentView`.

`$PrintCurrentViewNoSetup` (Type Click, write-only, owned by Network plane)

If this parameter is activated by a receive or update group signal, it will cause the current view to be printed using the current default printer and printer settings.

`$ExportEnlargementFactor` (Type String, read-write, owned by Network plane)

This parameter contains a small integer factor (1,2,3,. . .) which is used to enlarge the resolution of exported image files by the corresponding factor. Using enlargement factors larger than one allows the generation of bitmapped image files that exceed the current screen resolution. Note, however, that the file size of the exported image, as well as the time needed to generate the image, increases with the square of the enlargement factor – so don't exaggerate!

`$ExportScreenViewToFile` (Type String, write-only, owned by Network plane)

If a file name is sent to this system parameter, an image file is generated containing the current screen view, using the image format which by default corresponds to the specified file extension (such as `.jpg`, `.png`, `.bmp`, etc.).

`$ExportPrintViewToFile` (Type String, write-only, owned by Network plane)

If a file name is sent to this system parameter, an image file is generated containing the current print view (i.e. the same information which would be sent to the printer), using the image format which by default corresponds to the specified file extension (such as `.jpg`, `.png`, `.bmp`, etc.).

Communicating with an Enif Server

Enif's server mode is activated by calling Enif with the command line option "`-S` *`port`*" (where *port* is the number of the TCP/IP port which is to be used by the server). The only difference, compared to running Enif not in server mode, is that this option will install an Enif TCP/IP server configurable object as a child object of the primary Enif instance. This server object binds itself to the specified TCP/IP port and waits for client programs to open a connection.

When a client opens a connection, an Enif socket object is created as a child of the server object and a communication stream is established between the external client program and the corresponding socket object. When created, this object only defines the standard parameters needed for all objects (such as name, description, group filters, . . .), leaving it up to the client program to create additional

parameters as needed.

The client program can now send commands to Enif which then will be executed within Enif and the result will be sent back to the client. The result always ends either with positive or negative acknowledge, usually “OK” to indicate a successful command, respectively “KO” to indicate that the command could not be executed correctly.

The complete list of Enif server commands along with detailed technical specifications can be found in [2]. Here, we will just mention some of the more important ones:

`lp` ***[parameter]***
: List the specified parameter or, if no parameter is given as argument, list all parameters of the current object.

`new` ***partype parname [elementtype]***
: Create a new parameter of type *partype* with the name *parname*. The most used parameter types *partype* are `Click`, `Bool`, `Integer`, `Float`, `Expression`, `Selector`, `String`, `Stylus` and `Box`. For parameters which need further specifications, these are given after the parameter name. For expressions and selectors, e.g., the type of network element they refer to is given by *elementtype* as `Nodes`, `Links`, `Lines`, `Segments`, `Turns`, etc.

`delete` ***parname***
: This command is used to delete the parameter *parname* which was previously created with a “new” command.

`check` ***expressionpar***
: Check if the given parameter *expressionpar*, which must be of type Expression or Selector, contains a valid expression. The type of acknowledgment (`OK` resp. `KO`) indicates if the expression is valid or not.

`echo` ***anytext***
: The given text *anytext* is passed through Enif’s parameter substitution to replace occurrences of the type `%<...>%` and the result is returned to the client.

`exit`
: Close the socket and terminate the communication stream between the client program and Enif. Note that this command does not terminate the Enif process.

Besides the above server commands, the server also accepts the standard parameter specification commands which we already encountered in an earlier section of this paper, namely:

parameter **:** ***sendgroup[/receivegroup[/updategroup]]***
: Set the group specification of *parameter*.

parameter **=** ***value***
: Set the value of a non-indexed parameter (resp. set value at index 0).

parameter*[*index*] = *value
: Set the value of an indexed parameter.

Having learned about the Enif server commands, let’s try them out now. Even without having access to a specific Enif client program, it is possible to do this using the standard **telnet** program.

To do this, just start Enif in server mode using e.g. port 9090 with the command “`enif -S 9090`”. Now, from a unix shell or CMD window, you can contact the Enif server with the command “`telnet localhost 9090`” and try out some of the above commands.

