

## Cover and Contents Page

---



How to use

**FROGS**

**Freely available Object-oriented General purpose simulator**

### Revision History

Revision No.	Date	Prepared By with E-mail Id
1.0	Dec 31 2000.	<a href="mailto:Vasanth.varathan@wipro.com">Vasanth.varathan@wipro.com</a>

Authorised By:

Signature:

Name:

Date:

## Contents

1.Introduction to FROGS.....	4
1.1Simulation Executive (SIMEX).....	4

*How to use FROGS.*

1.2Statistical Objects (STATOBJ).....	4
1.3Configuration Class & TCL interface.....	4
2.General information on how FROGS actually works.....	5
3.FROGS features for creating a simulation system.....	5
Simulation Executive library.....	6
Figure 1: Simulation Executive's inheritance diagram.....	6
SxObject.....	6
SxInfo.....	6
SxDaemon.....	7
SxEvent.....	7
Figure 2: State Transition diagram for SxEvent.....	7
SxFlag.....	8
Figure 3: State Transition diagram for SxFlag.....	8
SxQueue.....	8
Figure 4: State Transition diagram for SxQueue.....	9
SxResource.....	9
Figure 5: State Transition diagram for SxResource.....	10
SxSource.....	10
SxTCP.....	10
SxTimer.....	10
Figure 6: State Transition Diagram for SxTimer.....	11
SxTrigger.....	11
SxThread.....	11
Figure 7: State Transition Diagram for SxThread.....	13
SxManager.....	14
SxMonitor.....	15
SxScheduler.....	15
SxTimed.....	15
SxListener.....	15
Statistical Class library.....	16
Statistical Class hierarchy Diagram.....	16
.....	16
Numerical law classes inheritance diagram.....	16
StObject.....	17
StCounter.....	18
StIntegrator.....	18
StHistogram.....	18
StScaler.....	19
StTimeScaler.....	19
StObjectScaler.....	19
StTimeGraph.....	19
StStateDiagram.....	19
StProto.....	20
StNumericLaw.....	20
4.Steps to create a simple simulation system with sample files.....	20
4.1Configuration file.....	20
4.2Simulation Model file.....	21
4.3Make file.....	21
4.4How to run the simulation through GUI.....	23
4.5How to run the simulation in command line.....	23
4.6Environment settings for libraries.....	23
5.Points to remember while creating simulation.....	24
5.1Co-operative Scheduling.....	24
5.2Incrementing simulation clock (through delay).....	24
5.3Instantiating statistical objects with appropriate construction parameter.....	24
5.4Defining and changing states of simulation system objects.....	24
5.5Exporting the simulation system objects to display front end.....	26
6.Unexplored areas.....	26
6.1Script Attribute.....	26
6.2Separation of ISE from kernel.....	27
6.3Other means to increment the simulation clock.....	28
.....6.4 Possibility of an application outside FROGS environment to communicate with model instances.	

.....28

## List of Figures

Figure 1: Simulation Executive's inheritance diagram.....	6
Figure 2: State Transition diagram for SxEvent.....	7
Figure 3: State Transition diagram for SxFlag.....	8
Figure 4: State Transition diagram for SxQueue.....	9
Figure 5: State Transition diagram for SxResource.....	10
Figure 6: State Transition Diagram for SxTimer.....	11
Figure 7: State Transition Diagram for SxThread.....	13

## 1. Introduction to FROGS

It is an object oriented general-purpose simulation tool using which one could evaluate complex architectures. It can be used interactively either as a tool for aiding design & tuning of time-constrained systems or as decision aid tool for adjusting the behaviour of these systems in production. FROGS is event-driven simulator based on unique, internally maintained time reference, which is conceptually continuous & totally independent from the host workstation's idea of time.

FROGS consists of SIMEX or the simulation executive & STATOBJ or the statistical objects. It also consists of configuration class in the SIMEX, which enables one to define attribute values through a TCL/TK-based GUI. This enables one to change the simulation settings for a given configuration without having to rebuild the simulator. The ISE co-ordinates the entire process of configuring the interface, defining events sources, defining the architecture attributes etc.

### 1.1 Simulation Executive (SIMEX)

This forms the core of simulation system. It provides rich set of C++ objects for system modelling, such as threads, synchronisation objects, interface to measurement tools, sources, events & few data structures like queues.

To use any of the facilities provided by the above mentioned classes, all that one need to do is, inherit these objects & implement the body method to perform appropriate actions as intended.

### 1.2 Statistical Objects (STATOBJ)

This gives a collection of C++ classes, which performs different kinds of statistical measurements. To perform statistical analysis on one's system objects, one has to instantiate the display front-end objects like histogram or time curve & pass the calculated statistical data from different statistical objects using the appropriate methods provided. All the statistical objects can define the sampling interval according to which they should collect data from system objects.

Some of the statistical objects provided are counter, integrator, histogram, scaler, numerical law & time curves. A set of compatible objects could be grouped to form a statistical group object to form group computation.

### 1.3 Configuration Class & TCL interface

Iterating through the design decisions by setting different parameters for the simulation system to find out its influence on the systems performance & functionality is basic expectation from a simulator. To do this without having to re-build the simulator is what

one would feel handy. This configuration class purpose is to satisfy this requirement. This configuration class abstraction provides a set of attributes, which provide dynamic support to the GUI for displaying automatically the configuration windows containing the model parameters. These collected values are finally stored into configuration database. The tuneable attribute of your simulation system must be defined in the model library using this configuration class interface.

## **2. General information on how FROGS actually works**

In FROGS the simulation model describes the behaviour of given activity. The Architecture associates various elements of a simulation system to define its initialisation rules. The librarian must explicitly load the model libraries before they can be used. The main function of the model library is to export definitions of simulation models, along with their configuration characteristics. The Frogs librarian's role is to offer GUI for creating & configuring the chosen instances according to required characteristics of behaviour. The ISE also provides features for setting behavioural parameters.

FROGS expects the model or your simulation system code implementing simulation system objects, resource definition for instantiation using GUI with tuneable system parameter (if present) in the implementation of class inherited from configuration class. It should also implement that instantiate method populating the simulation system with system object instances based on the parameters collected in the configuration database. It also expects a typical boot strap code which is the same for any simulation system to create an executable which is used to bootstrap the simulation.

