FlexSim/IRFlexSim Front-end Documentation

by

Lance Geiger
5/6/2002

Problem Statement

FlexSim is a UNIX program used by the SMART Interconnects Group at USC to perform flit-level simulation of torus or mesh networks. FlexSim is based off of FlitSim, originally developed at Georgia Tech. FlexSim is currently in version 1.2. FlexSim is a great tool for studying the performance of networks with various routing algorithms, switch architectures, and message characteristics. FlexSim also supports cycle and deadlock detection. Another program, IRFlexSim is very similar to FlexSim and supports a similar set of options, however it is meant to be used to simulate irregular networks as opposed to the regular networks supported by FlexSim.

Although FlexSim is a very useful tool, its interface is rather crude, requiring network parameters to be passed via the command line. In order to make use of the program, the user must know exactly which parameters he or she wishes to vary and then must supply these parameters with proper values to the flexsim program via the command line. Most of the parameters are documented in a README file or in the recently created FlexSim manual, however, the user must read this material and be familiar with it in order to be able to use the FlexSim program at all. Another confusing aspect is that some parameters use coded values that are meaningless to any one who has not used the FlexSim program before. Things such as these result in making FlexSim unattractive to new users and an inconvenience to others.

The fact that FlexSim can be a very useful program but is hindered by its interface is the problem I will address in this report. My proposed solution is an independent graphical front end for the FlexSim program. This front end will overcome much of the interface difficulties associated with the current FlexSim program and will allow users new to FlexSim to start using the program immediately without having to read the manual or look in the README file. My project was designed to function with both the FlexSim and IRFlexSim programs. Due to their similarity, I will usually refer only to FlexSim throughout this report.

The goals of the front end interface are outlined below:

- Provide a graphical method of entering parameters
- Provide context sensitive help to users
- Replace coded values with meaningful text choices
- Provide validation to warn users of incorrect entries
- Support relationships between parameters
- Use modular design so front end can grow with FlexSim itself
- Allow multiple runs to be set up at once and run simultaneously
- Provide output processing to extract needed data

The above goals will result in FlexSim becoming a much easier to use and more powerful tool. I believe that by accomplishing these objectives, FlexSim will become a more attractive option for groups and individuals wishing to analyze or simulate networks and varying parameters. By gaining more widespread popularity, FlexSim

could become somewhat of a standard amongst the groups which use it and this would facilitate easier comparison of results and sharing of information. Even if FlexSim remains relatively unknown outside of the SMART Interconnects Group, it will still remain a valuable tool for students taking EE659 or similar courses in the future as the tool will allow the students to concentrate on what they are trying to simulate and not trying to learn how to use a program.

Evaluation Methodology

As much of my project is rather subjective, I have no way of producing hard data to verify my “findings”. It would be possible to administer a survey to users of the program and discuss the results, but given the time constraints of this project, that was not possible.

So rather than presenting methods of obtaining hard results, I will use this section to explain what areas of FlexSim improvement I concentrated in and my reasoning for doing so. In the following pages I will present my case of why I believe my implementation was optimal.

Initial Design

About half of the time spent on my project was in designing it. One of my major goals was to keep the front end interface independent of the FlexSim itself but very flexible and easy to update. The front end needed to be independent of FlexSim itself primarily so that small changes in the FlexSim source code do not cause the front end to fail. It also allows myself and future maintainers of the front end source code to concentrate solely on the front end, and not worry about the details and inner-workings of FlexSim.

I also wanted the front end to be as easy to update as possible so that when a new parameter was added to the FlexSim program or if an old parameter was modified, it would be as simple as possible to update the front end. This of course requires a data file separate from the front end source code that is read upon program launch. My goal was to make the front end completely dynamic so that almost the entire front end interface could be modified by simply making a few changes to the data file.

Keeping the above two goals in mind, I had to choose a language with which to program the interface in. I chose to use Perl/Tk, which is the standard Perl language with Tk extensions to produce GUIs in X. Although Perl is an interpreted language and can be somewhat slow, the data processing capability greatly simplifies the process of reading, writing, and parsing through large files of data, which this program would need to be capable of doing.