The first example just changes the current view from its current setting to the coordinate window (-1,-1,1,1). To do this, a Box type parameter named `view` is created and its send group is set to `$CurrentView/`, i.e. the receive group of the corresponding system parameter. When we then change the value of the parameter `view` to "-1,-1,1,1", Enif's parameter grouping will propagate a send group signal which is caught by the `$CurrentView` system parameter, which causes Enif's view to be changed accordingly.

```
%telnet localhost 9090
Trying 127.0.0.1...
Connected to localhost (127.0.0.1).
Escape character is '^]'.
OK
new Box view
OK
view : $CurrentView/
OK
view= -1,-1,1,1
OK
exit
Connection closed by foreign host.
%
```

The second example contacts the enif server to generate a list of all links having an auto volume of more than 3000, sorted according to decreasing link volumes. To do this, a link expression parameter named `linkval` and a link selector parameter named `linksel` are created and their values are set to the desired expressions. The `check` command is not absolutely necessary, but it would show if the specified expression would not be valid (e.g. if the current scenario would not have an auto assignment). The `eval` command, finally, is used to evaluate the expression `linkval` by iterating through all links selected in selector `linksel` and the corresponding results are output:

```
% telnet localhost 9090
Trying 127.0.0.1...
Connected to localhost (127.0.0.1).
Escape character is '^]'.
OK
new Expression linkval Links
OK
new Selector linksel Links
OK
linkval = volau,volad,timau,speedau
OK
check linkval
OK
linksel = volau>3000,-volau
OK
eval linkval linksel
609-608;3329.708;55;2.508259;8.133132
458-59;3209;0;1.2;10
705-713;3140.356;23.07692;0.617797;11.65431
611-610;3132.645;55;2.741341;10.06806
665-674;3105.219;0;0.811095;23.67171
668-665;3105.219;0;1.066585;12.37595
669-668;3105.219;0;1.308991;12.37595
516-608;3035.614;29.79895;0.454977;26.37496
517-516;3035.614;29.79895;0.432228;26.37496
607-606;3027.583;159.174;1.369371;10.07762
```

```
610-609;3016.525;55;1.423182;11.38294
OK
exit
Connection closed by foreign host.
%
```

These two little examples using telnet should already give you an idea on the principles Enif's server is built upon. We now turn our attention to SEnC, a simple Enif client program which processes server commands that are read sequentially from given input files.

Calling the SEnC Program

The SEnC program is called as follows:

senc *[options] [file|varspec ...]*

where the specified input files contain the commands which are to be processed by the program. For the server commands discussed in the last section, this just implies sending the commands one by one to the Enif server and receiving the corresponding results.

The following command line options are recognized:

-d Generate debugging output. Specifying this option several times will increase the amount of debugging output which is generated.

-e Continue with input processing after errors. Unless this option is set, SEnC will stop processing the input as soon as it received a negative acknowledge (`KO`) from the Enif server.

-h Display this help summary.

-i Run SEnC in interactive mode. In interactive mode, the processed commands as well as the received replies are displayed in a separate window. In addition to executing commands read from the input file, in interactive mode it is also possible to enter server commands directly from the keyboard.

-l Launch the Enif program on the local machine upon starting SEnC and exit Enif when SEnC terminates. After having created the Enif process, SEnC waits for 3 seconds until it tries to contact the Enif server. when accessing a very large network and/or when using a slow or otherwise busy system, this delay may not always be sufficient. In this case, the `-l` option can be repeated more than once, each time adding another 3 seconds to the delay. Note that under Windows, SEnC tries to execute the program `senc.exe`, under Unix-like systems the program `senc` is used. In both cases, the corresponding program if first looked for in the standard locations pointed by the environment variables `ENIFPATH` or `EMME2`. Otherwise the called program must be located in a directory which is included in the user's `PATH` environment variable.

-L *progname* This option is similar to the `-l` option, but it contains an additional argument which allows the specification of the program name that is used to launch Enif. By default the program name `enif` is used. This option may be useful in situations (like testing of benchmarking) where several version of Enif coexist on the same system.

-o *outputfile* Direct the standard output to the specified file.

-p ***port*** This option defines the TCP/IP port used to communicate with the Enif server.

-q Quit after all input files are processed. This is the default unless running in interactive mode using the `-i` option.

-s ***server*** This option allows the specification of the host name or IP-address of of the system on which the Enif server is running. By default, the Enif server is assumed to run on `localhost`.

-v Generate verbose output.

SEnC also contains a simple mechanism which allows to perform **variable substitution** on the processed input commands. This is done by specifying on the command line, before or between input files, arguments of the form *variable*=*value* (no spaces allowed!). If such an argument is processed, SEnC will create a corresponding variable, if it does not exist yet, and set its value to string left of the equal sign. Variables which are defined in this way can be used for substitution in commands which are subsequently processed. If such a command contains the string {*variable*}, it will be replaced by the current value of the parameter before the command is executed.