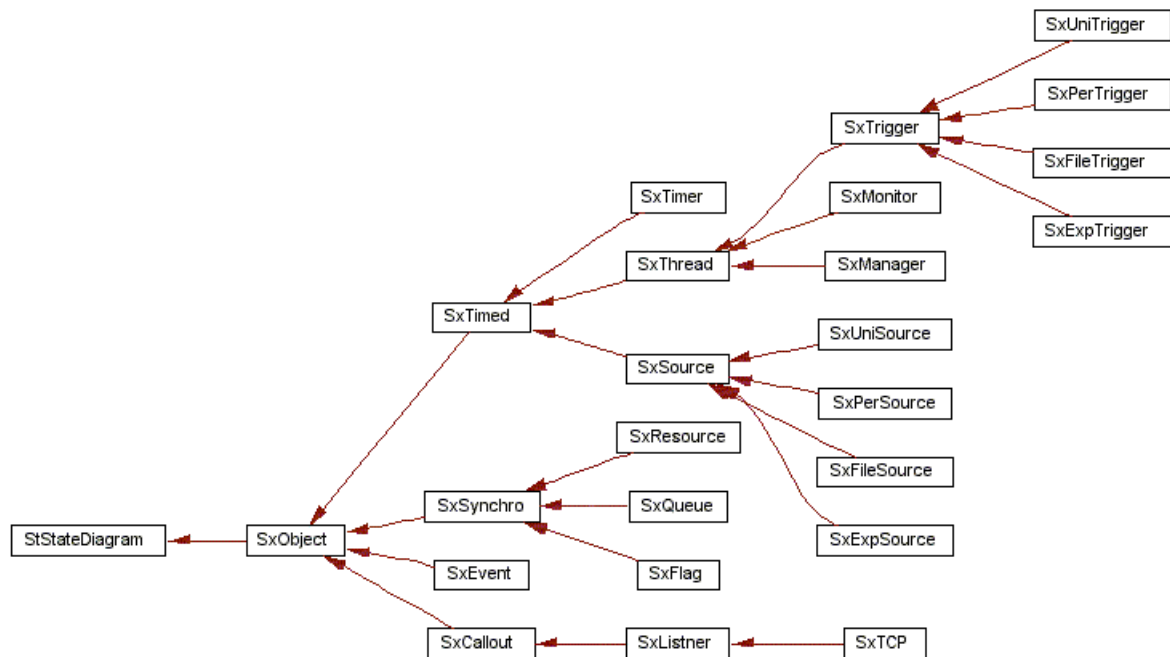
First the obscan tool parses the configuration support code, which is part of obstick package. This parsing result in the creation of Database implementation file, containing methods to support the persistence of instances from the configuration classes. Each instance holds a list of configuration attributes, which describes its settings. The memory layout of persistent classes found while parsing configuration class file are determined & saved in a repository named filename.pcr.

## **3. FROGS features for creating a simulation system**

FROGS provides a set of simulation kernel facilities to implement the simulation system. It also provides a set of statistical analysis tools to analyse the simulation systems output. Some of the important features are explained below. For detailed notes refer to FROGS programmers manual SIMEX and STATOBJ.

## Simulation Executive library

Figure 1: Simulation Executive's inheritance diagram



### SxObject

This is the base class for most of the simulation executive's simulation object classes. It implements basic simulation object, which can take multiple programmable states the simulation monitor can export to a display front-end upon transition from one state to another. Inherited statistical object's state diagram class performs state transition logging. This class object can hold a list of state event object instances that will be signalled when significant state transition occurs at the system object level. It provides methods to set state of the simulation object, associate a state event with the simulation object, and define states for the system object and also methods to manipulate its state index.

### SxInfo

This class is the super class for information message classes. Messages are usually generated by simulation source & stored into queue pending by threads. One could inherit this class & implement their own message class & add data members as per requirement. The basic information message class has the following attributes

- Priority level, used to order the messages when they are inserted in prioritised queues;
- A traffic identifier, which can be used to discriminate message categories when

performing statistical measurement on queue throughputs.

- A message generation time-stamp.

Methods to copy (i.e. copy all data members including the time-stamp), clone (copy the details except for the time-stamp information, re-setting it to current system clock time), free a message from queue, get & set traffic attributes are provided.

### **SxDaemon**

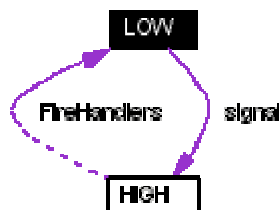
This is a super class implementing local asynchronous actions, which may occur upon state transitions or signals to system objects (i.e. any simulation object SxObject). Multiple daemons can be linked together to create an activation list. This class should be sub-classed to implement the body method, which is expected to perform the required actions. The priority value supplied during its creation determines the firing order of multiple daemons linked together in a given activation list. Methods to add a daemon, remove a daemon, fire or activate a daemon's body procedure are provided. There is also a method called "is linked" to check if a particular daemon is linked in the activation list.

### **SxEvent**

This class implements, the basic event object used to signal elementary actions. Multiple events can be linked together to create a signalling list. When an event is signalled, all the events in the signalling list which it heading are signalled in turn.

The event object on its creation takes a pointer to a handler, which is a daemon object that will be fired each time the event's internal state switches from low to high. Event objects can also be linked to form an event list. It provides methods to add and remove handlers to event objects. There are also methods to check if an event is currently armed with a particular handler. One could link, signal & set trace level for an event object using appropriate methods.

**Figure 2: State Transition diagram for SxEvent**



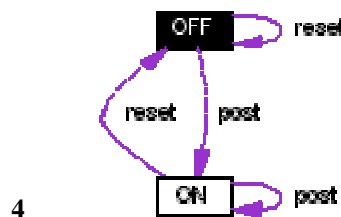
This class extends the functionality of an event object & specialises it to signal

state changes from simulation object instances. SxObject implementation causes the events associated with the system objects to be signalled after each significant state transition. The state event object must go through the off & on states alternatively for the event processing handlers to be fired. The on & off states are encoded into the object during its creation time.

### **SxFlag**

This class implements a synchronisation object, which allows a set of threads to wait for a condition. A simulation flag has two possible states, ON meaning the condition is satisfied & OFF meaning the condition is not satisfied. The initial state of the flag is set during its creation as a parameter to the constructor. Its default state is OFF. Methods like pend, post, set-on, set-off & reset are provide to perform the necessary operations on flag object.

