SIM : a C++ library for Discrete Event Simulation

Dirk Bolier - Anton Eliëns
Vrije Universiteit, Department of Mathematics and Computer Science
De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands
email: dbolier@cs.vu.nl - eliens@cs.vu.nl

October 31, 1995

Abstract

In this report we give a full description of *sim*, a C++ library for discrete event simulation. The *sim* library supports both an *event* and *process-oriented* approach to developing simulations. Events as well as entities (which may be considered as objects in the simulation combining several related events and having an additional phase signifying episodes in its life-time) are provided as abstract classes that must be refined by the application programmer to define the actual events and entities participating in the simulation.

The *sim* library is integrated with the *hush* library, thus offering powerful graphic and animation facilities. However, the *sim* library may also be used independently, on both Unix and MS-Dos platforms.

This report presents an overview of the classes constituting the *sim* library as well as two standard examples and various code fragments illustrating the deployment of the classes in writing simulation programs.

Keywords: discrete event simulation, events, entities, modeling, simulation experiments, simulation animation, simulation analysis

---

0

Contents

1 Introduction

Discrete Event Simulation A simulation is the execution in time of a model, represented by a computer program, that gives information about a system being investigated. The library *sim* adopts the method of the Discrete Event Simulation. With this approach, the components of the model consist of events, which are activated at certain points in time and in this way affect the overall state of the system. The points in time that an event is activated are randomized. Events exist autonomously and they are discrete so between the execution of two events nothing happens. The library *sim* provides an event-based, as well as a process-oriented approach of writing a simulation program. With the process-oriented approach, the components of the program consist of entities, which combine several related events. Events and entities are provided as abstract classes that must be refined by the application programmer to define the actual events and entities participating in the simulation.

Object Oriented *Sim* is written in the C++ programming language, which is presented in [Stroustrup 91]. The library provides classes, that can be used by writing a C++ simulation program. We provided the library with such a functionality, that the resulting language is similar to *simula*, which is described in [Pooley 86]. The structures underlying the scheduler, queue and histogram classes and most of the code implementing the probability distributions have originally been adapted from an excellent C simulation library, presented in [Watkins 93].

Hush The library can be compiled to both a pure ASCII version (which is the fastest) as well as a *hush* version (which has graphical features). Simulation programs can be linked to both versions of the library. The *hush* library is not described here. See [Eliens 95] for a description.

Availability The *sim* library has been in use for student programming assignments at the Vrije Universiteit for over a year. It may be obtained by anonymous ftp from `ftp.cs.vu.nl`, directory `eliens/sim`. You may also retrieve it via `http://www.cs.vu.nl/~eliens/sim/`. It comes with the sources, manual pages, a variety of examples and an extensive users-guide.

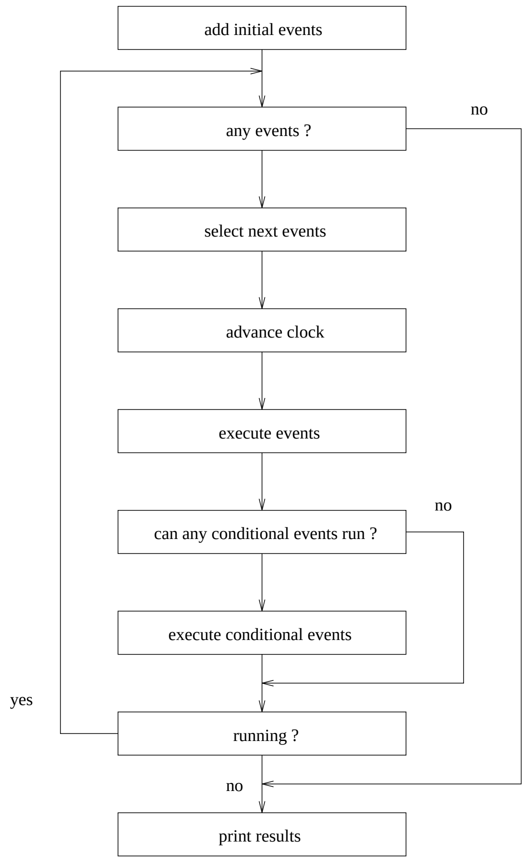
2 Program structure

A typical simulation program, employing the *sim* library, creates and runs an application derived from the *session* class. The *session::main* function then contains the creation of a simulation object, which makes it possible to use the simulation primitives. It creates the participating events and entities, which are derived from the corresponding abstract classes and which are given functionality in their function operators. Furthermore the required resources and queues are created. The resources represent passive objects used by events while queues are used by events waiting on a resource to become free. Histogram and analysis objects can be created to gather and analyze results. Before running the simulation, it should be set up by scheduling some events or entities depending on the type of simulation.

2.1 The event scheduling strategy

When a simulation is set up it runs after invoking the *simulation::run* method in the *session::main* function. The simulation then runs until the *simulation::quit* method is invoked, until there are no events left or for a specified number of time units or terminated events.

At the time an event is due to be activated, it is extracted from the scheduler and the main simulation routine executes the code from the *function operator* of that event. The scheduling strategy employed by *sim* is depicted in the following figure.



Notice that the algorithm extracts all events with the same activation time. The main simulation routine activates the events in priority order with the highest scheduling priority first. Before executing the events, the simulation clock is updated to the activation time of the current events. Furthermore, the FIFO conditional list is traversed in priority order with highest scheduling priority first. Events, that can run now, are executed. Events that are not used anymore should be terminated to prevent the system from overflow.

After the running of the simulation some statistics of the starting and end time, of the number of created, activated and terminated events and of the actual time taken are given.

2.2 Primitives and event states

When an event is activated it can be manipulated in various ways. The event can be appended to a queue, it can be put back in the scheduler and so on. On its way it changes state. We identify the following states an event can be in (based on the simula states) :

- *passive* - indicating that the event is currently not available for any processing.
- *active* - indicating that this is the event currently being processed.
- *queued* - the event is in a queue.

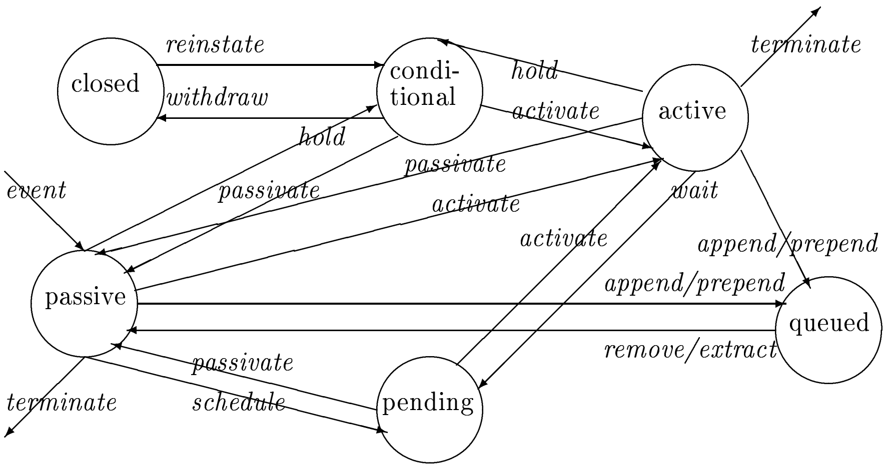
- *pending* - the event is in the scheduler.
- *conditional* - indicating the event is on the conditional list.
- *closed* - the event is on the conditional list but unavailable.

The various primitives that manipulate the event and cause the change of state are then:

- *passivate* - which takes an event out of the simulation but does not destroy the event.
- *hold* - which takes a passive or active event and adds it to the conditional list. This event then fires after each activated event.
- *schedule* - which takes a passive event and adds it to the scheduler. This event then executes after a specified period.
- *wait* - which puts the event currently executing on the scheduler.
- *activate* - which takes a passive, pending or conditional event and schedules it to become active at the current simulation time. A typical use is to bring forward the activation of an event at present time.
- *terminate* - which terminates an active or passive event.
- *append, prepend* - which add passive and active events to a queue.
- *extract, removefront, removeback* - these functions obtain events from the queue.
- *withdraw, reinstate* - which respectively prevent a conditional event from executing by changing its state to closed and put a closed event back in its conditional state.

The invoking of a primitive has not always effect. The event should be in the correct state first. The invoking of the *hold* primitive for example assumes a passive or active event. This state can be obtained by first passivating or activating the event.

Consider the following transition diagram that shows the different states an event can be in, in relationship with the different functions, that manipulate them. The central state of an event is the *passive* state. Most events should first be passivated before they continue. Notice that an event also changes state if it is *pending* and is due to execute. This event (the first event in the scheduler) is extracted from the scheduler and becomes *active*. An event that is on the conditional list stays *conditional* while executing until it is explicitly passivated.