Let's now come back to our second telnet example of the previous section. Assume that, for whatever reason, we would need to automate the process of creating a list of links with auto volumes exceeding 3000. Having SEnC, we could simply store the server commands we previously typed into the telnet session into a file, say, named `example1`

```
new Expression linkval Links
new Selector linksel Links
linkval = volau,volad,timau,speedau
linksel = volau>3000,-volau
echo Scenario %<$ScenarioNumber>% (%<$ScenarioTitle>%):
eval linkval linksel
```

and then run SEnC with the command

```
senc -l example1 >linklist.out
```

to obtain in the file `linklist.out` the lines:

```
Scenario 3000 (Kildonan Corridor):
609-608;3329.708;55;2.508259;8.133132
458-59;3209;0;1.2;10
705-713;3140.356;23.07692;0.617797;11.65431
611-610;3132.645;55;2.741341;10.06806
665-674;3105.219;0;0.811095;23.67171
668-665;3105.219;0;1.066585;12.37595
669-668;3105.219;0;1.308991;12.37595
516-608;3035.614;29.79895;0.454977;26.37496
517-516;3035.614;29.79895;0.432228;26.37496
607-606;3027.583;159.174;1.369371;10.07762
610-609;3016.525;55;1.423182;11.38294
```

If in addition, we want to be able to specify the scenario for which these links should be listed, we can change the input file to select the scenario by grouping with the system parameter `$LoadScenario` and use a variable which can then be set on the command line. The modified example input file `example1a` becomes then:

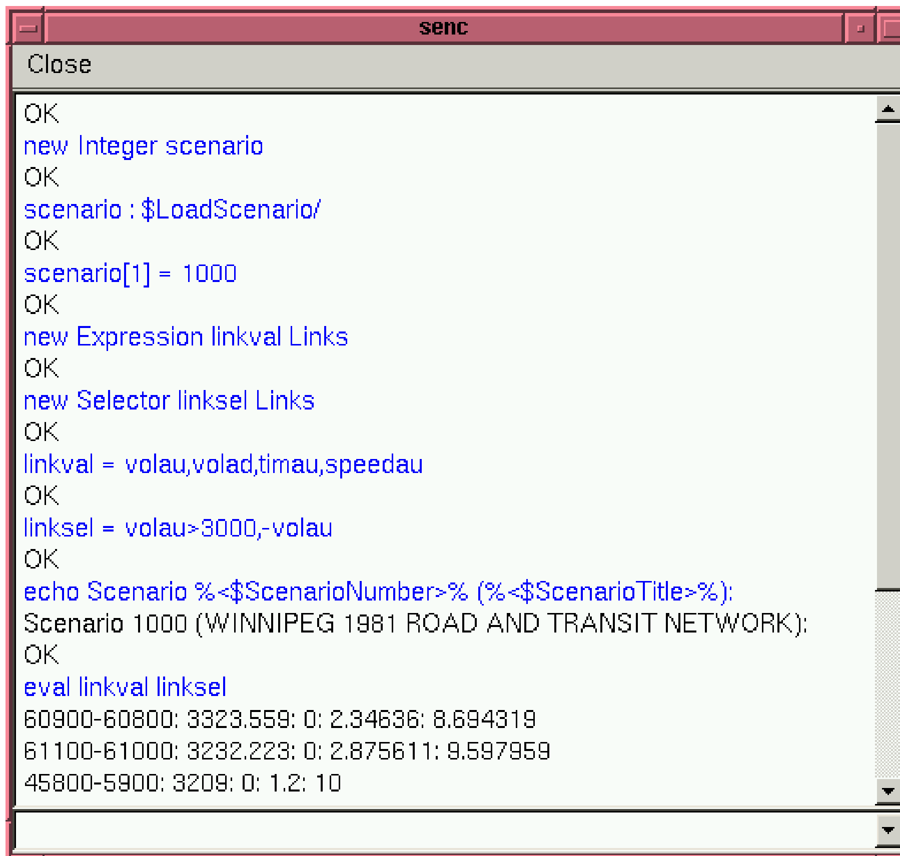
```
new Integer scenario
```

```
scenario : $LoadScenario
scenario[1] = {scen}
new Expression linkval Links
new Selector linksel Links
linkval = volau,volad,timau,speedau
linksel = volau>3000,-volau
echo Scenario %<ScenarioNumber>% (%<ScenarioTitle>%):
eval linkval linksel
```

We can produce the link list for a specific scenario, say scenario 1000, by calling SEnC as follows:

```
senc -l scen=1000 example1 >linklist1000.out
```

We conclude this example with a screen shot of the interactive mode display window that is displayed if the option `-i` is added to the above command. On screen, the displayed texts are color coded: the server commands sent from SEnC to the Enif server appear in blue, the output received from the Enif server in black, the client commands (which are discussed later on) in green, and interactively entered server commands are shown in red.



Note that the above example is also interesting since it does not use any of Enif's graphic capabilities at all, but it just uses Enif to access data from the EMME/2 data bank, using Enif's powerful expression and selector features. This opens the possibility for external programs which need to access some data from the EMME/2 data bank to incorporate an Enif clients and access the data bank via the Enif client/server interface. It avoids having to run EMME/2 macros to first punch out the desired data to sequential files from which it can then be read into the external program, or it provides a safe and protected alternative to the potentially dangerous method of having the external program read directly binary data from the EMME/2 data bank.

