jDisco
- a Java package for combined discrete and continuous simulation

Keld Helsgaun
E-mail: keld@ruc.dk

Department of Computer Science
Roskilde University
DK-4000 Roskilde, Denmark

Abstract

This report describes jDisco, a Java package for the simulation of systems that contains both continuous and discrete-event processes. The package is described mainly from the user's point of view; its implementation is only sketched.

1. Introduction

Most simulation languages belong to one of two types, *discrete* or *continuous.*

The discrete simulation languages stress the viewpoint that state variables interact discretely, that is, instantaneously and only at prescribed points in time (event times).

In contrast, the continuous simulation languages stress the viewpoint that state variables interact continuously – a viewpoint that leads to models expressed by differential equations.

However, there are systems containing important interactions between discrete and continuous subsystems so that neither type of language is adequate by itself for simulation.

Systems requiring a combined continuous and discrete description are typically systems in which discrete actions are superimposed on continuous subsystems. Many such systems are found in industry. For example, consider a steel-making process. Steel ingots with different arrival times are heated to a desired temperature in a furnace. The heating of each ingot is a continuous process, while arrivals and departures of ingots are discrete events.

Simulation of industrial systems, however, is not the only area in which a combined simulation approach may be appropriate. In fact, there exists a diversity of systems involving both continuous and discrete phenomena.

To name a few, first consider the treatment of diabetes. The biochemical reactions are continuous processes. Injections of insulin and ingestion of food may be considered as discrete events.

Another example is automobile traffic. The vehicle dynamics constitute the system's continuous part, and the queuing and driver decisions the discrete part.

Lastly, consider the human brain. The biochemical reactions are continuous processes, whereas the triggering of a neural impulse is a discrete event.

To describe such mixed systems, it may be necessary to use a language that combines the two types of languages, a so-called *combined-system simulation language.*

The Java package jDisco provides such a language. It is an elaboration of an earlier SIMULA class called DISCO [2]. It generalizes the process concept of the programming language SIMULA to include both continuous and discrete processes.

2. Modelling philosophy

A system is conceived of as a collection of processes, which undergo active and inactive phases and whose actions and interactions comprise the behaviour of the system.

In jDisco a distinction is made between two types of processes: *continuous processes* and *discrete processes*. Continuous processes undergo active phases during time intervals, and cause continuous changes of state. In contrast, discrete processes have instantaneous active phases, called *events*, and cause discrete changes in the state of the system. The events of discrete processes are separated by periods of inactivity, during which continuous processes may be active.

Any process may be created, activated, deactivated, or removed from the system at any time. However, jDisco permits more than coexistence of separate processes. It allows processes to communicate in a completely general way. Any process may reference and modify any variable in any other process and may affect delimiting and sequencing of active phases.

3. Basic concepts

jDisco is a true extension of javaSimulation [3], a Java package for discrete event simulation, making the concepts of the latter available to the user. Thus, class `Process` and the event scheduling methods (`activate`, `hold`, `passivate`, etc.) can readily be used in describing discrete processes.

In order to enable the description of systems that also involve continuous processes, jDisco provides the following additional concepts:

```
class Variable
class Continuous
void waitUntil(Condition cond)
```

3.1 Class `Variable`

Outline:

```
public class Variable {
    public Variable(double initialState);

    public Variable start();
    public void stop();

    public double state, rate;
}
```

Objects of class `Variable` can be used to represent state variables that vary according to ordinary first-order differential equations. The value of such a variable is denoted by `state`, while `rate` denotes its derivative with respect to time. The initial value of `state` is passed as a parameter on object generation.

After its `start` method is called, a `Variable` object becomes *active*, that is to say, its `state` undergoes continuous change between discrete events. The value of `state` is changed according to the value of `rate`, as computed by the active continuous processes. The active phase will cease when the object's `stop` method is called.

Example:

An object of class `Variable`, say `x`, may be generated with an initial `state` value of 3.14 by the statement

```
Variable x = new Variable(3.14);
```

The variable may be started by calling `x.start()`.