Entities An entity (that combines several related events) has the same states and primitives as an event. The difference is in the use of an additional phase, as is explained in section 3.4.

2.3 The event-based approach

With the event-based approach of writing a simulation program we first identify the events in the model. The behavior of an event is implemented by deriving it from the class *event* and overriding the *function operator* of this class. Consider the next problem.

2.3.1 The Dining Philosophers

The problem is as follows. Five philosophers sit around a table with five chopsticks in between. They think and if they are hungry and if two chopsticks are available, they eat. If a philosopher gets hungry and s/he can not acquire a chopstick, the philosopher waits until s/he can. The philosopher does not think, if s/he is waiting or eating. We are interested in the fraction of the time, that a philosopher actually thinks.

To model this problem, three events can be identified, *eat*, *think* and *await*. The corresponding classes are derived from the class *event*. Furthermore we need a chopstick for every philosopher. These are represented as a *resource*. The thinking times are gathered in an instance of the class *histogram* and the *generator* takes care of the variations in the time needed to think and eat.

We develop this program in the following steps. First, the library is included as *sim.h*. The declarations of the global variables and constants follow after that. The time unit in this simulation is an hour, so a philosopher has a mean eating time of two hours and a mean thinking time of five hours. The duration of the simulation is a year. Finally, we define the various events.

```
#include <sim/sim.h>
```

```
const double duration = 52*7*24.0;       // a year
const int number = 5;                    // philosophers
const int eatingtime = 2;                // 2 hours
const int thinkingtime = 5;              // 5 hours
simulation* sim;
generator* g;
resource* chopstick[number];
histogram* thinking;

class eat : public event
{
public :
  eat(int i);                  // constructor, taking identity
  virtual int operator()();   // function operator
private :
  int id;                      // identity of the philosopher
};

class think : public event
{
public :
  think(int i);                // constructor, taking identity
  virtual int operator()();   // function operator
private :
  int id;                      // identity of the philosopher
};

class await : public event
{
public :
  await(int i);                // constructor, taking identity
  virtual int operator()();   // function operator
private :
  int id;                      // identity of the philosopher
};
```

Next, we implement the various events. An event is given its functionality by deriving it from the class *event* and overriding its *function operator*.

The logic of the *eat* event is that the philosopher eats for a random time, exponentially distributed with a mean eating time. So, we first determine the actual eating time and schedule a *think* event to be activated after this eating time. The *eat* event can be terminated.

```
eat::eat(int i) : event()
{
  id = i;                                 // set identity
}

int eat::operator()()
{
```

```
  double t = g -> exponential(eatingtime); // determine eating time
  think* th = new think(id);               // create a thinking event
  sim -> schedule(th,t);                   // schedule thinking
  sim -> terminate(this);                  // terminate this eat event
  return OK;
}
```

If a philosopher starts to think, the philosopher first releases both chopsticks. The thinking time is determined and a sample is made of the percentage of this thinking time towards the total time. An *await* event is scheduled and the *think* event is terminated.

```
think::think(int i) : event()
{
  id = i;                                 // set identity
}

int think::operator()()
{
  chopstick[id] -> release();             // release left chopstick
  chopstick[(id+1)%number] -> release(); // release right
  double t = g -> exponential(thinkingtime);    // determine thinking time
  thinking -> sample(id,t/duration*100); // add a sample (%)
  await* aw = new await(id);              // create await event
  sim -> schedule(aw,t);                  // schedule waiting
  sim -> terminate(this);                 // terminate thinking
  return OK;
}
```

The *await* event acquires the left and right chopstick and schedules an *eat* event immediately, if both chopsticks are available. The *await* event is passivated as it could be on the conditional list. If no chopsticks are available, the *await* event stays on the conditional list or, if it was not conditional as is the case the first time it is activated, it is added to the conditional list.

```
await::await(int i) : event()
{
  id = i;                                  // set identity
}

int await::operator()()
{
  if ( (chopstick[id] -> available()) &&      // available ?
      (chopstick[(id+1)%number] -> available()) )
  {
    chopstick[id] -> acquire();               // acquire left
    chopstick[(id+1)%number] -> acquire();    // acquire right
    eat* e = new eat(id);
    sim -> passivate(this);      // extract from conditional list
```

```
    sim -> schedule(e,0);        // schedule eat event immediately
    sim -> terminate(this);      // terminate await event
  }
  else if (!conditional())       // not on conditional list
    sim -> hold(this);           // add to conditional list
  return OK;
}
```

The following step is the definition and implementation of an application, that is derived from *session*. The *application::main* function first creates the simulation object. Furthermore, a frequency histogram and five resources that represent the chopsticks are created. The histogram is created with its widget path and with its (default) options. Afterwards it is packed to the display. The simulation starts with all philosophers waiting and runs for a year (52*7*24 hours). After running the simulation, the resulting histogram is printed.

```
class application : public session
{
public :
  application(int argc,char** argv);
  int main();                          // must be defined
};

application::application(int argc,char** argv) : session(argc,argv,"philosophers")
  {}

int application::main()   // tk is an instance variable of session
{
  sim = new simulation();
  g = new generator(80,20,19);          // gets three seeds
  thinking = new histogram(".h","-columns 5 -title thinkingtime");
  tk -> pack(thinking);                 // add to display;
  tk -> update();                       // update display;
  for (int i=0;i<number;i++)
  {
    chopstick[i] = new resource(1);     // create chopsticks
    await* aw = new await(i);           // schedule each
    sim -> schedule(aw,0);              // philosopher waiting
  }
  sim -> run(duration);                 // run for duration time units
  cout << (*thinking) << endl;          // print resulting histogram
  delete thinking;
  delete sim;
  return 0;                             // successful termination
}
```

The *main* function then creates and starts the application.

```
main(int argc,char** argv)
{
  session* s = new application(argc,argv);        // create the session
  return s -> run();                              // start the session
}
```

2.3.2 The event class

The interface of the *event* class :

```
interface event
{
protected :
  event(int qp=0,int sp=0,int kf = FALSE);  // queuing and scheduling
                              // priority and kill flag as arguments

public :
  virtual ~event();                  // destructor.

  virtual int operator()() = 0;      // empty

  virtual int verify();              // returns OK

  int report(histogram* h,double interval = 0);  // create report

  void stamp();                      // add time stamp
  double timespent();                // time since the stamp

  int queuingpriority();             // return queuing priority
  int schedulingpriority();          // return scheduling priority
  void queuingpriority(int p);       // adjust queuing priority
  void schedulingpriority(int p);    // adjust scheduling priority

  int active();                      // is active ?
  int pending();                     // is pending ?
  int conditional();                 // is conditional ?
  int closed();                      // is closed ?
  int passive();                     // is passive ?
  int queued();                      // is queued ?
};
```

When creating an event it receives the queuing and scheduling priorities and a kill flag, which should be set to OK when the simulation is ran for a number of terminated events. The report method generates a histogram of the time the events stayed in the simulation. It is printed each *interval* time units.

2.4 The process-oriented approach

With the process-oriented approach the components of the model consist of entities, which represent the existence of some object in the system such as a philosopher. An entity receives a user-defined phase that determines the behavior of the entity.

We first identify the entities (or the types) in the model. The events are represented as methods of an entity. The *function operator* calls these events based on the *phase* the entity is in.

2.4.1 The entity philosopher

Consider the following definition and implementation of the entity philosopher

```
enum {EATING,THINKING,WAITING};      // phases of a philosopher

class philosopher : public entity
{
public :
  philosopher(int ph,int i);  // constructor, taking phase and id
  virtual int operator()();   // function operator
  int eat();                  // eat event
  int think();                // think event
  int await();                // await event
private :
  int id;
  generator* g;
};

philosopher::philosopher(int ph,int i) : entity(ph)
{
  id = i;                   // set phase and identity
  g = new generator(20,10,999);
}

int philosopher::operator()()
{
  switch (phase())          // what phase is the philosopher in ?
  {
  case EATING :
    return eat();           // the philosopher eats
  case THINKING :
    return think();         // the philosopher thinks
  case WAITING :
    return await();         // the philosopher waits
  }
  return FALSE;
}

int philosopher::eat()
{
  double t = g -> exponential(eatingtime);    // determine eating time
```

```
    sim -> wait(t);            // schedule this philosopher thinking
    phase(THINKING);                   // set phase to thinking
    return OK;
  }

  int philosopher::think()
  {
    chopstick[id] -> release();            // release left chopstick
    chopstick[(id+1)%number] -> release(); // release right
    double t = g -> exponential(thinkingtime);   // determine thinking time
    thinking -> sample(id,t/duration*100);      // sample (%)
    sim -> wait(t);              // schedule this philosopher waiting
    phase(WAITING);                        // set phase on waiting
    return OK;
  }

  int philosopher::await()
  {
    if ( (chopstick[id] -> available()) &&    // available ?
       (chopstick[(id+1)%number] -> available()) )
    {
      chopstick[id] -> acquire();          // acquire left chopstick
      chopstick[(id+1)%number] -> acquire();      // acquire right
      sim -> passivate(this);         // make passive
      sim -> activate(this);          // activate as eating
      phase(EATING);                  // set phase on eating
    }
    else if (!conditional())
      sim -> hold(this);                  // add to conditional
    return OK;
  }
```

Dependent on the phase the philosopher is in, the appropriate action on the simulation environment is taken. These actions closely resemble the events, described in the event-based approach of this problem. The main difference is in the use of *phase*. If, for example, a philosopher finishes eating, his/her phase is set to THINKING and s/he is scheduled after t time units, whereas in the event-based approach a think event is scheduled and the eat event is explicitly terminated. So, in the process-oriented solution a philosopher exists for the entire simulation. In the *session::main* function the simulation is set up by scheduling the five philosophers, initially waiting, instead of scheduling five *await* events.