The first major step in my design was a proof of concept program. In its simplest form, my front end is just a graphical version of Flexsim’s command line. It presents a list of possible options and allows the user to enter in values for these options, then click a button and pass the options onto the FlexSim program for execution. I decided to create this simple version of my program, just to demonstrate to myself that Perl/Tk would work properly on USC servers and would be adequately responsive. I created a small program that displayed a window with a list of field and a button that provided the functionality described above. The data file was an XML file containing names of possible parameters, descriptions, and default values. The proof of concept program

would read this file upon launch and then dynamically display a list of fields, one field per parameter listed in the data file. This program could successfully pass the values entered by the user onto FlexSim.

I learned many important things from this first program. Most importantly, I saw that Perl/Tk was supported and did work on aludra. The interface was also reasonably quick and easy to use. I also realized that dynamically adding fields to a window based on what was read from a file can be tricky in that you have no idea how many and what sizes of fields will need to be displayed when you are writing the program. I noted that I would need to think about the layout more so that the interface would have a cleaner look and feel. I also realized that FlexSim has a large number of parameters which can make a new user's first view of the interface somewhat alarming. I decided that my interface would implement a "simple" mode which would display only a subset of commonly used parameters and an "advanced" mode which would display all of the parameters. Finally, I decided that using an XML file was probably not the best option as future maintainers of the front end may not be familiar with XML and I would not have time to write another program to assist in changing the XML data file.

The Data File

Once I had a good idea of what the requirements of the interface were, I began to concentrate on writing the data file. This data file needed to contain every possible parameter that FlexSim accepted as well as information about each parameter. This extra information included the parameter's default value, if it was used in FlexSim or IRFlexSim or both, a helpful description of how that parameter was used, rules used to verify the values for the parameter, and the parameters relation to other FlexSim parameters. I began by creating a list of all of the parameters supported by FlexSim and sorted them into related groups (with help from the manual written by Wai Hong Ho). I also noted what restrictions each parameter had as far as values and what relationships it shared with other parameters. I quickly realized that the relationships between parameters was rather complex and that "hard-coding" work-arounds for the relationships was not a good solution because new FlexSim parameters would not be able to be incorporated into the front end.

I decided to use a very flexible and elegant method to deal with this problem. My solution involved creating my own "mini programming language". Using this language, one could define the relationships between parameters by writing code that affected the values and properties of parameters based on certain conditions. A similar approach could also be used in deciding how to validate whether a parameter had been given a proper value by the end user. This is best explained through example. An excerpt of the data file is given below:

```
[GROUP Network Paramters]
#ORD      option              label                    FS IR ADV VIS CMDLINE        TYPE         DEFAULT          LIST ITEMS / VALIDATION RULE / EFFECTS / DESCRIPTION
1,        D,                  Dimension,               1, 1, 0,  1,  0,             count,       2,               ,,,The dimension of the network.  For example D=3 for a 3-dimensional network.
2,        SIZE,               Size,                    1, 1, 0,  1,  0,             numstring,   44,               ,[ALL]:length([SELF])==[Dimension],,The cardinality of each dimension
3,        FAULTS,             Faults,                  1, 1, 1,  1,  0,             whole,       0,               ,,,The number of randomly placed virtual channel faults in the network.
4,        PFAULTS,            PhysicalFaults,          1, 1, 1,  1,  0,             whole,       0,               ,,,The number of randomly placed physical channel faults in the network.


[GROUP Routing & Selection]
#ORD      option              label                    FS IR ADV VIS CMDLINE        TYPE           DEFAULT        LIST ITEMS / VALIDATION RULE / EFFECTS / DESCRIPTION
21,       SELECT,             SelectionFunc,           1, 1, 0,  0,  0,             list,          "N=Normal",    "N=Normal";"M=Min Congestion";"O=Dimension Order",,,Selection function used
24,       RADIUS,             Radius,                  1, 1, 0,  0,  0,             count,         3,             ,,,Radius:in number of hops) for exhaustive search in MB-m SW


[GROUP Deadlock Detection]
#ORD      option              label                    FS IR ADV VIS CMDLINE        TYPE         DEFAULT          LIST ITEMS / VALIDATION RULE / EFFECTS / DESCRIPTION
80,       CD,                 CycleDetection,          1, 1, 0,  1,  0,             checkbox,    0,               ,,[SELF]==1:SHOW([CDFreq]);[SELF]==0:HIDE([CDFreq]),Turns cycle detection on
81,       CDFREQ,             CDFreq,                  1, 1, 1,  0,  0,             whole,         25,            ,,,Number of cycles over which cycle detection is run
82,       DD,                 DeadlockDetection,       1, 1, 0,  1,  0,             checkbox,    0,               ,,[SELF]==1:SHOW([DDFreq]);[SELF]==0:HIDE([DDFreq]),Deadlock detection
83,       DDFREQ,             DDFreq,                  1, 1, 1,  0,  0,             whole,         100,           ,,,Number of cycles over which deadlock detection is run
```