Combining Enif server command with SEnC's variable substitution mechanism already allows writing

more flexible scripts, in which some of the information is not “hard-coded” but can be provided at run time, without any need to edit the command file.

SEnC Client Commands

In order to provide possibilities for even more powerful Enif scripting, SEnC also supports its own set of **client commands**, i.e. commands which are not sent to the Enif server, but which are processed locally by the SEnC program.

Client commands are easily recognizable since they all start with an exclamation mark. If such a line is detected in the input stream, instead of being sent to the Enif server, it passed to a SEnC’s local command parser.

The current version of SEnC supports the following client commands:

`!include` ***filename***
: This command causes SEnC to open the specified file and process the commands it contains before continuing with the next command. The `!include` command can be used to read in header files containing required function libraries (as an alternative to specifying the required header files directly in the command line). Also, the `!include` command is convenient in interactive mode for telling SEnC to process an input file.

`!print` ***anytext***
: This command prints the given text to the standard output, after having passed it through SEnC’s variable substitution (if any), but without passing it on to the Enif server. Thus, if the text does not contain any Enif parameter substitutions `%<...>%`, the print command produces the same output as a corresponding `echo` server command, but is much more efficient, since it is handled locally.

`!wait` ***seconds***
: Pauses for the specified number of seconds (which can be specified with decimals if small delays are required) before the next command is processed. This command may be useful for automated demonstrations in which each image should stay on the screen for some specific amount of time.

`!prompt` ***message***
: Prompt the user with the specified messages and wait for the user to enter an answer. The typed in answer will be written to the special variable `ANSWER` which can be accessed via variable substitution by putting the string “`{ANSWER}`” as a place holder. This command may be useful in situations where the user should be given time to take some action (e.g. interacting directly with Enif) before the processing of the commands should be continued.

`!exit` ***status***
: Terminate SEnC immediately, without processing the remaining commands. If an integer value is specified as argument, it is used passed to the operating system as exit status code.

`!new` ***variable* = *value***
: Create a new variable or, if the variable already exists, create a new instance of this variable and set it to the specified value.

`!set` ***variable* = *value***
: Set the topmost instance of the specified variable to the given value. If the variable does not yet exists, create it.

`!delete` ***variable [variable ...]***

Delete the topmost instance of the specified variables.

`!replace` ***variable regexp [newstring]***

In the specified *variable* all occurrences of the regular expression *regexp* are placed by the string *newstring*. If newstring is not specified, a single blank is used by default.

`!begin`
...
commands
...
`!end`

Command enclosed between a `!begin` and an `!end` command constitute a **command block** which can be used instead of a single command in composed commands that are described below.

`!function` ***fctname [argname ...]***

The `!function` client command defines the function with the given function name to consist of the line or the following block of lines that is included between `!begin` and `!end` commands. Optional function arguments can be defined after the function name. Any occurrence of one of the function arguments enclosed in braces, {*argname*}, will be replaced by the corresponding argument value with which the function is called via the `!call` client command. Note that `!function` commands are not allowed within function definitions.

`!call` ***fctname [argvalue ...]***

Call the function *fctname* using the argument values which are specified after the function name to substitute the function arguments.

`!return`

The `!return` command can be used in function definitions to return from the function before having reached the end of the function. Note that before returning from the function, it is up to the programmer to delete any private instance of variables which have been created with `!new` client commands within the function body.

`!if` ***value***
`!if` ***val1 compop val2***

The `!if` command cause the execution of following command or command block to be conditional to the specified test. If the first form is used, i.e. if only one value is specified, the condition is only met when the specified value is either a non-zero numerical value or the string "`yes`". If the second form is used, the two values *val1* and *val2* are compared using the comparison operator *compop*, as shown in the following table:

String comparisons:		Numeric comparisons:			
`==`	equal	`=`	equal	`<>`	not equal
`!=`	not equal	`>`	larger than	`<`	less than
		`>=`	larger or equal	`<=`	less or equal

Note that the comparison operator must always be separated by at least one blank from values which are compared, i.e. the command "`!if {X} > 1`" will work as expected, whereas the command "`!if {X}>1`" will not!

`!iferror`

This command is similar to the `!if` command described above. But instead of testing the given argument values, its condition is the acknowledgment received in response to the last server command. If the preceding server command was successful (`OK` reply), the following command or command block is skipped, whereas if it returned with an error (`KO` reply), the following command or command block is executed.