2.4.2 The entity class

The interface of the *entity* class. It is derived from event and adds the phase of the entity to event.

```
  interface entity : public event
  {
  public :
    virtual ~entity();           // destructor
    int phase();                 // return phase
```

```
  void phase(int p);            // adjust phase
protected :
  entity(int ph=0,int qp=0,int sp=0,int kf = FALSE); // constructor
};
```

The constructor takes the phase, queuing priority and scheduling priority and the kill flag of the entity (default FALSE), The function *phase* adjust or returns this phase. Furthermore, the class *entity* inherits every function of the class *event*, so these functions can also be invoked for an entity.

2.5 The simulation class

The primitives and methods for running and ending the simulation are provided by the *simulation* class. The *simulation* class has the following interface :

```
interface simulation
{
   simulation();
   virtual ~simulation();

   void run();                       // run until no events left
   void run(double t);               // run for t time units
   void run(int count);              // run for count terminated events

   void quit();                      // quit simulation

   void terminate(event* e);         // terminate e

   int schedule(event* e,double t); // schedule e at t
   int wait(double t);               // schedule current
   int passivate(event* e);          // passivate e
   int activate(event* e);           // activate e
   int hold(event* e);               // add e to conditional
   int withdraw(event* e);           // put e on closed
   int reinstate(event* e);          // put e on conditional

   void reset();                     // reset the simulation

   double clock();                   // return clock

   void init(char* options = "");   // initialize

   int cancel();                     // cancel events

   screen* format(char* p,char* options = "");  // statistics
   screen* format(widget* w,char* p,char* options = "");

   void postscript(char* fn);        // generate postscript
```

```
    friend ostream& operator<<(ostream& os,simulation& s);
};
```

The simulation object should be created before other objects participating in the program are created. The method *run* is used to run the simulation for t time units, for *count* terminated events or until no scheduled events are left. The simulation can be stopped by calling *quit*. The primitives are described in section 3.2. *Reset* clears the scheduler and conditional list, resets the clock to 0 and resets the reports and widgets (if set) to their original state. The function *clock* returns the current simulation time. The *init* method handles the following options :

- -realtime (double) - default 0
- -scan (yes,no) - default no
- -suppress (none,reports,statistics,both) - default none

If the constructor takes the *-realtime* argument the library runs in real time mode. A run then ends if the simulation time divided by the double is greater then the actual time (in seconds) that elapsed. If the *-scan* option is set to yes, the list with the conditional events is traversed, even between events with the same activation time. Setting *-suppress* has as effect that generated reports and/or statistics are not printed after a run. The method *cancel* removes and deletes every event from the scheduler and conditional list that receives FALSE from its *event::verify* method. This method can be used to model the breakdown of a server for example, departure events should be canceled then. The *format* method creates a screen with the specified path for the statistics and handles the following options :

- -foreground (color) - default black
- -background (color) - default white
- -width (int) - default 400
- -height (int) - default 120

The size can be adjusted but the pre condition is that width/400 is equal to height/120. If *format* is not used the statistics are printed on a default toplevel screen (if in *hush* mode). The function *postscript* writes the window of the simulation object (if created) in postscript format to the specified file. The *operator<<* function can be used when debugging a program. It prints the contents of the scheduler and conditional list.

2.6 The generator class

The class *generator* represents the random number generator. The generator is used to get the randomized activation times of the various events. It gets three positive seeds (≤ 30000) and creates a random number stream, according to the method described in appendix A.

Each time a member of the generator is invoked to get a random number, the next number from the stream is calculated and is transformed, according to the probability distribution belonging to that member. Refer to appendix B for the definitions. This number is returned and can be used in the simulation. If, for example, a *normal* distribution with 10 as mean and 1 as standard deviation is specified to transform the random number stream, the generated numbers have 10 as mean and show little variation because of the low standard deviation.

The class *generator* has the following interface :

```
interface generator
{
 public :
   generator(unsigned int s1,unsigned int s2 = 0,unsigned int s3 = 0);

   void antithetic(); // use 1-random()

   void reset();      // reset random number stream

   int probability(double p = 0.5); // OK with prob. p

   double uniform(double a,double b);
   double exponential(double a);
   double hyperexponential(double p,double a,double b);
   double laplace(double a);
   double normal(double a,double b);
   double chisquare(unsigned int n);
   double student(unsigned int n);
   double lognormal(double a,double b);
   double erlang(unsigned int n,double a);
   double gamma(double a,double b);
   double beta(double a,double b);
   double fdistribution(unsigned int n,unsigned int m);
   double poisson(double a);
   double geometric(double p);
   double hypergeometric(unsigned int m,unsigned int n,double p);
   double weibull(double a,double b);
   double binomial(double p,unsigned int n);
   double negativebinomial(double p,unsigned int n);
   double triangular(double a);

 protected :
   double random();    // produces a random number between 0 and 1
};
```

A generator, with given seeds, produces the same random number stream every time, it is created with those seeds. If the defaults are chosen the last two seeds are equal to the first one. One should fluctuate seeds to get reliable random numbers.

If the method *antithetic* is invoked, the inverse of the current stream is used, i.e. instead of the generated random number, say x, *1-x* is used. This method can be used when the variance of the samples made is too high.

The function *reset* resets the random number stream, as if no random number was generated from this stream. The member *probability* returns OK with probability p and FALSE with probability 1-p. The *distribution* functions all return a random number from the stream, following the belonging probability distribution. The method *random* produces a new variate (between 0 and 1) from the random number stream. It is made protected so a derived generator can use this stream and add new distributions to the existing ones.

2.7 Hush classes

The library includes four classes from the *hush* library. If the library runs in ASCII mode their functionality is zero. We only describe the part of the interface that is essential for the *sim* library. Refer to [Eliens 95] for a complete description.

2.7.1 The session class

The *session* class is used to start up the *hush* toolkit as in the program of the Dining Philosophers. Its interface is (incompletely) characterized by:

```
interface session
{
  session(int argc,char** argv,char* name = NULL);   // create
  virtual int main();                                 // initialize
  int run();                                          // start and run
};
```

2.7.2 The kit class

The *kit* class is used to pack widgets to the display and to update this display. Its (incomplete) interface :

```
interface kit
{
  widget* pack(widget* w,char* options = "");    // add to display
  void update(char* options = "");               // update display
};
```

2.7.3 The widget class

The *widget* class is abstract. All the graphical widgets are derived from this class.

```
interface widget
{
  widget(char* p);
  widget(widget* w,char* p);
  widget* pack(char* options="");              // pack to display
  void destroy();             // remove from display
};
```

2.7.4 The toplevel widget

A widget can be packed to the display (successive packs result in a layer of widgets then), it can be embedded inside the screen widget or it can be made toplevel, which results in a stand-alone window for the widget. The histogram of the Dining Philosophers Problem could be made toplevel by giving a toplevel widget to its constructor.

```
toplevel* t = new toplevel(".histogram");
thinking = new histogram(t,".h","-title thinkingtime");
thinking->pack();
```

The interface of the *toplevel* class :

```
interface toplevel : public widget
{
  toplevel(char* p,char* options = "");
  toplevel(widget* w,char* p,char* options = "");
};
```

3 Queuing events

The notion of a queue as a waiting line of a service facility is familiar to most of us. Queues occur wherever an imbalance occurs between requests for a limited resource and the ability of a service facility to provide that resource. The size of the queue depends on the amount of the resource available and the demand for it by customers. You can think of queuing applications, that make a trade-off between the cost of providing more of a resource and the cost of allowing larger queues to form. In the following sections we look at some typical queue types and queue behavior. Furthermore we describe the interfaces of the classes *resource* and *queue* based on a standard queuing example.