The data file is simply a comma-delimited text file. Additional white space has been added between the commas to make the file as readable as possible. Any line beginning with a "#" is considered a comment and is ignored. All of the supported parameters are listed, one per line. The parameters are sorted into related groups. Groups are denoted using the group tag in the format of [GROUP NAME] above the list of parameters contained in that group. Groups not only help organize the data file, but help organize the user interface as well, as each group is drawn in a separate frame in the main application window.

Each line must contain 14 pieces of information, separated by commas. Blank values can be used but the commas must still be present. Each of the 14 pieces of information is explained in detail below:

ORD #: The order in which this parameter will appear in the list of options sent to the command line. This # must be unique and is primarily used to resolve conflicts between fields when FlexSim requires a particular order that parameters should be supplied in (for example, VIRTS and USEVIRTS).

option: The name of the parameter/option as it is known to FlexSim. This is the text that is actually passed to the command line so it must be correct.

label**:** The name of the parameter/option as it should be displayed to the user. In most cases this will be the same or similar to the "option" field but allows more flexibility, for example, using "Dimension" is more clear to the user than using "D". This field must be unique across all parameters.

FS: Set to "1" if this option is supported by FlexSim or "0" otherwise

IR: Set to "1" if this option is supported by IRFlexSIm or "0" otherwise

ADV: Set to "1" if this option is advanced, or "0" if it should be displayed in simple mode

VIS: Set to "1" if this option should be visible initially or "0" otherwise

CMDLINE: Set to "1" if this option should be passed to the command line no matter what. Set to "2" if this option should NEVER be sent to the command line. Most options will have this set to "0" which will use the VIS option to determine whether or not to send it to the command line.

TYPE: The type of data this option should expect. One of count (integer>0), whole (integer>=0), integer, real, checkbox, pow2 (integer power of 2), filename, list, numstring (list of digits). In the case of checkbox and list, this determines how the option will be displayed to the user. Also specifies how the option will be validated and determines what kind of values the user can supply to this option.

DEFAULT: The default value of this option

LIST ITEMS: A semicolon delimited list of value=name pairs that should be displayed in a drop down list. Must be contained in quotes since many options contain spaces and special characters.

VALIDATION RULE: A semicolon delimited list of Conditions and Rules that are used to verify whether the value contained in this field is valid

EFFECTS: A semicolon delimited list of Conditions and Effects to be applied to satisfy the relationships this option has with other options

DESCRIPTION: A free-form text description of this option and what it represents with (optionally) examples of how to use it. This is the text string displayed for context sensitive help when the user clicks the help button next to a field.

Although somewhat confusing at first, the format of the data file is relatively simple and extremely flexible. To add a new option to the front end, one only needs to add another line to the data file and the next time the front end is launched, the new option will be read and displayed automatically. The first three fields are straightforward: pick a new unique number, the name of the FlexSim option, and a slightly more descriptive name for that option which will be shown to the user.

The next set of fields is also simple. For options that are supported by both FlexSim and IRFlexsim, the FS and IR fields get a 1. For options that are only in IRFlexSim, FS and IR get a 0 and 1 respectively. If the option is commonly used, it should be visible in “simple” mode, so ADV should be 0. If the option is rarely used it should be visible only in “advanced” mode so ADV should be 1. When in advanced mode, both simple and advanced options are displayed.

Next, the CMDLINE option is usually “0” for most options. The exceptions occur when you create “ghost” fields such as “Load Rate” that FlexSim doesn’t actually support. Ghost fields are helpful because it allows the user to enter values for additional options that then affect other options, however, FlexSim would not understand these ghost options so they must not be passed to the compiler and thus CMDLINE=2 for these options. In some case you wish for a particular option to be passed to the compiler even if it is not visible, in this case CMDLINE=1.