**Figure 3: State Transition diagram for SxFlag**



### **SxQueue**

This class implements a thread synchronisation object allowing threads to wait for messages. Messages are prioritised, and must be subclasses of the SxInfo super class. The queue can operate in any of the following priority mode FIFO, LIFO, PRUP – thread having the lowest priority is always served first. When two threads have the same priority, the oldest pending one is chosen. In PRUPLF the lowest priority thread is server first. When two threads have same priority, the latest one is chosen. Similarly PRDN & PRDNLF exists where the highest priority thread is always served first. It provides methods to do post a message and get a message from the queue. It also provides methods to do some statistical analysis on queue throughput, retrieve the average number of messages over time and maximum / minimum number of messages in queue etc.

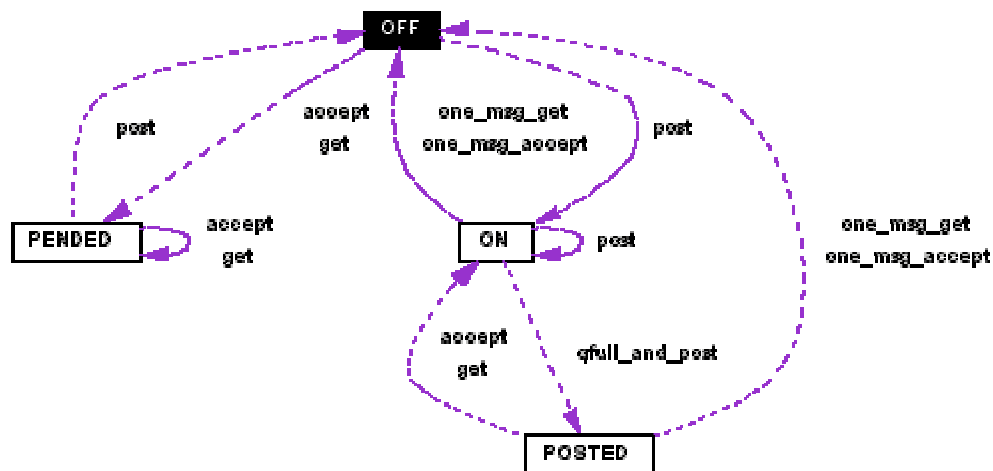
The instance of this class can take four active states

1. PENDED indicates that at least one thread is pending for messages on input. This state denotes an empty queue.



2. OFF means that the queue is idle, with neither messages stored, nor pending consumer threads.
3. ON means that at least one message is available to consumer threads, but no output contention exists.
4. POSTED means that an output contention currently exists on the queue, which needs to dispatch available messages before accepting further posting from suspended producer threads.

Figure 4: State Transition diagram for SxQueue



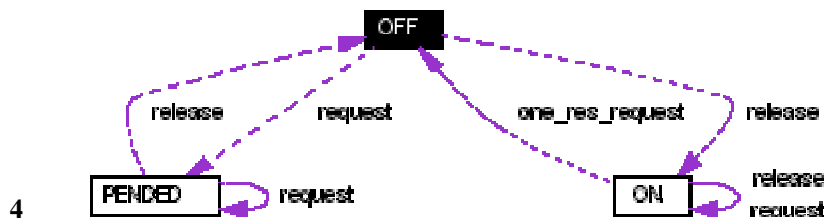
### SxResource

This class implements a thread synchronisation object, which would act like a semaphore. Thread execution can be serialised using this object, by allowing limited number of resources to be dispatched to them. The lack of resource causes the requesting thread to pend on the object until one becomes available. The three active states of the resource objects are

1. PENDED indicating that at least one thread is waiting for a resource. This state denotes that no resource unit is currently available.
2. OFF means that the resource is idle, with neither units, nor pending threads.
3. ON means that at least one resource unit is available for the threads.

The priority modes available in the queue apply to this object also.

Figure 5: State Transition diagram for SxResource



### SxSource

This class is the super class of message generating objects. The behaviour of a source consists in producing messages at specific times determined by a private generation law. Message types must be subclasses of the SxInfo super class. Once generated message is automatically posted to a destination queue. A pre-define set of typical subclassed sources is available with the simulation executive, including periodic source, exponential source, uniform source, and file-based sources. The type of message that should be generated is determined by the message template, which is supplied during the creation of source object. The source object is active only between the given time bounds specified by time start & time end during its creation. The virtual method clone will be invoked to obtain a cloned instance of the template each time it is necessary. The source object as part of its destructor actions deletes the message template, so the required way of initialising a source is to create a new template object each time a new source is built.

### SxTCP

This class implements a communication object above the socket interface, managing a bi-directional TCP/IP channel. This class only provides the needed support for monitoring through SxMonitor object. This class being the subclass of SxListener super-class has the capability of waiting for asynchronous input events on the TCP/IP channel. Messages sent and received through a SxTcp instances are made of a message type identifier followed by an optional block of dynamically sized unstructured data. The TCP channel can be created in either server or client mode by passing appropriate parameter during construction. The usual socket functions like connect, bind, accept, send, receive and poll are available for working with this object.

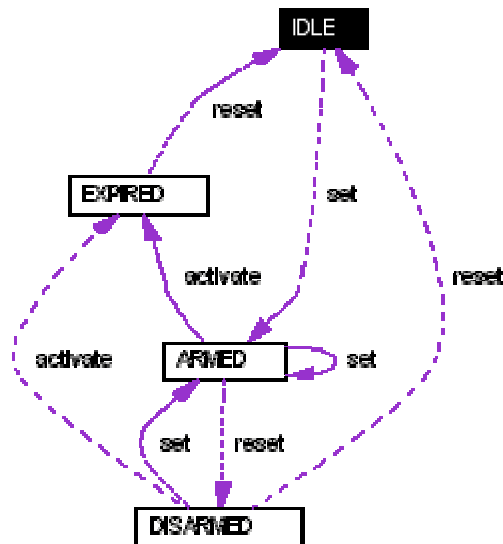
### SxTimer

The main purpose of this class is to perform specific actions at predefined simulation times. A timer class basically takes three active states as stated below

1. IDLE denotes idle timer object

2. ARMED indicates that an expiration time currently exists for the object.
3. DISARMED indicates that a request to disarm the previously armed timer has been issued.
4. EXPIRED denotes expired timer.