3.1 Queues and resources

In systems where customers arrive and depart, after being served we need at least three events (or an entity for both the customer and server). These types are in fact general for every queuing application. The difference then comes from the arrival and service pattern, from the service policy, from the resource amount and from the queuing priority of every event. These can be modeled as follows :

- *arrival and service pattern* - In the *arrival* event a next arrival is scheduled after the appropriate time. The *service* event schedules a *departure* event after the service time.
- *service policy* - When an arrival occurs it is served according to the service policy. The most common ones are the LIFO (last in first out) and FIFO (first in first out) queues. The first one can be modeled by adding an arrival to the front of the queue with the *queue::prepend* method, the last one is modeled by adding the arrival event to the back with the *queue::append* method. In both cases we then take the front of the queue with the *queue::removefront* method. Additional service policies can easily be modeled as variations of the ones above.
- *resource amount* - The number of arrivals that can be served depends on the resource amount. When a resource is created it receives its initial amount. This amount is then increased and decreased every time a certain amount is released or acquired, with the *resource::release* and *resource::acquire* methods. A resource can be released in the departure event (after being served) and can be acquired in the service event if the asked amount is available, which can be checked with the *resource::available* method.

- *queuing priority* - A priority queue can be modeled by giving a queuing priority to an event, when created. When appended or prepended an event that has a higher priority goes in front of an event with a lower priority.

3.2 An M/M/1 Queue

An M/M/1 queue has an exponential inter-arrival and service time and a single server. We are interested in the statistics of the size of the queue and the waiting times of a customer, with varying means for the inter-arrival times and service times. Consider the following process-oriented simulation program.

We first include the library and define the phases for a customer. The simulation object, the resource, the generators, the queue and the means for the arrival and service times, as well as the customer and server entities are declared.

```
#include <sim/sim.h>

enum {ARRIVAL,DEPARTURE};

simulation* sim;
resource* r;
queue* q;
double meanarrival,meanservice;  // mean inter-arrival and service time
generator* g;

class customer : public entity
{
public :
  customer(int ph);
  virtual int operator()();
  int arrival();
  int departure();
};

class server : public entity
{
public :
  server();
  virtual int operator()();
};
```

A customer arrives or departs, so it receives a phase, when created. The behavior of the entity *customer* is implemented in its application operator. When a customer arrives, first a new customer is scheduled to arrive. We take the inter-arrival time to be exponential, with *meanarrival* as the mean inter-arrival time. Then the customer is appended to the queue. When the customer is served the resource is acquired by the server, a departing customer then releases the resource and is terminated. The default amount of 1 is released and acquired each time.

```
customer::customer(int ph) : entity(ph)
{
```

```
}

int customer::operator()()
{
  switch (phase())
  {
    case ARRIVAL :
      return arrival();
    case DEPARTURE :
      return departure();
  }
  return FALSE;
}

int customer::arrival()
{
  customer* c = new customer(ARRIVAL);     // schedule new customer
  sim -> schedule(c,(g -> exponential(meanarrival)));
  q -> append(this);              // append to the queue
  return OK;
}

int customer::departure()
{
  r -> release();                 // release the resource
  sim -> terminate(this);         // terminate this customer
  return OK;
}
```

The behavior of the entity *server* does not depend on a phase. Every time it is activated, it takes a customer from the queue if possible, i.e. if the queue is not empty and if the required amount of the resource (in this case the default amount of 1) is available. If it takes a customer, it acquires the resource, adjusts the phase of the customer to DEPARTURE and schedules the customer to depart, after it has been served.

```
server::server() : entity()
{
}

int server::operator()()
{
  // can we serve a customer ?
  if ( (!(q -> empty())) && (r -> available()) )
  {
    customer* c = (customer *)q -> removefront();    // take first
    r -> acquire();                           // acquire the resource
    sim -> schedule(c,g -> exponential(meanservice));
    c -> phase(DEPARTURE);    // adjust phase and schedule departing
  }
```

```
    return OK;
  }
```

The function *application::main* creates the queue and two histograms, that are used for generating reports on the size of the queue and the waiting times of a customer. A resource with an initial amount of 1 is created. The simulation is set up by scheduling a customer to arrive immediately and making the server conditional. The server is never passivated, so it stays on the conditional list for the entire simulation and is activated, whenever a customer is. If the simulation has various queues and servers the server should be scheduled whenever a customer departs to make the program more efficient.

```
class application : public session
{
public :
  application(int argc,char** argv);
  int main();
};

application::application(int argc,char** argv) : session(argc,argv,"M/M/1-queue")
  {}

int application::main()
{
  cout << "mean time between two arrival events" << endl;
  cin >> meanarrival;
  cout << "mean time between two service events" << endl;
  cin >> meanservice;

  // create objects and reports
  sim = new simulation();
  g = new generator(8,2,9);
  histogram* h1 = new histogram(".h1","-type weighted -form statistics -title size");
  histogram* h2 = new histogram(".h2","-type frequency -form statistics -title waitingtimes");
  tk -> pack(h1);
  tk -> pack(h2);
  tk -> update();
  q = new queue();
  q -> reportsize(h1);    // generate a report on the size of the queue
  q -> reportwaiting(h2); // generate a report on waiting times
  r = new resource(1);

  // set up and run the simulation
  customer* c = new customer(ARRIVAL);
  sim -> schedule(c,0.0);         // initially a customer arrives
  server* s = new server();
  sim -> hold(s);                 // the server is conditional
  sim -> run(10000.0);

  delete h1;
  delete h2;
```

```
  delete q;
  delete sim;
  return 0;
}

main(int argc,char** argv)
{
  session *s = new application(argc,argv); // create the session
  return s -> run();                       // start the session
}
```

3.3 Queue behavior

In this section we look at some examples of how more complex scenarios involving queues can be modeled.

Queue swapping This behavior occurs when a member of the queue leaves in preference for an alternative queue. We can model the swapping of an event e between the queues $q1$ and $q2$ as follows :

```
if (condition)
{
  if (q1 -> extract(e))         // from every position
    q2 -> append(e);            // append to second queue
}
```

where condition could be something as the checking of the sizes of the two queues.

Balking If a queue is too long it usually deters further additions. This phenomenon is called balking and its implementation is straightforward :

```
if (q -> size() < MAX)
  q -> append(e);               // append if possible
```

The event is appended when there is room for it.

Reneging Reneging is the premature departure of a customer from a queue before it has had any service. This is typically the case when its waiting time exceeded some upper limit. We could model this by checking the entire queue with the *queue::cancel* method, but more efficient is the following addition to the *arrival* event :

```
cancel* cn = new cancel(this);   // give customer to the cancel event
sim -> schedule(cn,MAX);         // schedule cancel event
```

and the extracting of the event in the *cancel::operator* function. The *queue::extract* function does nothing if the event is not in the queue.

```
int cancel::operator()()
{
  if (q -> extract(e))          // extract and terminate e if in the queue
    sim -> terminate(e);
  return OK;
}
```

Service pre-emption Pre-emption occurs when a customer being served is interrupted by the server in favor of a new customer that has a higher priority. The interrupted customer can then wait to complete its service, wait to be served again or simply leave the system. The interrupting arrival then takes the place of the interrupted event. Consider the following behavior of the server entity of the M/M/1 queue. Events should then receive a queuing priority when created and the event currently being served should be set to NULL when terminated.

```
int server::operator()()
{
  if (!(q -> empty()))                 // if an event waiting
    if (!current)                      // if nothing being served
      return serve();                  // just serve
    else if (q -> front() -> queuingpriority() > current -> queuingpriority())
      return preempt();                // front has a higher priority
}

int server::serve()
{
  customer* c = (customer* )q -> removefront();
  r -> acquire();
  current = c;                          // record as current
  sim -> schedule(c,g -> exponential(meanservice));
  c -> phase(DEPARTURE);
  return OK;
}

int server::preempt()
{
  sim -> passivate(current);            // extract departure
  r -> release();                       // and depart
  sim -> terminate(current);

  customer* c = (customer* )q -> removefront();
  r -> acquire();
  current = c;                          // record as current
  sim -> schedule(c,g -> exponential(meanservice));
  c -> phase(DEPARTURE);
  return OK;
}
```

If the front can be served we look if there is currently a customer being served (the current variable is set to NULL in the *departure* event). If so the front is served only if its queuing priority is higher. The front is then recorded as the current.

Time dependencies The arrival and service rates have been considered constant so far. This is not always realistic. The arrival rate at a supermarket shows a noticeable peak around evening for example. We can easily model this by constructing an *arrivaltime* function as follows :

```
double arrivaltime()
{
  double t = g -> exponential(MEAN1);
  if (sim -> clock() + t < TIME)        // use the appropriate mean
    return t;                           // based on the current
  else                                  // simulation time
    return (g -> exponential(MEAN2));
}
```

and we have different arrival rates at different times, if this function is used when a new arrival is scheduled.