Next, the TYPE field specified which type of option this is. If it is a toggle or yes/no option a check box should be used. If the option has a set of possible values, a list should be used. If the option is a string of digits (such as the SIZE field) a numstring should be used. All other types of options should use the type that best matches what the option represents. See above for an explanation of the other available types.

The DEFAULT field simply specifies the initial value the option should contain. If the end user hits the run button without changing the option, this will be the value passed to the compiler.

LIST ITEMS contains a list of possible choices for options that have a small set of values that are acceptable. Lists should be used in place of “coded” options such as DIST and PROTO. A list of text options is much more intuitive than numeric or letter codes.

DESCRIPTION is simply the text displayed when the user clicks the help button next to the option. The more detailed this text, the more helpful it is to the user.

The Mini Programming Language

In order to support the flexibility needed in defining relationships among options and validation rules, a simple programming language was needed. The format of both the VALIDATION RULES and the EFFECTS is the follow:

condition1:effect1a:effect2a;condtion2:effect2a;condition3:effect3a:effect3b:effect3c

The validation rule or the effect is a semicolon delimited list of strings. Each string is then delimited into pieces by colons. The first piece of every string is the condition. If this condition evaluates to TRUE, then the commands in the remainder of the string are evaluated, otherwise they are skipped. This process continues until the end of the last string is reached. Each command is a simple comparison or an assignment. The following operators are supported:

+ - < > <= >= == != eq ne

where “eq” and “ne” are the string comparison equivalents of == and !=

In addition to the operators, the following functions are supported as well:

length(string)	returns the length of string
SHOW([NAME])	makes the field NAME visible
HIDE([NAME])	makes the field NAME invisible
ENABLE([NAME])	sets cmdline=1 forcing this option to be included in command line
DISABLE([NAME])	sets cmdline=2 forcing this option to NOT be included in command line
POW2([NUM])	returns the closest power of 2 >= NUM
[~DIAMETER]	returns the diameter of the network
[~NUMLINKS]	returns the number of physical channels used by the network
[~NUMNODES]	returns the number of nodes in the network

In addition, the current values of all options are available for use in the commands. The value of an option can be referenced with the following format: [NAME] where NAME is the label of the option that is desired. Another useful feature is the [ALL] condition which always evaluates to TRUE. Finally, [SELF] will always reference the option for which the command applies to – useful if the option name is ever changed – then the rules do not need to be updated.

A few examples of using the mini language are presented next.

Example 1: BUFFERS Validation rule

The BUFFERS option represents the buffer depth on each input virtual channel. It must be greater than 1. The closest TYPE that matches this option is the “count” type, but this type only guarantees than an integer greater than 0 is entered. Since BUFFERS cannot be 1, an extra validation rule must be supplied:

[ALL]:[SELF]>1

The first string (before the colon) is the condition. Since [ALL] is used, this condition always evaluates to true which means this validation rule will always be checked. Conditions other than [ALL] can be used, for example, if you have two options and either one could be 0, but not both – in

this case you would have a validation rule conditioned on the value of the second option. The second string (after the colon) in the above example is the actual validation rule. [SELF] gets replaced with the actual value input by the user. For example, let's assume it is 5. The rule now looks like "5 > 1". This rule is then evaluated. Since it evaluates to TRUE, no error is displayed. If it had evaluated to false, an error message box would pop up informing the user which rule had been violated.

Example 2: SIZE Validation rule

The SIZE parameter is an odd option – it is a string of digits where each digit specifies the cardinality of a dimension in the network. It is related to the option Dimension in that SIZE should be exactly "Dimension" digits long, otherwise some error has occurred. To define this validation rule, one would use:

```
[ALL]:length([SELF])==[Dimension]
```

Once again the condition (first string before the colon) is [ALL] meaning that this validation rule is always checked. The second string is the rule. On the left side, [SELF] will be replaced with the current value of SIZE, for example let's say it is "424". [Dimension] will be replaced with the current value of the Dimension option, for example 3. Thus the rule would now look like "length('424')==3". The length() function would then be invoked and would return the length of the string, in this case 3. We would then have "3==3" which evaluates to TRUE, so no error is generated.