**Figure 6: State Transition Diagram for SxTimer**



Methods like set time, reset time, activate are provided for manipulating this object.

### **SxTrigger**

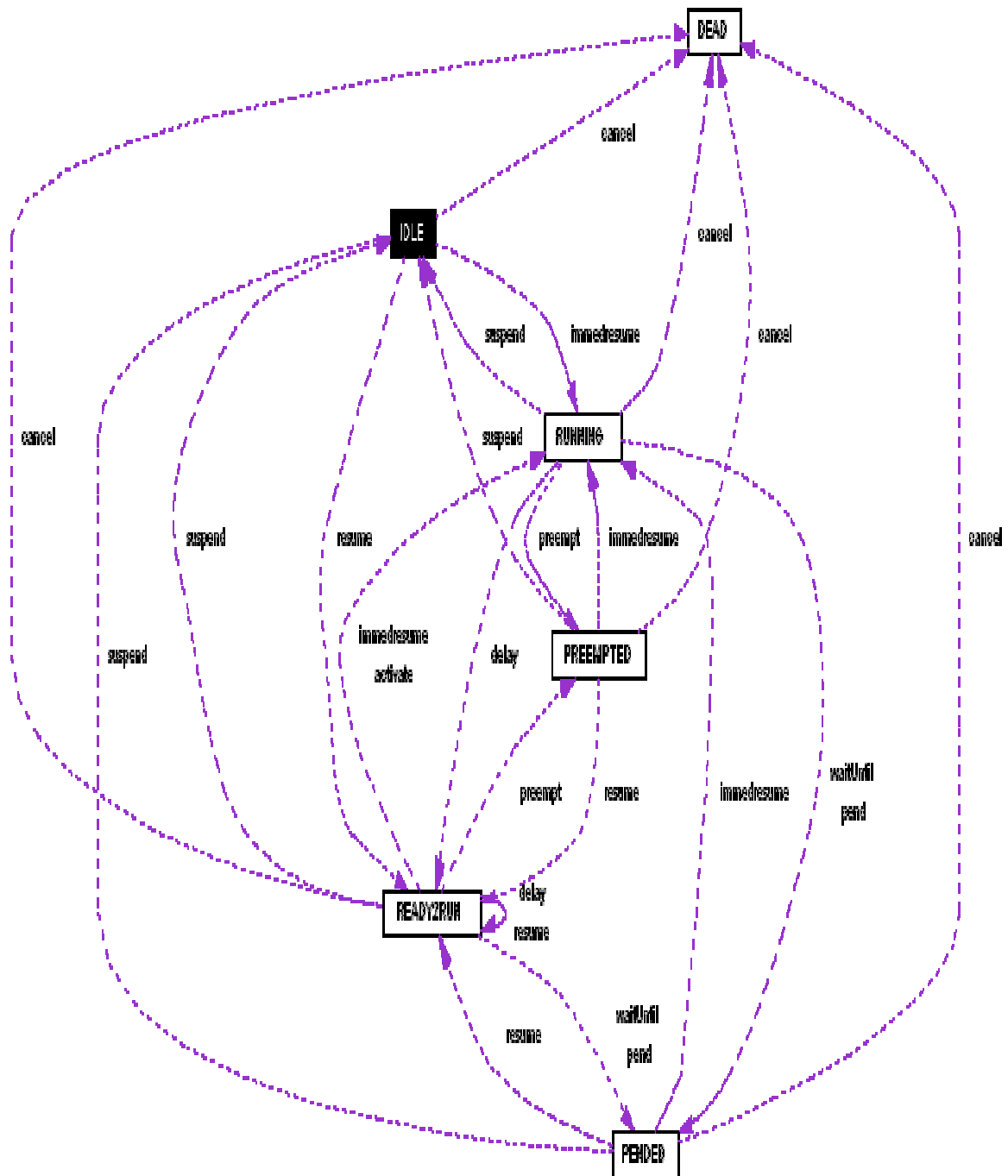
Triggers are threads driven by event sources to start a client handler. The handler is fired according to the event source object. This class cannot be used "as is", but can be extended by specific trigger body implementation. Method to check the validity of the trigger is also provided.

### **SxThread**

This class implements the simulation executive's thread object behaviour. Threads are concurrent activities inside the simulation system having a private execution stack while sharing the global address space of the host application. As timed objects, threads are prioritised objects scheduled according to their requested activation time. The list of threads that are ready to run is maintained in run chain, which is unique in the system. The run chain is an instance of scheduler class. Control of the CPU is always given to one simulation executive's thread at any time during the simulator's lifetime. The executing thread changes only whenever it issues a suspensive call, yielding control to other. The static pointer current thread has the pointer to the currently executing thread object. Threads have four basic states:

1. IDLE referring to unconditionally suspended state. The thread is removed from the run chain, and will not regain the CPU until it is explicitly resumed.
2. PENDED indicates the thread is waiting for the condition of synchronisation object to be met. Here also the thread is removed from the run chain and will not regain the CPU until explicitly resumed.
3. PREEMPTED indicates that the thread has been explicitly pre-empted by another one. The thread is removed from the run chain, and will not regain the CPU until explicitly resumed.
4. READY to RUN denotes that the thread is part of the run chain and is waiting to be chosen for execution by the scheduler.
5. RUNNING denotes a thread currently owning the CPU.

Figure 7: State Transition Diagram for SxThread



Different methods provided by this class are as follows

1. Set context-switching mode – This method enables one to make context switching either in conservative mode or normal mode. The conservative mode ensures that the entire context of the suspended thread is saved and restored when a switch occurs; it is slower than the light switch is but is safer under certain circumstances. Technically, the conservative mode differs from the faster one by calling the sigsetjmp/siglongjmp pair to save/restore the context instead of the sigjmp/longjmp.

2. Resume, This resumes the thread object. It takes an optional synchronisation object as parameter, which could have invoked this call. The actions taken by this call depends on the target thread state:
  - If previously pre-empted, the thread is put into run chain. If the thread was pre-empted while running the delay time remainder is renewed.
  - If the thread was ready to run the thread activation time is set to current clock time.
  - If the thread was idle or pending, the thread is inserted into the run chain and enters ready to run state.
3. Prioritise, This method takes an increment as a parameter and adds it to the base priority of the thread. Then the run chain is re-ordered to reflect the change, but does not pre-empt the executing thread in any case.
4. Renice, This invokes prioritise method with the increment that comes in as parameter, then it attempts to pre-empt the executing thread if a priority thread is leading the run chain as a result of the operation.
5. There are several thread synchronisation methods like wait until, wait or until, wait or etc. The working of them should be evident from the state transition diagram that is given above.
6. Methods to save the stack context, check stack over flow, get the stack size and stack top are also provided.
7. Methods like activate, suspend, immediate resume, pre-empt and delay manipulate the execution of the thread body method implementation.