3.4 The resource class

The class *resource* has the following interface :

```
interface resource
{
   resource(double a);          // constructor

   int report(histogram* h,double interval = 0);  // generate report

   void reset();                // reset amount left to initial

   void release(double a = 1);  // release amount left by a
   void acquire(double a = 1);  // acquire a from amount left

   int available(double a = 1); // is an amount a available ?

   int full();                  // is full ?
   int empty();                 // is empty ?
   double left();               // how much left ?
   double used();               // how much used ?
   double occupied();           // returns occupation (%).

   friend ostream& operator<<(ostream& os,resource& r);
};
```

The resource class also has a report method. When invoked a histogram is filled with samples of the occupation of the resource.

3.5 The queue class

The class *queue* has the following interface :

```
interface queue
{
   queue();                    // constructor
   virtual ~queue();           // destructor

   void reset();               // reset queue

   int reportsize(histogram* h,double interval = 0);
   int reportwaiting(histogram* h,double interval = 0,int qp = 0);
   int reportwaitingall(histogram* h,double interval = 0);

   int prepend(event* e);      // add to front
   int append(event* e);       // add to back

   event* removefront();       // remove front
   event* front();             // return front
   event* removeback();        // remove back
   event* back();              // return back

   int extract(event* e);      // extract e

   int terminatefront();       // terminate front
   int terminateback();        // terminate back

   int cancel();               // cancel specified events

   int size();                 // return size
   int empty();                // is empty ?

   friend ostream& operator<<(ostream& os,queue& q);
};
```

The reports fill the given histogram with data of the size of the queue or with data of the waiting times. The latter regardless of the queuing priorities.

4 Gathering and analyzing results

In this section we present the *histogram* and *analysis* class from the *sim* library. These classes can be used to gather and analyze the results of a simulation. Results can be gathered by taking explicit samples with the method *sample*, as is the case in the example of the dining philosophers, or by using the *report* members, as is the case in the example of the M/M/1 queue.

The class *analysis* makes it possible to analyze a histogram. It provides members for taking confidence intervals, for determining the covariance and correlation of two variables, for analyzing the behavior of the simulation and for fitting curves and probability distributions.

4.1 Gathering results with histograms

The library uses the histogram as a means to gather data. In the following sections we explain the use of the *histogram* class.

4.1.1 Histogram options

A histogram takes the widget path and an extensive list of parameters, when created, so it can be used in a very flexible way. The following options can be handled :

- -start (double) - default 0
- -columnwidth (double) - default 1
- -columns (int) - default 10
- -type (frequency,weighted,series) - default frequency
- -form (graph,cumulative,table,statistics) - default graph
- -title (string) - default histogram
- -foreground (color) - default black
- -background (color) - default grey
- -width (int) - default 100+40*(columns+2), 600, 400
- -height (int) - default 360, (columns+7)*20, 100

The first three parameters determine the start value, the width of a column and the number of columns. The type FREQUENCY is used for the distribution of the frequency data of a variable. The type WEIGHTED also takes the length of the simulated time, that the variable persisted, in account. Finally, the type SERIES is used to create time series histograms, that are used to show the fluctuation of a variable over time. The *-form* parameter is used when the histogram is printed. The other parameters are used when the histogram is printed. The defaults for the width and height are for the (cumulative) frequency graph, the table and the statistics respectively.

4.1.2 Samples and reports

Data are recorded in a histogram, when using the method *sample*. This method takes the value of the sample and its frequency, say x and y and adjusts the internal histogram data. If the type of the histogram is equal to FREQUENCY, the y value is added to the sum of frequencies of the column, that x falls in. If the type of the histogram is equal to WEIGHTED, the y value is multiplied by the time since the last sample was taken. In a histogram of type SERIES, the x-axis represents time, so the x value is ignored and the y value is added to the columns between the last column in which a sample was put and the column that represents the current simulation time. In the case of the dining philosophers, for example, a histogram of type FREQUENCY suffices. If we were interested in weighted averages, as is the case in analyzing queue sizes, the histogram should be of type WEIGHTED.

The *report* methods use the *sample* function. If these methods are used the library adds the samples.

4.1.3 The histogram class

The class *histogram* has the following interface :

```
interface histogram
{
   histogram(char* p,char* options = "");
   histogram(widget* w,char* p,char* options = "");
   virtual ~histogram();

   void sample(double x,double y = 1);     // adds a sample

   ofstream* file(char* fn);          // write samples to fn
   void closefile(char* fn);          // write samples to fn

   void reset();                  // resets histogram

   double mean();                 // return mean
   double variance();             // return variance
   double standarddeviation();    // return standard deviation
   double sumfrequencies();       // return sum of frequencies
   double minimum();              // return minimum value
   double maximum();              // return maximum value

   friend ostream& operator<<(ostream& os,histogram& h);
};
```

The *reset* method resets a histogram as if nothing was recorded yet (by setting the internal data to its original state). The simulation, generator, resource and queue classes also contain this method. The use is made clear in the following section.

4.2 Realizing accuracy goals

An increase in the number of samples made obviously gives more accurate results. Reducing the variance can then be obtained by splitting a long run in several subruns or by repeating a single run. Methods to do these are the methods of batch means and of repetition.

4.2.1 Batch means and repetition

The method of batch means splits a run into several subruns. It records the means of each subrun. The following code fragment uses this method to obtain the mean of the queue size for the M/M/1 queue.

```
histogram* h3 = new histogram(".h3","-title meansizes");
tk -> pack(h3);
tk -> update();

for (int i=0;i<100;i++)
{
```

```
    sim -> run(10000.0);
    h3 -> sample(h1 -> mean());        // sample the mean
    h1 -> reset();                     // reset for the next subrun
  }
  cout << (*h3);
```

After sampling the mean of each subrun, the histograms are reset, so the results only cover a single subrun.

The method of repeating a single run has the disadvantage that on each run we have an initial transient phase. It can be implemented as follows :

```
  histogram* h3 = new histogram(".h3","-title meansizes");
  tk -> pack(h3);
  tk -> update();

  for (int i=0;i<100;i++)
  {
    sim -> run(10000.0);
    h3 -> sample(h1 -> mean());        // sample the mean
    h1 -> reset();                     // reset objects
    r -> reset();
    q -> reset();
    sim -> reset();
    customer* c = new customer(ARRIVAL); // set up again
    sim -> schedule(c,0.0);
    s = new server();
    sim -> hold(s);
  }
  cout << (*h3);
```

In this case we also reset the resource, queue and simulation object so we have several independent runs. The simulation is set up at the beginning of each run by scheduling an arrival and making the server conditional.

The *reset* methods can also be used when running the simulation for an initial transient phase or when designing experiments.

4.3 Analyzing the results

The class *analysis* makes it possible to analyze a histogram. It provides members for taking confidence intervals, for covariance and correlation, for analyzing the behavior of the simulation, for fitting curves and probability distributions and for comparing the mean of the histogram with theoretical queue values.

4.3.1 Confidence intervals

Suppose we are not only interested in the sample mean, but also in some probability judgement of the accuracy of this mean. The usual way to do this is to construct a confidence interval for the value of the true mean. The following code shows how to take a (90%,95%,99%) confidence interval with the library

```
analysis* a = new analysis(h1);    // h1 is the histogram to be analyzed
a -> confidence("-percent 95");    // take 95% conf. int. of the mean
```