!else

The `!else` command can optionally follow just after the conditional command or command block of any `!if` or `!iferror` command. Only if the condition of the preceding if-command was not satisfied, the command or command block following the `!else` command will be executed. The following example illustrates the implementation of an if-then-else test:

```
!if 0 <= {OFFSET}
   !begin
      !print positive offset -> shifting to the right
      !call rightshift 0 {OFFSET}
   !end
!else
   !begin
      !print negative offset -> shifting to the left
      !call leftshift {OFFSET} 0
   !end
```

!for ***variable startval endval [increment]***

The `!for` command implements a for loop by iterating the loop variable specified as the first argument from a start value up to and not exceeding a end value using, if specified, the given increment or else the value 1. The command or command block immediately following the `!for` command is executed for each of the values assigned to the loop variable.

E.g. the sequence

```
!for scenario 1000 5000 1000
  !call generateStandardPlots {scenario}
```

will call the function `generateStandardPlots` for each of the loop value, i.e. it is equivalent to the following commands:

```
!call generateStandardPlots 1000
!call generateStandardPlots 2000
!call generateStandardPlots 3000
!call generateStandardPlots 4000
!call generateStandardPlots 5000
```

The specified specified start, end and increment values can contain decimals. If the increment value is zero or has a different sign than the difference between end and start value, the command or command block following the `!for` command is not executed at all.

!foreach ***variable val1 val2 val3*** **...**

The `!foreach` command command executes the following command or command block once for each of the values *valX* as arguments. For each given value, the specified variable is first set to this value and then the line following the `!foreach` is executed.

E.g. the sequence

```
!foreach scenario 2000 2001 2002 3001 1000
  !call generateStandardPlots {scenario}
```

will call the function `generateStandardPlots` for each of the loop value, i.e. it is equivalent to the following commands:

```
!call generateStandardPlots 2000
!call generateStandardPlots 2001
!call generateStandardPlots 2002
!call generateStandardPlots 3001
!call generateStandardPlots 1000
```

!while ***value***

`!while` ***val1 compop val2***

The syntax of the `!while` client command corresponds to the `!if` client command. The command or command block following immediately the `!while` command will be executed for as long as the specified condition is true.

In addition to the client commands which are described above, SEnC also recognizes any input line which starts with **#** as a **comment line** and does not process it any further.

Variables

SEnC allows the definition of arbitrary string valued variables. The name of a variable is composed of upper and lower case letters, digits and the underscore character ("_"), always starting with a letter. Variable names are case sensitive.

Variable substitution occurs whenever a variable name enclosed in braces (e.g. "`{MyVar}`") occurs in a command.

All SEnC variables are global, i.e. once defined they can be accessed from any command, regardless of the context. In order to avoid variable collision conflicts when using variables in a function, SEnC supports explicit **variable instancing** using the client commands `!new` and `!delete`. A `!new` command always creates a new instance of the variable with the given name. This new topmost instance "hides" all lower level instances of the same variable (if any), so that all following `!set` commands and variable substitutions will act on this instance until either another new instance is created on to of the current one, or this instance is deleted. After a `!delete` command, the next lower instance, if one exists, becomes again accessible.

This mechanism of variable instancing makes it possible to write functions that will not cause problems even if they are called from other contexts in which the same variable name is already used for some other purpose. On the other hand, it is also possible to share variables between different contexts and/or functions, which is the default behavior when no new instances are created/deleted.

As the variable instancing is based on the explicit calls to `!new` and `!delete`, it is the user's (or "SEnC programmer's") responsibility to make sure that, for normal use, for each `!new` command there is a corresponding `!delete` command. Of course, a more advanced used can also use (or "abuse"?) variable instancing for implementing more advanced mechanisms, such as e.g. a value stack, where `!new` is used as **push** and `!delete` as **pop**.

SEnC creates no variable on its own. But some variables, once they have been created, will take a special meaning:

`ANSWER` Contains the last answer given to a `!prompt` command.

`OUTPUT` Contains last output line received from the Enif server.

`FILENAME` Contains the name of current input file.

`LINENUMBER` Contains the current line number in current input file.

`NOECHO` By default, SEnC echoes all output it receives from the Enif server (except for the `OK`/`KO` acknowledgments) to the standard output. If the variable `NOECHO` exists, the output received from the Enif server is not echoed.

Note that in order to be able to access any of the above variables, it must first be explicitly created, either on directly the command line or using a `!set` or `!new` client command.

A Simple Example

In the following we show how server commands, variable substitution and client commands can be combined in order to create a small environment which can be used for generating Enif plots automatically.