The following classes are rarely inherited and implemented unless a low level kernel operation is required. It is advised not to inherit and instantiate the following class objects more than once.

### ***SxManager***

This class implements a set of services aimed at controlling the simulation initialisation and termination. The simulation manager should be created during start of simulation. It creates a set of simulation management objects like simulation monitor when interactive mode is in effect. This monitor runs as a separate thread in the simulation system, and establishes a bi-directional communication link between the GUI application displaying simulation data and the active objects that are monitored in the system. This class should be instantiated only once

during the simulation lifetime. The main purpose of the simulation manager thread is to perform global sampling of all declared statistical objects, if no such object is available it simply waits till the simulation ends. Some of the attributes in the simulation manager class are execution time, warm-up time, finish time, number of samples, sampling period. It also has flags to indicate if it is monitored, running, infinite execution time etc.

### ***SxMonitor***

This class inherits the properties of SxThread; it implements a set of services aimed at providing monitoring capabilities. Monitoring refers to the ability to interact with the simulation objects from an external application during the simulation lifetime. This monitor runs as a separate thread in simulation; primarily waiting for commands from the external (usually GUI) application, listening to a TCP socket for input. This is created only when the simulation runs in an interactive mode. It is also in charge of holding and releasing the simulation process as requested by the display front-end. The monitor uses the callout management service from the scheduler class to register itself for asynchronous input notification from the communication channel.

### ***SxScheduler***

The simulation executive's scheduler is in charge of managing a set of runnable time-related objects, maintaining their respective activation order. Timed objects are first ordered by scheduled activation time, then by decreasing priority. The highest priority thread is chosen for activation. The current time reference used by this class is the contents of the SxClock variable. The thread manager's run chain is an instance of this class.

### ***SxTimed***

This pure class is the super class of objects having behaviour related to the simulated time, such as threads and sources. The scheduler object manages all simulation system objects that are inherited from this class. This classes object takes two states one being IDLE & the other RUNNING. IDLE refers to a state where the object is not a member of the scheduler's activation list. RUNNING refers otherwise; denoting an object is scheduled to run. This class should rarely be inherited directly by the user-level classes, but is rather used by the low-level simulation executive's kernel code. This object provides methods to set time, get time, insert into activation list, delay the current by specified time, suspend the object, resume the object and activate the object.

### ***SxListener***

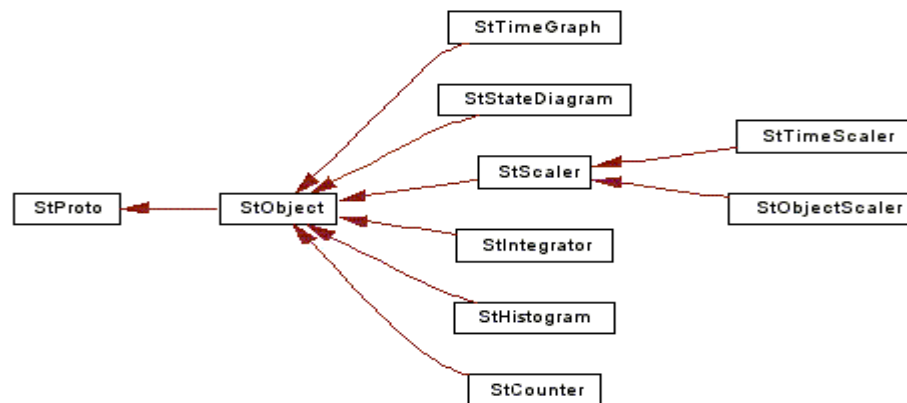
This class implements file descriptor-monitoring object, used by threads to detect input conditions pending on input channel synchronously or asynchronously, while the other threads are allowed to run concurrently. The simulation executive implements multi-threading on behalf

of the co-routines, not native threads from the operating system. This is relevant with the idea of time independence, which is needed to have an event-driven scheduling kernel, which guarantees reproducible behaviour of threads across simulation sessions. Hence it is obvious that issuing indeterminately blocking call would make the entire simulation system process wait for an event/resource which is external to the simulator itself. This object is designed to circumvent this constraint. A listener monitors a given set of file descriptors for input, resuming a target thread each time the condition is met. Once created, a listener should be attached to the thread manager's run chain using the scheduler's callout method. The callout objects are the one, which provide a simple means of having call back routines executed all along the simulation process, irrespective of the thread that is currently holding the control. These callout objects get fired each time the scheduler is asked to switch control to the next activable object.

### ***Statistical Class library***

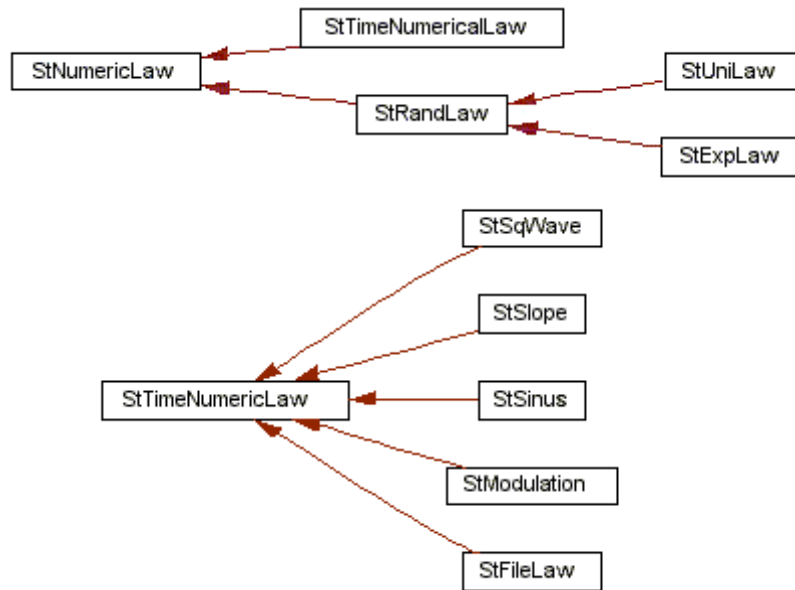
All the statistical classes have a corresponding simulation executive class, which exports the statistical classes computation to the display front-end using the monitor when the simulation is started in interactive mode. A simple basic set of statistical classes as stated below has been provided. Adding a few more for interested readers after understanding the clean interface with which the exiting ones have been implemented should be straightforward.