4.3.2 Covariance and correlation

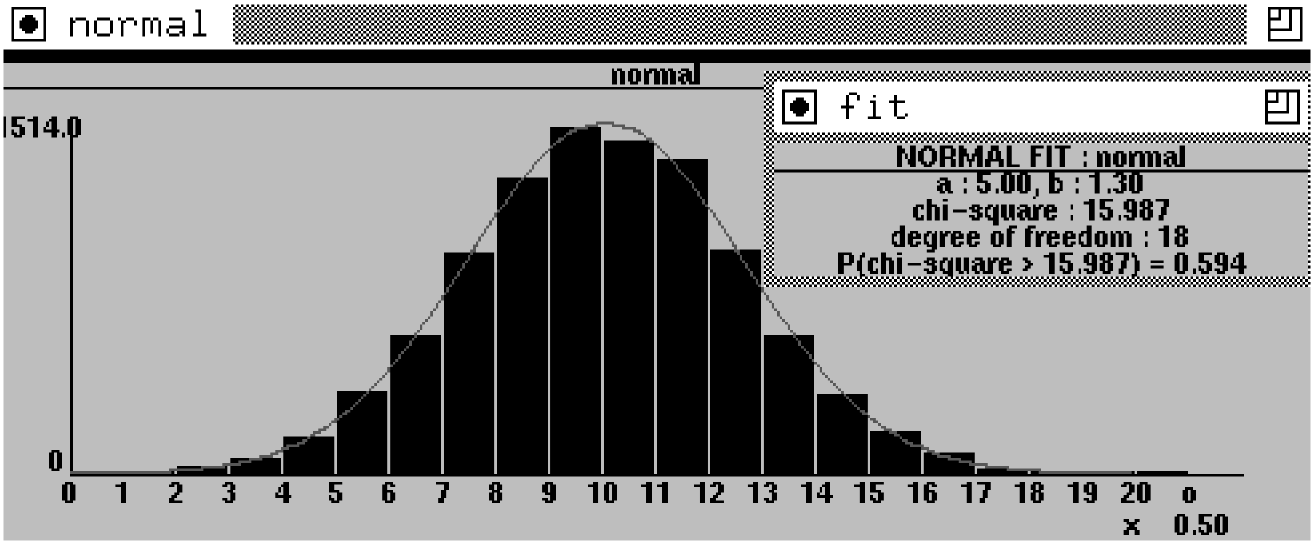
If the variables that are analyzed are dependent, i.e. their values affect each other, we sometimes need information about their dependency. A useful measure is the correlationcoefficient, that is determined from the covariance of two variables. The square of the correlation coefficient tells us what proportion of the variation of the y values can be attributed to a linear relationship with the x values. If the variables are independent (or uncorrelated) the covariance and correlation coefficient are equal to 0.

The methods assume that the samples of the two histograms are written to a file. If we have a simulation that exhibits queue swapping we could determine the correlation of the sizes of the queues as follows :

```
q1 -> reportsize(h1);              // add reports
q2 -> reportsize(h2);

h1 -> file("sizes1");              // raw samples
h2 -> file("sizes2");

sim -> run(10000.0);

analysis* a -> covariance(h2);     // covariance h1 and h2
a -> correlation(h2);              // correlation h1 and h2
```

4.3.3 Fitting curves and distributions

The library also provides methods for fitting a linear or quadratic curve and for fitting a probability distribution on the histogram data. In both cases, the belonging χ^2 statistic and probability are given. The following window was obtained by filling a histogram (with a column width of 0.5) with 10000 samples from the normal distribution with a mean of 5 and a standard deviation of 1.3. The *analysis::normal* method received these parameters, when invoked. If not, the method had calculated the a and b values from the histogram data.



4.3.4 The analysis class

The interface of the class *analysis* is :

```
interface analysis
{
  analysis(histogram* h);           // taking histogram to be analyzed
  virtual ~analysis();

  screen* format(char* p,char* options = "");
  screen* format(widget* w,char* p,char* options = "");

  double confidence(char* options = ""); // confidence interval of mean
  double confidence(histogram* h,char* options = ""); // difference

  double covariance(histogram* h2);  // covariance h and h2
  double correlation(histogram* h2)  // correlation h and h2

  double chisquaretest(double* expected); // chi-square test

  double linear();                   // linear fit, return chi-square
  double linear(double a,double b);
  double quadratic();                // quadratic fit, return chi-square
  double quadratic(double a,double b,double c);

  double uniform();                  // uniform fit, return chi-square
  double uniform(double a,double b);
  double exponential();              // exponential fit, return chi-square
  double exponential(double a);
```

```
            .
            .
            .
  double laplace();                   // laplace fit, return chi-square
  double laplace(double a);

  int behavior(histogram* h,int n);   // report behavior, gets buffer size
};
```

The *analysis* class provides a format method to create the output screen. If this function is not invoked the screen is by default toplevel. The format method takes the following options

- -foreground (color) - default black
- -background (color) - default grey
- -plot (color) - default red (for curve fitting)
- -width (int) - default 400
- -height (int) - default 100

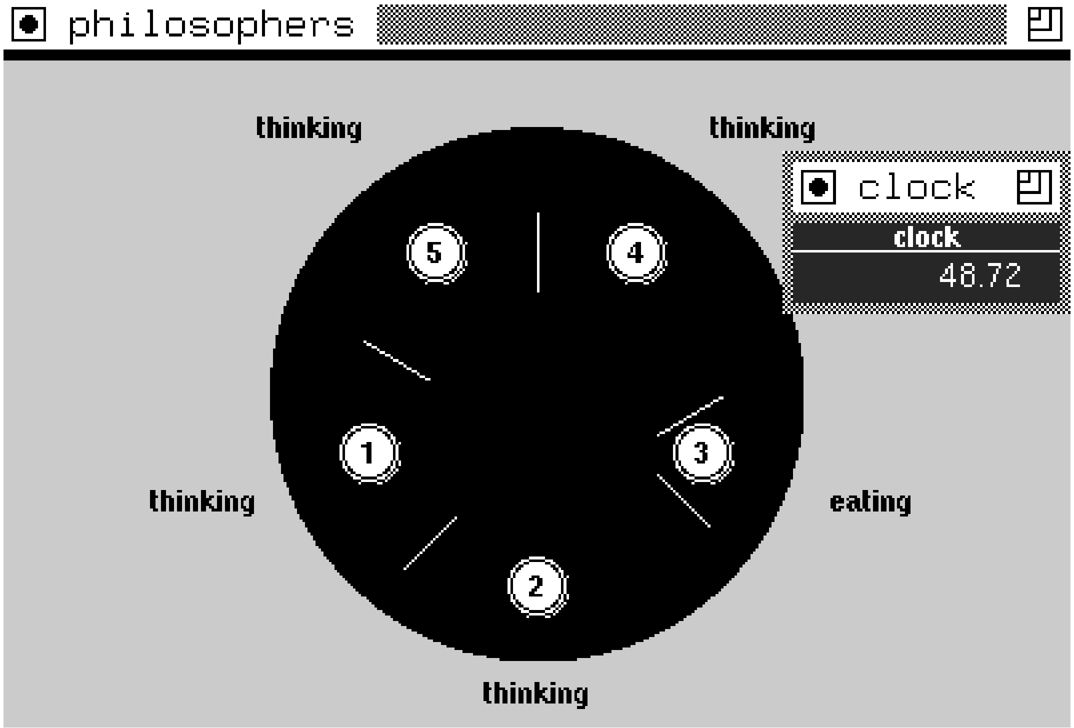
The method *behavior* can be used to analyze the behavior of the simulation, for example when detecting the steady state. It generates a report on the moving average of the given histogram.

5 Graphics

The library also offers several classes to develop a graphical interface for the simulation program. The screen widget offers the means for doing structured graphics. The other widgets (including the histogram and statistics graphics) are all based on the screen widget.

5.1 A graphical interface for the Dining Philosophers Problem

The following figure shows a display for the Dining Philosophers Problem. Its implementation is straightforward and will not be considered in detail here. It consist of the creation of the *simulationclock* and *screen* widgets. The widgets are made toplevel by giving it a *toplevel* widget when created. After that they are packed with the *kit::pack* function. The *screen* methods are used to add, remove and move the items and after a change the display is updated with the *kit::update* function.



5.2 Graphical widgets

The library contains the following widgets :

5.2.1 The screen widget

The *screen* widget is based on the *canvas* widget from the *hush* library. It also offers methods to add, move and remove items from the screen and to group items into a figure. The main difference is in the use of world coordinates, which makes it possible to scale the screen and therefore all the graphical widgets (which are derived from the *screen* widget). Adjusting the graphical widgets (and embedding it inside a screen or hypertext for example) is now done by simply adjusting the width and height of the widget (taking the world coordinates into account).

The screen widget has the following interface :

```
interface screen : public widget
{
  screen(char* p,char* options = "");
  screen(widget* w,char* p,char* options = "");
  virtual ~screen();

  // add item
  int line(int x1,int y1,int x2,int y2,char* options = "");
  int line(char* linespec,char* options = "");
```

```
    int oval(int x1,int y1,int x2,int y2,char* options = "");
    int circle(int x,int y,int r,char* options = "");
    int polygon(char* linespec,char* options = "");
    int rectangle(int x1,int y1,int x2,int y2,char* options = "");
    int square(int x,int y,int r,char* options = "");
    int bitmap(int x,int y,char* bitmap,char* options = "");
    int window(int x,int y,widget* w,char* options = "");
    int text(int x,int y,char* txt,char* options = "");

    void tag(int id,char* tg);           // group items
    char* tags(int id);

    void move(int id,int x,int y);       // move item / figure
    void move(char* tg,int x,int y);

    void del(int id);                    // remove item / figure
    void del(char* tg);

    int overlapping(int x,int y);        // which items ?
    char* items(int x,int y);

    void postscript(char* fn);           // generate postscript
  };
```

The constructor takes the path and the options of the screen. The following options are provided :

- -background (color) - default grey
- -width (int) - default 800
- -height (int) - default 600
- -worldx (int) - default 800
- -worldy (int) - default 600

Items are added and moved according to the world coordinates. Their sizes (for example the radius for a circle or the font for the text) are adjusted by a factor width/worldx. If you change the coordinates of the screen the factor width/worldx should be equal to height/worldy. Items can be grouped into a figure by specifying the same string for the items with the *tag* method. An item is identified by the integer that is returned after adding the figure. Items as well as figures can be deleted and moved. Remember that you should use the *kit::update* method if you want the change on the screen to be seen on the display.