Example 3: DeadlockDetection Effect

FlexSim uses a pair of deadlock parameters: DD and DDFREQ. DD stands for DeadlockDetection and is a yes/no value representing whether or not deadlock detection will be run during the simulation. DDFREQ represents how often this detection is run. DDFREQ is meaningless if DD=0 (deadlock detection is off), therefore we want to hide the DDFREQ option whenever DD=0 and we want to show the DDFREQ option whenever DD=1. This can be accomplished with the following effects:

```
[SELF]==1:SHOW([DDFreq]);[SELF]==0:HIDE([DDFreq])
```

Here we actually have 2 effects, separated by a semicolon. The first effect has the condition [SELF]==1. In this case [SELF] refers to DD and if DD=1, then the command SHOW([DDFreq]) will be executed. The SHOW() function simply sets the visible flag of the desired option and will display that option the next time the window is refreshed. We have the complementary effect next – when DD=0 we need to hide the DDFREQ option. Please note that these commands are always executed in order, and if multiple conditions are satisfied, multiple effects can be applied at once.

Example 4: DIST and the ghost field X

The DIST parameters was one of the trickiest to implement. Since DIST can specify any one of a set of communication patterns, FlexSim uses numeric codes to represent each communication pattern. These codes are replaced with text string in the front end to make it easier on the user. However, DIST also supports numeric values greater than 0. So in this case we have a kind of dual-field. It is impossible to have both a list box and text field for the same option so the solution is to use a ghost field. A ghost field is just like any other option listed in the data file, however, the ghost option is

purely for the front end; it is never sent to the command line, therefore it's CMDLINE is always 2. The solution to the DIST problem that I came up with was the following: create two options, one named DIST and another named X. Both options point to the same FlexSim parameter DIST, but they have different labels and different ORD #'s. Now, whenever to user selects "random within X hops" as the communication pattern, the "X" option will appear and disable the list box. If another communication pattern is selected, the "X" option will disappear again and the list box will be reenabled. The "effects" to accomplish this are given below:

```
[ALL]:HIDE([X]):ENABLE([Distance]):DISABLE([X]);
[SELF] eq "X=random within X hops":SHOW([X]):ENABLE([X]):DISABLE([Distance])
```

There are 2 effects used. The first has the condition [ALL] meaning it is always executed. The command it performs hides X, enables Distance (the list box) and disables X. The second effect has the condition [SELF] eq "X=random within X hops" so when the user chooses this value from the list box, the X option is displayed, enabled, and the list box is disabled. When an option is disabled, it is still visible, but it will not pass on its value to the command line. Since we have two options that both point to the same command line option we can only have on enabled at a time or otherwise FlexSim would get passed two different parameters for DIST which could cause an error.

New Option Walkthough: LoadRate

This final example is a walkthrough on how to add a new option to FlexSim. One useful parameter to specify when simulating a network is the load rate. However, FlexSim does not support such an option. In order to achieve a particular load rate, the user must calculate the value of PER needed to approximately create the desired load rate using the formula (once again, thanks to Wai Hong Ho)

```
PER = # nodes * MSGL * Avg Dist / LoadRate / # channels
```

In order to support this new option I had to create a few new special variables, but I believe this new variables will be helpful with other new options as well. The variables are:

[~DIAMETER]	returns the diameter of the network
[~NUMLINKS]	returns the number of physical channels used by the network
[~NUMNODES]	returns the number of nodes in the network

Thus the PER equation now becomes:

```
[ALL]:[Per]~=[~NUMNODES]*[MessageLen]*[~DIAMETER]/2/[SELF]/[UseVirts]/[~NUMLINKS]
```

Here the average distance is approximated with Diameter/2. Note the "~=" operator. It is the assignment operator and will assign the result of the expression on the right had side to the Per option. Since the LoadRate is a percent, a validation rule must be created to make sure the user does not enter a value greater than 1 or less than 0, like the following:

```
[ALL]:[SELF]>=0:[SELF]<=1
```

Now that the hard part is completed, filling in the rest of the fields for LoadRate is easy:

#ORD:	the next available number, for example 104
option:	since this is a ghost option this value can be anything such as "DUMMY"

label:	this is of course “LoadRate”
FS:	this should be “1” since it applies to FlexSim
IR:	this should be “1” since it applies to IRFlexSim as well
ADV:	this should be “0” since it would be a commonly used option
VIS:	this must be “1” since it should be visible initially
CMDLINE:	this must be “2” so it is never passed to FlexSim
TYPE:	this must be “real” since non-integer numbers must be entered
DEFAULT:	this can be anything, for example 0.05
LIST ITEMS:	blank since this is not a list
VALIDATION:	explained above
EFFECT:	explained above
DESCRIPTION:	“The relative load rate on the links of the network.”