**Statistical Class hierarchy Diagram**



**Numerical law classes inheritance diagram**





### **StObject**

The basic behaviour of the statistical objects is implemented in this class. The StProto class is inherited to connect each measurement object to a display front-end through a protocol pilot. In other words, a statistical object's value can be exported to a front-end for display (usually an external application), and receive specific directives interactively that may alter its value and/or behaviour. The protocol pilot should be seen as a communication object aimed at exchanging messages back and forth between a measurement object and its display. A statistical object can be used in two separate contexts either as backend object, where it performs statistical measurements and send the collected data or as a front-end object, where it receives the data sent by the back-end object.

A statistical object has five major properties:

- A current value (floating-point).
- A count of received values since the object was last reset or created.
- Two varying bounds (lower and upper) that are updated each time a new value is logged
- A current count of samples.

Two primary methods that this object implements are add method using which values are added to the statistical object and get value method using which appropriate values are received based on the parameter passed to the get value method. Some of the valid parameters that can be passed to the get value method are as follows:

- VAL causing the last added value to be returned.

- NUM represents the number of collected values since the object was created or reset.
- MINVAL represents the object's lower bound value. In other words it is the lowest value collected through the add method.
- MAXVAL represents the object's upper bound value.

It also implements methods to reset values, sample values and calculate the final result on the object just before enquired.

### ***StCounter***

This class implements a measurement object, which is aimed at summing numerical values, with sampling capability. This class provides all the methods provided by the statistical object class.

### ***StIntegrator***

This class implements a time integrator object, which is aimed at computing the integration of a given variable with respect to time. The current time is obtained from the contents of the global variable clock. The start and end time duration for the integration process is specified during the creation of the integration object. In addition to the methods available from statistical object base class, it also provides methods like increment and decrement which adds the last entered value incremented or decremented from 1.0 to the object. Passing CMES as parameter to the get value function can retrieve the mean of the variable of the current sample. Similarly SUM would return the global integral measure since the beginning of measure. The rest are same as the basic statistical object's property.

### ***StHistogram***

This measurement object is aimed at determining the probability density of any statistical law, and computes the mean and standard deviation and accuracy evaluation for a given confidence interval. During the creation of a histogram values like number of bins, left bound and right bound are compulsory requirement.

Histograms can automatically adjust themselves to the actual range of the entered values. The expected behaviour when a value falls outside the current range of a histogram is selectable by an adjustable mode. Which can either be multiply or garbage. Multiply causes the histogram range to be adjusted by successive multiplication's by two, either to the left or to the right side according to the bound that is exceeded. This is the default mode. In the case of garbage mode all the values falling

outside are collected in the leftmost and the rightmost bins. By calling get value method with appropriate parameter one can retrieve the last added value, number of summed values since the histogram was created, sum of the values and the square of summed values.

### ***StScaler***

This class instances are statistical measurement objects aimed at computing the ration of a given value by another. A scaler establishes a relationship between a scaled object and a scaling one.

### ***StTimeScaler***

StTimeScaler instances are statistical measurement objects aimed at computing the division of a scaled object's value by the multiplication of the current clock value by a given factor. Whenever the inherited StScaler's get value method is invoked the following takes place

1. The current value of the scaled object is obtained by a call to its get value method with appropriate value type.
2. The divisor is computed by multiplying the current clock value by the object's time factor.
3. Finally, the current time scaler's value is set to the result of the dividing the scaled object's value by the divisor.

### ***StObjectScaler***

This class instances are statistical measurement objects aimed at computing the ratio of a scaled object's value by the value of a scaling object.

### ***StTimeGraph***

This class implements a measurement object, which is designed to grasp the temporal evolution of a give statistical object. A time graph can be connected to a plotter drawing its graphical representation through the StProto interface.

### ***StStateDiagram***

This class implements a backend object, which is designed to log and export state information to a drawing program. A state diagram can be connected to a plotter drawing the transitions between states through the StProto interface. Basically the state diagram is used to log state transitions along with the current clock value at which they

occur. It makes easy to propagate the state changes to a graphical display for monitoring those transitions. Known states are stored in array of strings. A state diagram object's current value is the array index of the last state entered by a call to add method.

### ***StProto***

This class implements a basic export protocol feature for statistical object's to send their current values and/or values to a display front-end. StProto is the super class of all exportable objects defined in the statistical object library. The purpose of this class is to define a conventional way of sending and receiving data between objects communicating remotely from computation back-end and the display front-end. It makes the sub classed objects exhibit a simple interface to accomplish this task in a well-defined manner. The StProto class does not define the actual mean of exchanging data, such as using sockets, shared memory, files or simple function calls. It rather defines a protocol to follow for each side involved in communication. The protocol pilots provided by the programmer perform the actual I/O operations. Messages are composed of type identifier and an optional dynamically sized data buffer.

### ***StNumericLaw***

This class implements the basic object for all numeric law and number generators. Though this class can be instantiated directly, it is usually subclassed to implement specific behaviours. Some of the methods that this class provides are iget to get an integer value and get to get a double value.

## **4. Steps to create a simple simulation system with sample files**

### **4.1 Configuration file**

The configuration file defines the GUI for tuneable parameters to the model. FROGS provides a set of pre-defined configuration attributes under its cfclass implementation. Using these configuration attributes one can define the necessary tuneable parameters for the model. The attributes should be defined in the constructor of the model defined under the RDEFINE (resource definition) structure. The values of the attributes are collected in the in the instantiate method. These configuration attributes allow the user to create a TCL/TK graphical user interface to set model's tuneable parameters.