5.2.2 The control widget

The *control* widget provides the means to adjust the behavior of the simulation. It provides means to adjust the speed, to stop and restart and to quit the simulation. Its interface :

```
  interface control : public screen
  {
```

```
  control(char* p,char* options = "");
  control(widget* w,char* p,char* options = "");
  virtual ~control();
  void factor(double fct);        // adjust speed factor
  void reset();                   // to initial state
};
```

The following options can be handled :

- -background (color) - default grey
- -foreground (color) - default white
- -widgets (color) - default blue
- -factor (double) - default 1
- -width (int) - default 180
- -height (int) - default 120

When created and packed a screen appears with a button to start and stop the simulation, a button to quit the simulation and a scale widget to adjust the speed. The color of these widgets can be adjusted by chancing the *widgets* option. The *-factor* option is the delay factor. The execution of an event is delayed by factor times the inter-activation time (in seconds) times (100-x)/100, where x is the current speed. Again, when adjusting the width and height you should take into account that width/180 should be equal to height/120.

5.2.3 The simulationclock widget

When created and packed the *simulationclock* widget shows the current simulation time. Its interface :

```
interface simulationclock : public screen
{
  simulationclock(char* p,char* options = "");
  simulationclock(widget* w,char* p,char* options = "");
  virtual ~simulationclock();
};
```

The following options can be handled :

- -background (color) - default blue
- -foreground (color) - default white
- -width (int) - default 200
- -height (int) - default 60

5.2.4 The query widget

The *query* widget can be used when user-input is required. Its interface :

```
interface query : public screen
{
  query(char* p,char* options = "");
  query(widget* w,char* p,char* options = "");
  virtual ~query();
  char* get();         // returns NULL or value
};
```

fig 6.5. The interface of the query widget

The following options can be handled :

- -background (color) - default lightblue
- -foreground (color) - default black
- -width (int) - default 160
- -height (int) - default 80
- -text (string) - default input

The query consists of a string that appears at the head of the query and which can be set by specifying the *-text* option. The additional input entry can be activated by clicking on it. You can then enter a value which is set when clicking on the *ok* button. This value is then returned by the *get* method. The general use of the *query* widget is create and pack the widget and stay in a while-loop until the *get* method returns a value not equal to NULL (i.e. when a value was entered and the *ok* button was pressed).

6 Conclusions

The library *sim* extends C++, so that the resulting language exhibits the same functionality as a widely used language as *simula*. *Sim* provides a convenient, yet flexible way to write simulation programs in C++. It offers in addition the advantages of C++ :

- efficiency - C++ compilers generate efficient code.
- features - C++ provides advanced object-oriented features. These features affect the structure of a program in a positive way.
- generality - C++ is known and used in other areas as simulation.
- the means to capture complexity - we wrote for example a distributed interactive simulation using the library.

The *report* and *reset* methods, the *analysis* class and the graph ical features are original and are extensively used.

Acknowledgements We would like to thank the students and staff of the Free University for their comments on the draft of this paper.

References

[Birtwistle 86] G.M. Birtwistle : Discrete Event Modelling on Simula, Macmillan 1986.

[Eliens 95] A. Eliens : Hush - a C++ API for Tcl/Tk, The X resource 1995.

[Gross 85] D. Gross, C.M. Harris : Fundamentals of queuing theory, John Wiley & Sons 1985.

[Kalvelagen 90] E. Kalvelagen, H.C. Tijms : Exploring Operations Research & Statistics in the Micro Lab, Prentice Hall 1990.

[Pooley 87] R. J. Pooley : An introduction to programming in Simula, Blackwell Scientific Publications 1987

[Stroustrup 91] B. Stroustrup : The C++ Programming Language, Addison Wesley 1991.

[Watkins 93] K. Watkins : Discrete Event Simulation in C, McGraw-Hill 1993.

[Wichmann 87] B.A. Wichmann, I.D. Hill : Building a random-number generator, Programming insight 1987

A The generator and some tests

In this section we present the generator and some tests. The generator uses the linear congruential method as described in [Wichmann 87] to produce its random numbers (section A.1.). The tests cover the randomness of the random number stream (section A.2.), the quality of the distributions that transform the stream (section A.3.) and the quality of the event scheduling routine (section A.4.).

A.1 The linear congruential generator

The linear congruential generator consists of the following four numbers :

- m, the modulus, $m > 0$
- a, the multiplier, $0 \leq a < m$
- c, the increment, $0 \leq c < m$
- X_0, the seed, $0 \leq X_0 < m$

from which a sequence of random numbers $< X_n >, n \geq 0$ can be generated by setting

$$X_n = (aX_{n-1} + c) \text{ mod m}$$

These numbers should be chosen in such a way, that the random number stream provides a long cycle time and has good statistical properties. The library combines three generators :

- $X_n = 171X_{n-1} \text{ mod } 30269$
- $Y_n = 172Y_{n-1} \text{ mod } 30307$

- $Z_n = 170 Z_{n-1} \bmod 30323$

The random number then comes from the formula :

- $temp = X_n/30629.0 + Y_n/30307.0 + Z_n/30323.0$
- $random = temp - (int)temp$

The implementation in the library (the *generator::random* method) rewrites these formulas to ensure compatibility of this method. This generator has a period of $6.95.10^{12}$, which means that you won't get a cycle. Furthermore, it has passed several statistical tests.

A.2 Tests on the random number stream

We wrote the following tests for the random number stream :

- *the Uniformity Test.* The interval is split up into 10 intervals. For randomness we would expect the counts in each interval to be equal. A χ^2 test is performed with each member of the expected array equal to the number of random fractions divided by 10.

  We also wrote two variations of this test. The first one groups the random numbers into pairs, the second one into triples. The expected values of each pair and triple are then the number of random fractions divided by 2*100 and by 3*125 (5 intervals) respectively.

- *The Poker Test.* In this test we multiply the random fractions with 10 and cast them to integers. Furthermore we group the sequence of integers into sequences of five consecutive integers. The frequency of the following categories is recorded :
  - 1 - All of the same or two different integers in a group.
  - 2 - Three different integers in a group.
  - 3 - Four different integers in a group.
  - 4 - Five different integers in a group.

  A χ^2 test is performed with the following expected values for each category (n is the number of fractions):
  - 1 - 0.0136 n.
  - 2 - 0.1800 n.
  - 3 - 0.5040 n.
  - 4 - 0.3024 n.

- *the Gap Test.* An interval between a and b is defined ($0 < a < b < 1$). Each random number is replaced with 0 if outside and with 1 if inside the interval. We are interested in the lengths of the gaps. The expected value of a gap of length k is :

$$(b-a)(1-(b-a))^k n$$

- *the Coupon Collector Test.* In this test we divide the interval [0,1) into five intervals [0,0.2),..,[0.8,1). The number of random fractions to get at least a count of 1 in each interval is counted. We define 16 categories as follows :
  - 1 - sequence of 5.

– 2 - sequence of 6.

– ..

– 15 - sequence of 19.

– 16 - sequence of 20 or more.

The expected values of each category are calculated as :

$$p_i n, 1 \leq i \leq 16$$

with n the number of fractions and p_i from the following sequence of probabilities : 0.03840000, 0.07680000, 0.09984000, 0.10752000, 0.10450944, 0.09547776, 0.083816 45, 0.07163904, 0.06011299, 0.04979157, 0.04086200, 0.03331007, 0.02702163, 0.02 184196, 0.01760857, 0.07144851

In all cases we fill an array with the expected values for the taken test and perform a χ^2 test with the observed values with the method from the *analysis* class. The hypothesis of randomness is rejected if the probability is less then 0.05 (no randomness) or more then 0.95 (fabricated) in 95% or more of the experiments.

A.3 A test on the distributions

The distributions that transform the random number stream are tested with the methods from the *analysis* class. We fill a histogram with values taken from the specified probability distribution with the given parameters and fit the theoretical values on the histogram with these parameters. The hypothesis of randomness is rejected if the χ^2 probability is less then 0.05 or more then 0.95 in 95% or more of the experiments .