3.2 Class `Continuous`

Outline:

```
public abstract class Continuous {
    protected abstract void derivatives();

    public Continuous start();
    public void stop();
}
```

Class `Continuous` can be used to describe continuous processes defined by ordinary differential equations. The description is given in one ore more subclasses that compute derivatives of state variables.

After its `start` method is called, a `Continuous` object becomes *active*, that is to say, its user-defined `derivatives` method is executed “continuously”. This active phase will cease when the object’s `stop` method is called.

Example:

A continuous process defined by the differential equations

$$\frac{dx}{dt} = (A + By)x$$

$$\frac{dy}{dt} = (C + Dx)y$$

can be described by the following declaration

```
class Dynamics extends Continuous {
    public void derivatives() {
        x.rate = (A + B * y.state) * x.state;
        y.rate = (C + D * x.state) * y.state;
    }
}
```

where `x` and `y` are `Variable`-objects.

An object of this class, say `evolution`, is generated by the statement

```
Continuous evolution = new Dynamics();
```

and is started by calling `evolution.start()`.

3.3 Class `Process`

Outline:

```
public abstract class Process {
    protected abstract void actions();

    public static double time();
    public static void activate(Process p);
    public static void hold(double t);
    public static void passivate();
    public static void wait(Head q);
    public static void waitUntil(Condition cond);

    public double dtMin, dtMax;
    public double maxAbsError, maxRelError;
}
```

```
public interface Condition {
    boolean test();
}
```

The discrete processes of a system are described by means of class `Process`. The `actions` method is used to describe the lifecycle of such processes. A process may be suspended temporarily and may be resumed later from where it left off.

The `activate` method is used to make a specified process start executing its actions. The `hold` method suspends the execution of the calling process for a specified period of time. The `passivate` method suspends the execution of the calling process for an unknown period of time. Its execution may later be resumed by calling `activate` with the process as argument. The `wait` method suspends the calling process and adds it to a queue. For a more thorough description of these scheduling methods, see [3].

The method `waitUntil` can be used to schedule a discrete event to occur as soon as a prescribed system state is reached. Such an event is called a *state-event*, in contrast to a *time-event* which is an event scheduled to occur at a specified point in time.

The call `waitUntil(cond)`, where `cond` is a `Condition` object, causes the calling discrete process to become passive until the `test` method of `cond` returns `true`.

Example:

Typically `waitUntil` is used to schedule an event to occur when a state variable crosses a prescribed threshold.

By calling

```
      waitUntil(new Condition() {
          public boolean test() {
              return x.state > 100;
          }
      });
```

the active discrete process postpones its actions until `x`’s `state` becomes greater than 100.

The state-condition could have been more complex, as for example in the call

```
      waitUntil(new Condition() {
          public boolean test() {
              return x.state > 100 || y.rate < 0;
          }
      });
```

In fact, a state-condition may be of arbitrary complexity.

Between the event times the state of the model is automatically updated in steps of varying size.

`dtMin` and `dtMax` are used to specify the minimum and maximum allowable step-sizes.

`maxAbsError` and `maxRelError` can be used to specify the maximum absolute and maximum relative error allowed in updating the `state` values of the active `Variable` objects.

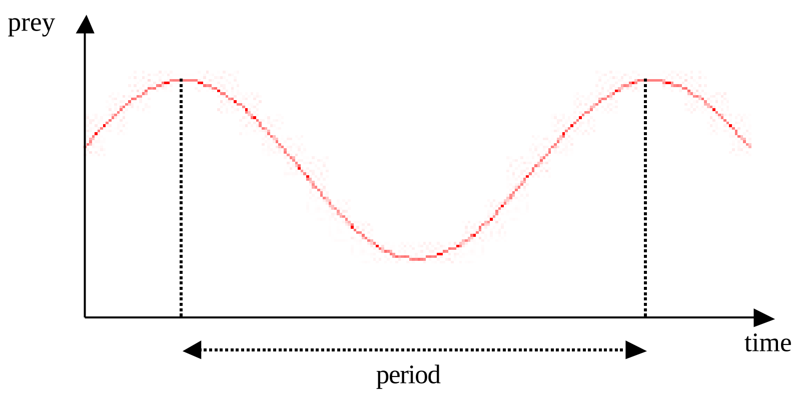
3.4 A small example

Below is given a complete jDisco program that simulates a predator-prey system.