Some of the predefined configuration class attributes are:

1. Integer Attribute – specifies integer value limited by a range
2. String Attribute – specifies strings.

3. Time Attribute – specifies time either in micro sec, Millie sec, or sec.
4. Option Attribute – specifies a set of options from which one can be chosen.
5. Resource Attribute – specifies another model resource for linked initialisation.
6. Real Attribute – specifies real values limited by range.
7. Radio Attribute – provides a set of from which one can be chosen.
8. Check Attribute – provides options from which multiple choices can be chosen.

## 4.2 Simulation Model file

This file defines the actual simulation code, which is to be simulated. An object of this class serves as a starting point for model instantiation, instantiated by the configuration class instantiate method.

## 4.3 Make file

Typical make file for model library creation would be something like this

```
CC = c++

RM = rm -f

LN_S = ln -s

# Set the FROGS variable to point to your installation root
FROGS = /usr/local/frogs-1.1

DESTDIR = /home/vasanth/project/lib/

OBSCAN = $(FROGS)/bin/obscan

INCLUDES = -I. -I$(FROGS)/include

# -fwritable-strings is for Tcl-related code

CXXFLAGS = -fwritable-strings -fpcc-struct-return -fnonnull-objects -fno-
exceptions

LIBS = -L$(FROGS)/lib -lsimex -lstatobj -lcfclass -lobstick -ldevkit -ltoolshop -ltcl
-lelf -lm -lnsl -ldl

%.o: %.cc %.h SarDefs.h SarReg.h

        $(CC) $(CXXFLAGS) -c -g $< -o $@ $(INCLUDES)

all: SAR15

# "SAR" is the executable used to bootstrap the simulation

SAR15: $(DESTDIR)libSAR15.so Main.o
```

```
$(CC) -o $(DESTDIR)$@ Main.o -L$(DESTDIR) -ISAR15 $(LIBS) -Wl,--
export-dynamic -Wl,--rpath -Wl,$(FROGS)/lib
```

Main.o: Main.cc

```
$(CC) $(CXXFLAGS) -c -g $< -o $@ $(INCLUDES)
```

# This is how to build the model library

```
$(DESTDIR)libSAR15.so: DbImpl.o SarConfig.o $(OBJFILES)
```

```
$(RM) $(DESTDIR)/libSAR15.so*
```

```
$(CC) -shared -o $(DESTDIR)libSAR15.so.0.0.0 DbImpl.o SarConfig.o
$(OBJFILES) -Wl,-soname -Wl,$(DESTDIR)libSAR15.so.0
```

```
$(LN_S) $(DESTDIR)libSAR15.so.0.0.0 $(DESTDIR)libSAR15.so
```

```
$(LN_S) $(DESTDIR)libSAR15.so.0.0.0 $(DESTDIR)libSAR15.so.0
```

# Update the p-class repository from config file

DbImpl.pcx: SarConfig.cc

```
$(OBSCAN) -i SarConfig.cc -r $@ $(INCLUDES)
```

# Generate the obstick support code from the repository contents

DbImpl.cc: DbImpl.pcx

```
$(OBSCAN) -o $@ -r $<
```

# The compiled obstick support code is part of the model library

DbImpl.o: DbImpl.cc

```
$(CC) $(CXXFLAGS) $(INCLUDES) -fPIC -c -g -o $@ $<
```

SarModel.cc Main.cc: SarModel.h

SarConfig.cc: SarConfig.h

clean:

```
rm -f *.o $(OBJFILES) *.pcx $(DESTDIR)SAR15 $(DESTDIR)libSAR15.so*
```

```
DbImpl.cc *~ *.~
```

rm:

```
rm -f *.cc~ *.h~ *~
```

.PHONY: clean

.PHONY: rm

#### 4.4 How to run the simulation through GUI

First the model library has to be loaded and the directory path should be exported in the LD\_LIBRARY\_PATH environment variable. Then once the model library is loaded u can instantiate model instance by right clicking on the model instances class under FROGS root node, Any configurable setting for the model can also be set by double clicking the model instances. After instantiating all the necessary models they should be associated in the architecture, create a new architecture using the ISE and add models by right clicking on the instantiated new architecture instance. The above would open the resource browser with available models which can be added to the architecture by double clicking on the model instance appearing in the resource browser. Then the project file is created by associating the executable created during model library creation and the created architecture. Subsequently the simulation can be started using the simulation run command in GUI. The FROGS user manual gives a detailed description on this process with necessary diagrams. Kindly refer to it for more details.

#### 4.5 How to run the simulation in command line

To run the simulation through the command line you need to do the following:

1. <Simulation executable name> -C <architecture name> or
2. <simulation executable name> -F <project file name>

The section 9.2 simulator's start options in FROGS's user manual would give a detailed overview on other start up facilities provided by FROGS.

#### 4.6 Environment settings for libraries

Some of the simulation environment settings that are provided can be set using the simulation configuration menu provided in the ISE. Some of them are listed below:

1. TCP/Server port is the TCP/IP port number used for communication between the ISE's monitor and the simulator.
2. Watchdog timeout is the time limit during which the connection must be made between the monitor and the simulator, in seconds.
3. Trace Buffer is the number of lines of text that can be stored in the monitor's trace result window.
4. Simulation Time is the simulation's duration.
5. Warm up Time is the start up period during which no statistical samples are taken.
6. Sampling Time is the number of statistical samples that must be taken. If the value

is null, the simulation is considered as having indefinite duration and the time indicated for simulation time is take as the sampling period.

7. Display tick defines the smallest time interval that can be displayed
8. Time unit is the default unit for displaying time.

## 5. Points to remember while creating simulation

### 5.1 Co-operative Scheduling

Since the event drive simulation kernel implements parallelism using co-routines and co-operative scheduling. The simulation system programmer should ensure that every process yields control for the execution of the other programmatically. Moreover it should also be ensured that all the simulation systems threads are not blocked at the same time, resulting in empty event list or schedule queue. The previous stated condition would abort the simulation process with a fatal error.

### 5.2 Incrementing simulation clock (through delay)

The only know way of incrementing the simulation system clock is delay method which bumps the system cock to the specified activation time when chosen by the scheduler for gaining the CPU. This seems to appear a major drawback when working out statistical analysis with respect to time which the user pre-defines in his program code.

### 5.3 Instantiating statistical objects with appropriate construction parameter.

Instantiation of statistical objects with appropriate parameters in very essential to get observable and meaningful statistical output. If the methodology for adding / collecting simulation data is selected without understanding its functionality then some absurd results can be observed. To avoid this it is better to have clear picture about the statistical object's functionality from the source. The brief explanation provided by the STATOBJ documentation would also help.