In the following header file named `plotgen.sen`, all necessary Enif parameters are created and grouped to the corresponding Enif system parameters with which they have to synchronize their values, and some functions are defined which can afterward be called to perform specific tasks:

```
new Integer Scenario
Scenario : $LoadScenario/

new String PlotConf
PlotConf : $LoadPlotConfiguration/

new Box View
View : $CurrentView/

new Click FullView
FullView : $FullView/

new Integer ExportEnlargementFactor
ExportEnlargementFactor : $ExportEnlargementFactor/

new String ExportScreenView
ExportScreenView : $ExportScreenViewToFile/

new Click PrintView
PrintView : $PrintCurrentViewNoSetup/

!function scenario SCENARIO
!begin
  !if 0 < {SCENARIO}
  Scenario[1] = {SCENARIO}
  echo Scenario %<$ScenarioNumber>%: %<$ScenarioTitle>%
!end

!function view VIEW
!begin
  !print Changing view to {VIEW} ...
  !if full == {VIEW}
    FullView = 1
  !if airport == {VIEW}
    View = -10;-3;-7;3;Airport;
  !if cbd == {VIEW}
    View = -3;-3;3;3;CBD;
  !if kildonan == {VIEW}
    View = -1.2;1.5;4.2;4.8;Kildonan corridor;
!end

!function plot CONFIGURATION
!begin
  !print Loading {CONFIGURATION} ...
  PlotConf = {CONFIGURATION}
!end

!function action
```

```
!begin
  # variable ACTION is defined outside!
  !if prompt == {ACTION}
    !prompt Continue?
  !if wait == {ACTION}
    !wait 5
  !if print == {ACTION}
  !begin
    PrintView = 1
    !print Printing view ...
  !end
!end

!function export FILENAME
!begin
  ExportScreenView = {FILENAME}
  !if export == {ACTION}
    !print Exporting view to file {FILENAME} ...
!end
```

The above header file does not produce itself any visible results, but it implements the five simple functions which can afterward be called by the task definition file as follows:

`!call scenario` ***scenarionumber***
: Loads the scenario with the given number into the primary network.

`!call view` ***viewname***
: Changes to the current view to one of the predefined views `full`, `cbd`, `airport` or `kildonan`.

`!call plot` ***plotconfiguration***
: Loads the specified plot configuration file.

`!call action [wait|prompt|print]`
: This function represents a general processing routine that can be called to perform some action after a plot has been generated. Which action is taken by this function depends on the supplied parameter. If it's called with the parameter `wait`, it will wait for a few seconds before proceeding with the following commands. If it's called with the parameter `prompt`, it will issue a prompt message and wait for a user input before proceeding. If it's called with the parameter `print`, it will generate a printout of the plot. Finally, if none of these parameters is specified, no action is taken and the processing of the following commands continues.

`!call export` ***filename***
: Generates and exports an image file containing the current view. If needed the value of the parameter `ExportEnlargmentFactor` can be changed before calling this function, in order to change the resolution of the generated image file.

Based on the commands defined in the header file `plotgen.sen` it is now possible to define a specific plot generation task quite easily. E.g. the following task file `volumeplots` generates auto volume plots for three different views and one transit volume plot for the CBD:

```
!include plotgen.sen
!call scenario {SCENARIO}
!call plot autovol.e2p
!call action
!call view cbd
!call action
```

```
!call view kildonan
!call action
!call plot transitvol.e2p
!call view cbd
!call action
```

If we now call SEnC with the header and the task file, i.e.

```
senc -l volumeplots
```

The four plots will be generated one after the other for the default scenario of the data bank. But this will be done very quickly and except for a rapidly flickering screen, not much will be seen.

We can now use variable substitution to define the scenario to be used with a command line argument of the form “`SCENARIO=...`”, as well as the action to be taken for each plot with an argument “`ACTION=...`”. Note that for achieving the desired results, these variable specifications should be put after the header file and before the task file.

Thus, the command

```
senc -l ACTION=wait SCENARIO=1000 volumeplots
```

will generate the same plots as before, but this time for the specified scenario 1000, and after each plot it will pause for a few seconds to give the user a chance to look at it.

Similarly, the command

```
senc -l ACTION=print SCENARIO=3000 volumeplots
```

will generate the four plots for scenario 3000 and send them to the default printer. Reduced versions of the four resulting printouts are shown below:

Some More Advanced Examples

The example in the preceding were chosen to be very simple so that a novice SEnC user can easily understand them. In this section, we will show a more advanced example, in which the Enif server commands and SEnC client commands are combined to perform more elaborate functions. This section is really meant for the more advanced SEnC and Enif users. So if you are reading the first time about SEnC and have not actually used SEnC yet, you can safely skip the following and move on to the next section.

For this, let us consider the function `view` used in the preceding section. It could be called to change the view to one of the named views which were coded right into the function code. Since each Enif application contains itself a set of predefined views, let us try to rewrite the function `view` in such a way that it actually looks up the the given view name in the Enif application and, it a predefined view with the given name exists, sets the current view accordingly.