An object of class `PredatorPreySystem` governs the simulation (line 24). It starts two state variables (lines 8-11) and one continuous process (line 12), and finally suspends itself for a period of 100 time units, the simulation period (line 13).

```
 1. import jDisco.*;
 2. import jDisco.Process;

 3. public class PredatorPreySystem extends Process {
 4.     Variable predator, prey;

 5.     public void actions() {
 6.         dtMin = 1.0e-5; dtMax = 1;
 7.         maxAbsError = 0; maxRelError = 1.0e-5;
 8.         predator  = new Variable(1000);
 9.         predator.start();
10.         prey = new Variable(100000);
11.         prey.start();
12.         new Dynamics().start();
13.         hold(100);
14.     }

15.     class Dynamics extends Continuous {
16.         public void derivatives()
17.             predator.rate = (-0.3 + 3.0e-7 * prey.state)
18.                                 * predator.state;
19.             prey.rate = (0.3 – 3.0e-4 * predator.state)
20.                             * prey.state;
21.         }
22.     }

23.     public static void main(String[] args) {
24.         activate(new PredatorPreySystem());
25.     }
26. }
```

The system is known to be cyclic. The period can be determined through the `waitUntil` method by computing the time interval between two consecutive maximum points of a state variable, say `prey`.



In order to achieve this the `hold`-statement of line 13 may be replaced by the following:

```
// *** Find first maximum ***
waitUntil(new Condition() {
    public boolean test() {
        return prey.rate > 0;
    }
});
waitUntil(new Condition() {
    public boolean test() {
        return prey.rate <= 0;
    }
})
// *** First maximum found ***
double cycleStartTime = time();

// *** Find second maximum ***
waitUntil(new Condition() {
    public boolean test() {
        return prey.rate > 0;
    }
});
waitUntil(new Condition() {
    public boolean test() {
        return prey.rate <= 0;
    }
});
// *** Second maximum found ***
System.out.println("period = " +
                   time() - cycleStartTime);
```

4. Examples

The descriptive power of jDisco is best appreciated through examples. This section presents six examples illustrating jDisco's capability for combined modelling.

4.1 Three-stage rocket

This example has been taken from [7] and has to do with the launch of a three-stage rocket from the earth's surface.

The model has both continuous and discrete elements. During the rocket's flight well-known physical laws govern its motion, and its mass decreases continuously as a result of the expulsion of burnt fuel. However, when a stage separates from the rest of the rocket, an instantaneous change of the rocket's mass and acceleration takes place.

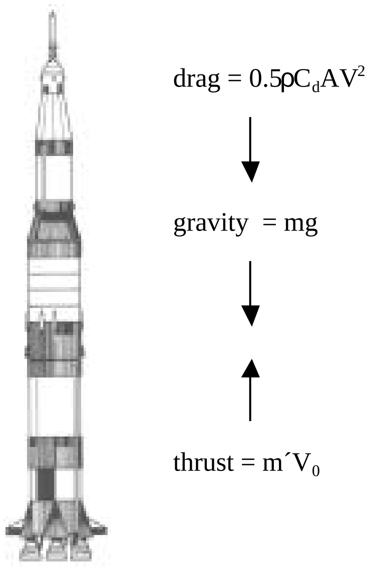


Figure 4.1 *Saturn three-stage rocket*

A complete simulation program is given on the next page.

The continuous motion of the rocket is described in class `RocketMotion`. (lines 7-20).

The change in mass takes place at a constant rate, `-massFlow` (line 9).

There are three forces acting upon the rocket: `thrust`, `drag` and `gravity`. (lines 10-15).

`thrust` is the force generated by the expulsion of burnt fuel, `drag` is the air resistance, and `gravity` is the earth's gravitational attraction on the rocket.

From Newton's second law of motion the rocket's acceleration can be determined by summing these three forces and dividing by the rocket's mass (lines 16-17). The change of rate of altitude is equal to the rocket's velocity (line 18).

The discrete changes (that is, the separation of stages) are described in the `actions` method (lines 21-39) of the discrete main process, an object class `ThreeStageRocket`.