### 5.4 Defining and changing states of simulation system objects

Any instance of SxObject can be defined a set of states and can be manipulated as follows:

```
void SxThread::protoInit ()  
{  
    const char *stateArray[6];
```



```

stateArray[0] = "DEAD";
stateArray[1] = "IDLE";
stateArray[2] = "PENDING";
stateArray[3] = "PREEMPT";
stateArray[4] = "READY2RUN";
stateArray[5] = "RUNNING";
defineStates(sizeof(stateArray) / sizeof(stateArray[0]),stateArray);
SxTimed::protolnit();
}

```

The above method shows how a simulation system object can be defined to take different states.

```

int SxThread::stateIndex (int s)
{
return s < 2 ? 0 : s - 7;
}

```

The above implementation shows how the subclasses can re-scale the system object's state values before sending them to the state diagram superclass. The SxObject::signal typically calls this method with the new entered state before passing it to the StStateDiagram::add() method for logging and export to display front end.

For instance, one could need to map states DEAD and KILLED to the display state DORMANT indexed on value 0. So this method could be reimplemented as in the following one:

```

Int SomeTaskObject::stateIndex(int _state)

{

if(_state == DEAD || _state == KILLED)

return 0;

return _state;

}

void SomeTaskObject::protolnit()

{

const char* stateArray[3];

```

```

stateArray[0] = "DORAMANT";

stateArray[1] = ?

?

}

```

The system Objects State can be retrieved and set through the following methods

```

Int getState();

Int setState(int state);

```

## 5.5 Exporting the simulation system objects to display front end.

To display the state transitions of the simulation system objects or other statistical objects time-curves, the appropriate object should be exported to the display front-end by the setProtoname and setProtoexportable in its constructor.

## 6. Unexplored areas

### 6.1 Script Attribute

Basically, the "script" attribute is designed to allow implementing any other kind of attributes that's not provided by default by the CfClass package. This means that instead of using a built-in Tcl script to draw an attribute on the Tk frame, collect then pass back its value to the manager (such as they already exist for time values, integers, handlers, resource links etc.). A user-provided Tcl script, which is given as part of the attribute definition, is invoked to do the job. Create a script attribute like this, in the C++ constructor of a class extending CfClass:

```

ScriptAttributePointer scriptPtr =

new ScriptAttribute("<attribute name>", this, "<tcl-plugin>", "<tcl-prefix>", "<config tab
name>");

```

- <attribute name> is used in the attribute label.
- <tcl-plugin> is a directory name under <frogs-install>/share/frogs/tcl from where the user-provided Tcl script is expected to be loaded (ensure that a tclIndex file is available from it). If your script is embedded into your executable, just pass a default value like "generic" or whatever.
- - <tcl-prefix> is the prefix of the (conventionally named) procedures the attribute manager expects to find in your script (There are two procs to provide to the

manager, Check the example below, where the prefix is "MyAttribute").

- – <config tab name> tells to which configuration group your attribute will be attached by the manager. Each group has its own graphic tab from the configuration (Tix) notebook associated to each instance. The protocol is as follows: When a script attribute is encountered as part of a configuration window, its configure{} proc is called to draw any number of graphic widgets it needs to allow the user to give this attribute a value. When the "Save" button is called for the configuration window (i.e. Tix notebook) holding a script attribute, the script's validate{} proc is called to fetch back its value. This value will be stored into the resource database by the attribute manager. The value will be passed as the last argument ("settings") of the script's configure{} proc the next time this attribute is activated, and so on.
- proc MyAttribute:configure {context frame name settings} { # "context" identifies the TkBridge context. You can use it # as a unique key naming the configuration window. # "frame" is the Tk frame on which the script is expected to # place some kind of widget enabling the user to enter a value # for the attribute (a text field, a checkbox...). The script # "owns" this frame exclusively. # "name" is the name of the attribute you can use to label a # widget with. # "settings" is the current value of the attribute (i.e. on # entry). }
- proc MyAttribute:validate {context frame} { # Return the value of the attribute (i.e. on exit) return { { key1 value1 } { key2 value2 } ... { keyN valueN } } }
- Take a look at the carbonkernel sources in carbonkernel/ise/frontend/tcl/panel.tcl to find a current use of this attribute. It is used to implement the CarbonKernel "magnet" feature.
- Note: The CfClass package defines the protocol and the C++ part of the attribute manager, but the Tcl/Tk (i.e. graphical) parts are always provided by the high-end application using it, such as "frogs" or "ck" (i.e. frogs/ise/tcl or carbonkernel/ise/frontend/tcl).

## 6.2 Separation of ISE from kernel

The integrated simulation environment or the ISE communicates with the simulation kernel through a SxTCP socket interface implemented in the SIMEX implementation. The simulation configuration window, which requests for a monitor port substantiates this claim. The way the current implementation works is as follows:

- The monitor or the ISE is the first one to get instantiated by the simulation manager or simulation system thread.

- The simulation monitor starts in server mode and opens a socket in the specified simulation configuration port and forks for the simulation kernel, which then connects back to the simulation monitor.
- Manipulating this process after understanding the StProto the protocol defining the communication can extend this to a pseudo distributed environment. This extension would help in better performance of the simulation system. As the simulation workload (computation and graphical display) is done on separate processors.

### **6.3 Other means to increment the simulation clock**

Currently the usage of delay method seems to be the only way incrementing the simulation system clock. Hence estimating any desired parameters with reference to simulation time is very inaccurate. The above statement simply means that in event driven simulation process, where simulation of non-interacting parallel processes is simulated the concept of analysing any parameter with respect to time is baseless. This is because the simulation clock as such is incremented by the user's simulation code.

### **6.4 Possibility of an application outside FROGS environment to communicate with model instances.**

Unfaithfully, there is no such programming interface. But since the ISE is written in Tcl, you could use the remote command feature of Tcl "send". To get requests from an outsider app, then call the C++ and/or Tcl layers of the receiving ISE. Warning, the Tcl "send" command is inhibited being replaced by an empty procedure in the main.tcl module. You'll need to provide your own if you follow this path. If you only want to start the simulation, you can use the command-line options to start it from the mere simulator executable instead of using the ISE. At any cost the configuration and creation of models and architecture should once again be done only with the help of ISE.