For implementing this task, we first have to know where Enif stores the predefined views. This is done in an indexed Box type parameter called `PredViews` which is owned by the configurable object

containing the preferences. E.g. in my Winnipeg application, the `PredViews` parameter have the following definition:

```
PredViews[0] = -10;-3;-7;3;Airport;
PredViews[1] = -1.2;1.5;4.2;4.8;Kildonan;
PredViews[2] = -3;-3;3;3;CBD;
```

The easiest way of accessing this information is using Enif's parameter substitution mechanism via the Enif server command `echo`, which was discussed earlier. Using e.g. a the command "`echo %<PredViews[1]%>`" will in our example data bank return the string "`-1.2;1.5;4.2;4.8;Kildonan;`" So what we need to do is loop through the indices of the `PredViews` parameter and find the one with the given name.

Before actually writing the new version of our function `view`, we will need to define first an auxiliary functions. This function is called `get` and allows retrieving one of the given arguments by its positional numbers and set the specified variable to the corresponding argument:

```
!function get VAR POS V1 V2 V3 V4 V5 V6 V7 V8 V9
!begin
  !set {VAR} = {V{POS}}
!end
```

Now we have all the ingredients to write the new version of our function `view` as follows:

```
!function view VIEWNAME
!begin
  !new OUTPUT = 1
  !new NOECHO = 1
  !new PVIEW =
  !new PNAME =
  !new I =
  !for I 0 999
  !begin
    echo %<PredViews[{I}]>%
    !set PVIEW = {OUTPUT}
    !if x == x{PVIEW}
    !begin
      !delete OUTPUT NOECHO PVIEW PNAME I
      !return
    !end
    !set PNAME = {PVIEW}
    !replace PNAME ;
    !call get PNAME 5 {PNAME}
    !if {VIEWNAME} == {PNAME}
    !begin
       View = {PVIEW}
       !delete OUTPUT NOECHO PVIEW PNAME I
       !return
    !end
  !end
  !delete OUTPUT NOECHO PVIEW PNAME I
!end
```

The function first defines new instances of the variables `OUTPUT`, `NOECHO`, `PVIEW`, `PNAME` and the loop variable `I`. This means that we have to make sure that these variable instances are deleted before returning from this function.

The main part of the function is a for-loop which iterates the variable `I` through the index values 0, 1, 2, . . .

For each index, the `echo` server command is used to write the corresponding view box value into the SEnC special variable `OUTPUT`. If the value is empty, this indicates that the parameter `PredViews` does not contains a value at the corresponding index, i.e. that we have searched through all predefined views without having found the specified view. In this case, a message is issued and we return from the function without having changed the view.

Once we have obtained a valid view box value, we now break it into its components *xmin ymin xmax ymax name [description]* by replacing the separators ";" by blanks. We can now use the previously defined function `get` to extract the view's name from the fifth field and write it to the variable `PNAME`.

Finally, we have to compare the name of the current predefined view with the view which was asked for. If the names match, we set the `View` parameter (remember, it is grouped to Enif's current view!) to the corresponding value, at which time the view displayed by Enif will change to the requested view. The task is now completed and we can return from function `view`.

We can now e.g. try out the new function by calling it as to find and set the "Kildonan" view:
`!call view Kildonan`

The last example show how the SEnC / Enif framework can be used to do computations. As an example, let's try to write a function `zoom` which implements a concentric zoom-in or zoom-out of Enif's current view. The zoom factor is provides function argument, where values > 1 implying a zoom-in and values < 1 a zoom-out operation.

Again, we first need to define some auxiliary functions. The first function is called `shift` and is used to discard the first of a set of arguments and set a specified variable to the remaining arguments:

```
!function shift VAR V0 V1 V2 V3 V4 V5 V6 V7 V8 V9
!begin
  !set {VAR} = {V1} {V2} {V3} {V4} {V5} {V6} {V7} {V8} {V9}
!end
```

The second function is in itself very interesting. It is called `eval` and sets the variable specified in the first argument to the result of the expression given in the second argument. E.g. the command "`!call eval X (1+0.5*(7-2))`" will set evaluate the expression $(1 + 0.5 * (7 - 2))$ and set the variable `X` to the result value, i.e. after the command we have `X`$= 3.5$.

```
!function eval VAR EXPRESSION
!begin
  !new NOECHO = 1
  !new OUTPUT = 1
  new Expression eval_expression Nodes
  new Selector eval_selector Nodes
  eval_expression = {EXPRESSION}
  eval_selector = index==1
  check eval_expression
  !iferror
  !begin
     !print Invalid expression {EXPRESSION}
     !set OUTPUT = 0 0
  !end
  eval eval_expression eval_selector
  !call shift {VAR} {OUTPUT}
  !delete NOECHO OUTPUT
  delete eval_expression
```

```
  delete eval_selector
!end
```

Since SEnC does not have any client commands which perform calculation, function `eval` actually uses Enif to perform this task. For this, an expression and a selector are set up accordingly and the `eval` server command is used to actually perform the computation. The result is then extracted and stored in the specified variable.