A.4 A test on the event scheduling strategy

The event scheduling strategy is tested by running a general queuing simulation. The arrival and service pattern (M or D), the number of service points and the arrival and service rate are the input for the simulation. The results are compared with the analytical values with the methods of the *analysis* class. If these values lie within the 95% interval of the simulation results on 95% or more of the experiments the event scheduling strategy passed the test. For example, if we look at the analytical values of the M/M/1 queue, with an arrival rate of 3 and a service rate of 4, the mean of the queue size, according to [Gross 85], is equal to

$$\frac{\lambda^2}{\mu^2 - \lambda\mu}$$

where λ is the number of arriving customers per time unit and μ is the service rate per time unit. So the queue size in our M/M/1 queue is equal to $\frac{9}{4}$.

The mean time spent queuing is

$$\frac{\lambda}{\mu^2 - \lambda\mu}$$

or in our M/M/1 queue $\frac{3}{4}$. These values do indeed agree with the results of the simulation program of the M/M/1 queue, taking the confidence intervals into account.

B Probability distributions

The *generator* and *analysis* classes use the following probability distributions as described in [Watkins 93] and [Kalvelagen 90]. Some use the Γ function, which is defined as :

$$\Gamma(\alpha) = \int_0^\infty z^\alpha e^{-z} dz$$

The distributions are :

- *the uniform distribution*, this distribution represents the situation whereby a random number can take values within a finite range $[\alpha, \beta]$ with equal probability. It is characterized by the following function :

  $$\begin{array}{ll} \frac{1}{\beta-\alpha}, & \text{for } \alpha \leq x \leq \beta \\ 0, & \text{otherwise} \end{array}$$

  The method in the library takes the upper and lower limit as parameters.

- *the exponential distribution*, takes the non-negative mean α as parameter. Its function :

  $$\begin{array}{ll} \frac{1}{\alpha}e^{-\frac{x}{\alpha}}, & \text{for } x \geq 0 \\ 0, & \text{otherwise} \end{array}$$

  This function is used to model purely random events such as the time between arrivals of a customer.

- *the hyperexponential distribution*, takes the non-negative means α and β of two exponential distributions as parameters. It then generates a variate from the first distribution with chance p and a variate of the second with chance *1-p*. Its function :

  $$\begin{array}{ll} \frac{p}{\alpha}e^{-\frac{x}{\alpha}} \frac{1-p}{\beta}e^{-\frac{x}{\beta}}, & \text{for } x \geq 0 \\ 0, & \text{otherwise} \end{array}$$

- *the normal distribution*, the first parameter to this distribution is the mean α, the second is the standard deviation β. It is characterized by the following probability distribution :

  $$\frac{1}{\sqrt{2\pi}\beta}e^{-(x-\alpha)^2/2\beta^2}, \quad \text{for } -\infty \leq x \leq \infty$$

  This function is used for the modeling of the additive effect of several independent trials.

- *the lognormal distribution*, It is calculated as the logarithm of the normal distribution with α as mean and β as standard deviation. Its probability function :

  $$\begin{array}{ll} \frac{1}{\beta x\sqrt{2\pi}}e^{-(lnx-\alpha)^2/2\beta^2}, & \text{for } x \geq 0 \\ 0, & \text{otherwise} \end{array}$$

- *the erlang distribution*. It is used to model service times in a queuing system that has n servers where each server has a mean service time of α. The overall service time of this system is given by the *erlang* distribution. It is characterized by the following distribution :

$$\begin{array}{ll} \frac{\alpha(\alpha x)^{n-1}e^{-\alpha x}}{(n-1)!}, & \text{for } x \geq 0 \\ 0, & \text{otherwise} \end{array}$$

- *the gamma distribution*, this distribution is parameterized by two values α and β, both greater then 0. It has the following probability distribution :

$$\begin{array}{ll} \frac{\beta^{\alpha} x^{\alpha-1} e^{-x\beta}}{\Gamma(\alpha)}, & \text{for } x \geq 0 \\ 0, & \text{otherwise} \end{array}$$

- *the poisson distribution*, gets the mean α as parameter. This function is used to model the number of occurrences, such as the number of component failures, over a given period of time. It is characterized by the following distribution :

$$\begin{array}{ll} \frac{\alpha^{x}}{x!} e^{-\alpha}, & \text{for } x = 0, 1, ..., \infty \\ 0, & \text{otherwise} \end{array}$$

- *the geometric distribution*, this distribution is characterized by p, the chance on success. It is used to model the number of occurrences between two significant events as the number of failures to acquire a particular resource until the first success. Its distribution :

$$\begin{array}{ll} (1-p)^{x-1} p, & \text{for } x = 0, 1, ..., \infty \\ 0, & \text{otherwise} \end{array}$$

- *the hypergeometric distribution.* Suppose that a sample of size n is to be randomly chosen (without replacements) from an urn containing m balls of which mp are white and *m(1-p)* are black. The chance of x white balls selected follows the hypergeometric distribution and is equal to :

$$\begin{array}{ll} \frac{\binom{mp}{x}\binom{m(1-p)}{n-x}}{\binom{m}{n}}, & \text{for } x = 0, 1, ..., min(mp, n) \\ 0, & \text{otherwise} \end{array}$$

- *the weibull distribution*, it gets α and β as parameters, both greater then 0. It is used for reliability measures such as the lifetime of components. It is characterized by the following distribution function :

$$\begin{array}{ll} \frac{\beta}{\alpha} x^{\beta-1} e^{-x^{\beta}/\alpha}, & \text{for } x \geq 0 \\ 0, & \text{otherwise} \end{array}$$

- *the binomial distribution*, gets as parameters p, the chance on a success and n, the number of trials. It is used to model the number of successes in n independent Bernoulli trials. It has the following probability distribution :

$$\begin{array}{ll} \binom{n}{x} p^x(1-p)^{n-x}, & \text{for } x = 0, 1, ..., n \\ 0, & \text{otherwise} \end{array}$$

- *the negative binomial distribution*, it is parameterized by p, the chance on a success, and n, the number of successes. The negative binomial distribution returns the number of failures before the n^{th} success. You can think of the number of retransmissions of a file. The negative binomial distribution is characterized by the following function :

$$\begin{array}{ll} \binom{n+x-1}{x} p^n(1-p)^{x}, & \text{for } x = 0, 1, ..., \infty \\ 0, & \text{otherwise} \end{array}$$

- *the beta distribution*, this probability function is characterized by the two parameters α and β. It is used to model random proportions as the fraction of customers that could not be served. It has the following distribution :

$$\begin{array}{ll} \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}, & \text{for } 0 \leq x \leq 1 \\ 0, & \text{otherwise} \end{array}$$

- *the laplace distribution.* The laplace distribution looks like the exponential distribution with a mirror image in the y-axis. It has the following probability function :

$$\frac{1}{2\lambda} e^{-|x|/\lambda}, \quad \text{for } -\infty \leq x \leq \infty$$

- *the χ^2 distribution*, with n, the degrees of freedom as parameter. It has as probability distribution function :

$$\begin{array}{ll} \frac{x^{n/2-1} e^{-x/2}}{2^{n/2}\Gamma(n/2)}, & \text{for } x \geq 0 \\ 0, & \text{otherwise} \end{array}$$

  The χ^2 distribution is a special case of the gamma distribution with $\alpha = n/2$ and $\beta = 1/2$.

- *the student's t distribution*, it also gets the degrees of freedom as parameter. If n is large enough (≥ 30), it can be approximated by the standard normal distribution. It has as function :

$$\frac{\Gamma(n/2+1/2)}{\sqrt{n\pi}\Gamma(n/2)} \frac{1}{(1+x^2/n)^{(n+1)/2}}, \quad \text{for } -\infty \leq x \leq \infty$$

- *the F distribution*, returns a value of the F distribution, which is often used for testing variances. The distribution takes two positive integers as arguments, defining the degrees of freedom of the variances being tested. It has a probability distribution function :

$$\begin{array}{ll} \frac{\Gamma((n+m)/2)(n/m)^{n/2}}{\Gamma(n/2)\Gamma(m/2)} x^{(n/2)-1}(1+\frac{nx}{m})^{-(n+m)/2}, & \text{for } x \geq 0 \\ 0, & \text{otherwise} \end{array}$$

- *the triangular distribution*, it gets the mode of the distribution as parameter. Dependent on the mode m the distribution is in the function takes on various triangular shapes. If the mode is equal to 0.5, the triangular distribution is symmetrical. It has the following probability function :

$$\begin{array}{ll} \frac{2}{m}x, & \text{for } 0 \leq x < m < 1 \\ \frac{2}{1-m}(1-x), & \text{for } m \leq x \leq 1 \\ 0, & \text{otherwise} \end{array}$$