Now take all of these fields and put them on one line, separated by commas, then insert it into the data file. The new option will be integrated into the front end automatically.

Program Flow

Upon launch, the program assumes some default parameters such as the location of the data file. The program then attempts to read the data file and if it is not successful, it waits for the user to specify the location of the data file by choosing “Config …” from the File menu. Once the data file has been successfully loaded, it is parsed and the groups are extracted along with every option and its associated information. The main window is then created and all of the options are drawn.

Each group is drawn in its own frame and the frames are arranged in the main window. Once all of the options have been displayed the program enters the event loop and awaits user interaction. Clicking a list box or checkbox will refresh the window and process the effects and validation rules. Changing the text in an entry box will not refresh the window, however the window can be refreshed manually by clicking on the “Refresh” button.

The top of the window contains several list boxes. The first selects whether the user wishes to run FlexSim or IRFlexSim. The next chooses between simple mode and advanced mode. Finally the last listbox chooses between normal execution and batch mode.

While in normal mode, clicking the Run button will exit the application and pass the command line to FlexSim and run it. When in batch mode, clicking the Run button will add the command line to the batch list without exiting the application. The user can continue to add multiple command lines to the list. When the user is done, the “Begin” button can be clicked and all of the command lines in the batch list will be submitted.

Validation rules are check and effects applied during a window refresh. If any of the validation rules fail, a window will popup informing the user which option and rule was violated. Help can be displayed for any option by clicking on the “?” button next to the option of interest.

The source code for the program is contained in a single file (fsgui.pl) and is thouroughly commented.

Performance Results

I was able to achieve almost all of my objectives and I did achieve the most important ones. I have designed and implemented a simple yet flexible front end for the FlexSim network simulator that makes the program much easier and more efficient to use. In summary, I achieved the following:

- Created a graphical method for entering parameters
- Program provides context sensitive help to users
- Replaced coded values with meaningful text choices
- Program provides validation to warn users of incorrect entries
- Complex relationships between parameters are supported
- The front end is as flexible as possible
- Simple batch capability was added to the front end

The only major goal that I did not achieve was the output processing part. I decided that it would be too difficult for the front end itself to do the output processing and that a second, independent program would be necessary to process the output files after they had been created. Unfortunately I did not have time to implement a program such as this.

As far as what I did achieve, I have laid the foundation for a front end to FlexSim that is fully functional and easy to use. The are many things that could be improved (discussed in the next section), but it should not be that difficult for another student to take up that task at a later time.

Discussion of Results

I encountered several major obstacles during my project that prevented me from achieving everything I had wished to accomplish with this project. The primary problem was the USC servers. Although my initial proof of concept program demonstrated that Perl/TK is supported on USC servers, many of the more advanced user interface elements are not supported or are very unstable. I did not discover this until late in the development and it was much too late to switch programming languages so I had to spend a significant amount of time figuring out how to work around the problems related to common Perl/Tk components not being supported. This resulted in wasted time and a front end that does not look as crisp and fluid as possible.

Having completed this project, it is my opinion that the front end and FlexSim itself could be ported to another platform such as Windows where it could possibly run much faster. Although my front end greatly increases the efficiency with which FlexSim can be used, it does nothing to improve the speed of Flexsim itself, which is very slow, especially during the end of the semester. I had many difficulties getting results due to the slow speed of FlexSim and the USC servers being overloaded.

Despite these problems I was able to produce a working product that I am decently happy with. I was able to use the final version of my front end to redo homework #3 and it was dramatically easier than before. Hopefully future students of EE659 will be able to use this for their homework as well.

The front end has no known major bugs as of right now but there are several areas it could be improved in, including:

- Clean up interface, add colors
- Add FlexSim Manual to front end as a help option
- Write output processing programs
- Display transient statistics in front end as FlexSim is running
- Add more options to batch runs such as parameter sweeps and the ability to queue up 100 runs but only allow 10 to run at a time
- Add more options such as “LoadRate” to make it easier on users