With the `eval` function, implementing the previously mentioned `zoom` function now becomes relatively straight forward:

```
!function zoom ZOOMFACTOR
!begin
  !new NOECHO = 1
  !new OUTPUT = 1
  echo %<$CurrentView>%
  !new CVIEW = {OUTPUT}
  !print CVIEW={CVIEW}
  !replace CVIEW ;
  !print {CVIEW}
  !new XMIN =
  !new YMIN =
  !new XMAX =
  !new YMAX =
  !new DX =
  !new DY =
  !call get XMIN 1 {CVIEW}
  !call get YMIN 2 {CVIEW}
  !call get XMAX 3 {CVIEW}
  !call get YMAX 4 {CVIEW}
  !call eval DX {XMAX}-{XMIN}
  !call eval DY {YMAX}-{YMIN}
  !call eval XMIN {XMIN}+{DX}*(1-1/({ZOOMFACTOR}))/2
  !call eval XMAX {XMAX}-{DX}*(1-1/({ZOOMFACTOR}))/2
  !call eval YMIN {YMIN}+{DY}*(1-1/({ZOOMFACTOR}))/2
  !call eval YMAX {YMAX}-{DY}*(1-1/({ZOOMFACTOR}))/2
  !print View = {XMIN};{YMIN};{XMAX};{YMAX};zoom
  View = {XMIN};{YMIN};{XMAX};{YMAX};zoom
  !delete NOECHO OUTPUT CVIEW XMIN XMAX YMIN YMAX DX DY
!end
```

The `zoom` function can now be tested with the command
`!call zoom 2`
which will cause Enif to zoom-in by a linear factor of 2, i.e. showing only one quarter of the previously displayed part of the network.

Availability of SEnC

A copy of SEnC program can be downloaded from the web site of the EMME/2 Support Center at `http://www.spiess.ch/emme2`, along with the PDF version of this paper.

In addition to binary executables for the major platforms supported by Enif (Microsoft Windows, Linux and someday hopefully Sun), the source code of SEnC will also be available. This will allow users who want to (or have to ;-) implement their own Enif clients to look at the inside details of how SEnC

has been programmed. SEnC has been written using the same programming environment which is also used for the Enif program: it is written in C++ [5] and is using TrollTech's Qt library [6].

The development of the SEnC client has only just started, so the functionality described in this paper is by no means meant as the final version. Assuming that the program will be found useful by the Enif community, the development of SEnC will continue and new features will be added to enhance its possibilities.

Note that SEnC is not part of INRO's official EMME/2 / Enif software product, but is a voluntary contribution of the author. This means that its availability does not imply any obligation on the part of INRO or the EMME/2 Support Center, nor does SEnC come with any other implicit or explicit warranty.

Conclusions

Enif's server mode provides a very powerful mechanism to access and control Enif from external client programs. The SEnC program presented in this paper implements a base that allows scripting of simple sequential repetitive Enif tasks. Since SEnC does not depend on any particular Enif functionality, but instead provides generic access to Enif's internal configuration mechanism, there are also no limits imposed by the SEnC program on what its input scripts can be used for. These limits are essentially defined by Enif itself and by the Enif expertise of the person who writes these scripts...

It is important to understand that SEnC is only one example of an Enif client. The way it is implemented, its command line interface and the command processing are by no means meant as "the" way of doing things. Different ways of adding scripting mechanisms to Enif may be more suitable for certain applications. While e.g. EMME/2's macro language is directly a part of the EMME/2 program and, thus, can not be modified or replaced by the user, Enif clients are completely independent of the Enif program. Such client can be written by anyone, using any programming language and implement whatever user interface is felt to be the best for the task to be solved.

References

[1] INRO Consultants Inc. (2003) *EMME/2 User's Manual*

[2] INRO Consultants Inc. (2003) *Enif On-Line Documentation*

[3] Spiess H. (2000) *Enif - Toward a New Interface for EMME/2.* EMME/2 Support Center, CH-2558 Aegerten, Switzerland. available at `http://www.spiess.ch/emme2`.

[4] Spiess H. (1984). *Contributions à la théorie et aux outils de planification de réseaux de transport urbain.* Ph.D. thesis, Département d'informatique et de recherche opérationnelle, Centre de recherche sur les transports, Université de Montréal, Publication 382.

[5] Stroustrup B. (1999). *The C++ Programming Language.* Third Edition. Addison-Wesley, ISBN 0201889544.

[6] TrollTech AB (2003). *QT: The Official Documentation.* New Riders Publishing, ISBN 1578702097.