The rocket's launch begins when the three `Variable`-objects `mass`, `velocity` and `altitude` are generated and `start`ed together with an object of class `RocketMotion` (lines 26-29).

Each time a stage separates, discrete changes in `mass`, `massFlow`, `flowVelocity` and `area` take place (lines 32-33 and lines 36-37). These events are scheduled to occur by the method `hold` (line 30 and line 34).

```
import jDisco.*;
import jDisco.Process;

public class ThreeStageRocket extends Process {
    Variable mass, velocity, altitude;
    double thrust, drag, gravity, massFlow,
           flowVelocity, area;

    class RocketMotion extends Continuous {
        public void derivatives() {
            mass.rate = -massFlow;
            thrust = massFlow * flowVelocity;
            drag = area * 0.00119 *
                   Math.exp(-altitude.state / 24000) *
                   Math.pow(velocity.state, 2);
            gravity = mass.state * 32.17 /
              Math.pow(1 + altitude.state / 20908800, 2);
            velocity.rate = (thrust - drag - gravity) /
                            mass.state;
            altitude.rate = velocity.state;
        }
    }

    public void actions() {
        dtMin = 0.00001; dtMax = 100;
        maxAbsError = maxRelError = 0.00001;

        // first stage
        massFlow = 930; flowVelocity = 8060; area = 510;
        mass     = new Variable(189162).start();
        velocity = new Variable(0).start();
        altitude = new Variable(0).start();
        new RocketMotion().start();
        hold(150);

        // second stage
        mass.state = 40342; massFlow = 81.49;
        flowVelocity = 13805; area = 460;
        hold(359);

        // third stage
        mass.state =  8137; massFlow = 14.75;
        flowVelocity = 15250; area = 360;
        hold(479);
    }

    public static void main(String[] args) {
        activate(new ThreeStageRocket());
    }
}
```

This example illustrates one type of continuous-discrete interaction, namely discrete changes of “continuous” variables.

Two other types may also be modelled with jDisco.

The triggering of a discrete event as a consequence of a state-condition is one type. This type can be expressed using the method `waitUntil`. In the example it is known at which times stage separation occurs, so the *imperative* scheduling method `hold` is used to schedule these events. If this were not the case, for example if stage separation were dependent on altitude, then the *interrogative* scheduling method `waitUntil` could be used instead, e.g.

```
waitUntil(new Condition() {
    public boolean test() {
        return altitude.state > 25000;
    }
});
```

The other type of continuous-discrete interaction allows for dynamic starting and stopping of continuous processes. This capability makes possible simulation of systems in which the differential equations themselves vary with time. In the example we could have represented the rocket’s motion by three continuous processes, one for each phase of motion. A stage separation would then cause the current active continuous process to be stopped and replaced by the continuous process that corresponds to the next phase.

4.2 Fire-fighting

This example is a simulation of a fire station. The model is formulated by R. W. Sierenberg [6].

A small city owns a fire station with three fire engines. Fire alarms are given randomly at exponentially distributed intervals with a mean of six hours.

Each house on fire contains a certain amount of inflammable material. When a fire is discovered, it already has a certain size. The fire increases with a rate that is proportional to its size as no extinguishing activity takes place.

At the moment an alarm is given, one fire engine, if it is available, will be sent to the fire. When a fire engine reaches the fire and finds out that its capacity is smaller than the rate with which the fire increases, it will request assistance by sending a second alarm for the same fire. The fire engine returns to the fire station after the fire is put out or when all inflammable material is consumed.

A program for simulating the fire station is shown on the next pages. The program should be fairly self-explanatory.

Class `Burning` (lines 20-30) describes, through differential equations, the continuous process associated with a house on fire.

Class `HouseOnFire` (lines 31-59) describes the discrete events associated with a house on fire: alarm call (line 45) and fire termination (lines 53-55).

Class `FireEngine` (lines 66-95) defines the fire engines.

An `Incendiary`-object (lines 96-103) sets houses on fire.

The simulation period is one month (line 17).

```
import jDisco.*;
import jDisco.Process;

public class FireFighting extends Process {
    Head fireStation = new Head();
    Head alarmQ = new Head();
    static final int oneHour = 60;
    static final int oneMonth = 30 * 24 * oneHour;
    Random rand = new Random(54521);
    Histogram percDamage =
        new Histogram("Perc. damage", 0, 100, 20);

    public void actions() {
        dtMin = 0.0001; dtMax = oneHour;
        maxRelError = maxAbsError = 0.0001;
        for (int i = 1; i <= 4; i++)
            new FireEngine(10).into(fireStation);
        activate(new Incendiary());
        hold(oneMonth);
        percDamage.report();
    }

    class Burning extends Continuous {
        Burning(HouseOnFire house) {
            this.house = house;
        }

        public void derivatives() {
            house.size.rate = house.c * house.size.state
                            – house.extinguishRate;
            house.damage.rate = house.size.state;
        }

        HouseOnFire house;
    }
```

```
class HouseOnFire extends Process {
    double c, material, extinguishRate, travelTime;
    Variable size, damage;
    Burning fire;

    public void actions() {
        c = rand.uniform(0.01, 0.06);
        material = rand.uniform(100, 600);
        travelTime = rand.uniform(5, 15);
        size = new Variable(rand.uniform(0, 5));
        size.start();
        damage = new Variable(size.state);
        damage.start();
        fire = new Burning(this);
        fire.start();
        new Alarm(this).into(alarmQ);
        activate((Process) fireStation.first());
        waitUntil(new Condition() {
            public boolean test() {
                return size.state <= 0 ||
                       damage.state >= material;
           }
        });
        fire.stop();
        damage.stop();
        size.stop();
        percDamage.update(damage.state * 100 /
                          material);
    }
}

class Alarm extends Link {
    Alarm(HouseOnFire house) {
        this.house = house;
    }

    HouseOnFire house;
}
```

```
    class FireEngine extends Process {
        FireEngine(double capacity) {
            this.capacity = capacity;
        }

        public void actions() {
            while (true) {
                Alarm alarm = (Alarm) alarmQ.first();
                if (alarm != null) {
                    alarm.out();
                    HouseOnFire house = alarm.house;
                    if (house.fire.isActive()) {
                        out();
                        hold(house.travelTime);
                        if (house.size.rate > capacity) {
                            alarm.into(alarmQ);
                            activate((Process)
                                    fireStation.first());
                        }
                        house.extinguishRate += capacity;
                        while (house.fire.isActive())
                            hold(5);
                        hold(house.travelTime);
                        into(fireStation);
                    }
                } else
                    passivate();
            }
        }

        double capacity;
    }

    public class Incendiary extends Process {
        public void actions() {
            while (true) {
                hold(rand.negexp(1 / (6.0 * oneHour)));
                activate(new HouseOnFire());
            }
       }
   }

    public static void main(String[] args) {
        activate(new FireFighting());
    }
}
```

The program reports the percentage of damage using jDisco’s class `Histogram`:

```
title       /  (re)set/   obs/  average/est.st.dv/  minimum/  maximum/    conf./
Perc. damage     0.000    138    13.817    13.257     0.062    73.831     2.240

cell/lower lim/    n/   freq/  cum %
                                      |----------------------------------------
   0 -infinity     0    0.00    0.00  |
   1     0.000    16    0.12   11.59  |*****************
   2     5.000    38    0.28   39.13  |****************************************
   3    10.000    31    0.22   61.59  |*********************************
   4    15.000    17    0.12   73.91  |******************
   5    20.000    13    0.09   83.33  |**************
   6    25.000     4    0.03   86.23  |****
   7    30.000     4    0.03   89.13  |****
   8    35.000     6    0.04   93.48  |******
   9    40.000     2    0.01   94.93  |**
  10    45.000     4    0.03   97.83  |****
  11    50.000     1    0.01   98.55  |*
  12    55.000     0    0.00   98.55  |
  13    60.000     0    0.00   98.55  |
  14    65.000     0    0.00   98.55  |
  15    70.000     0    0.00   98.55  |
  16    75.000     2    0.01  100.00  |**
  17    80.000     0    0.00  100.00  |
  18    85.000     0    0.00  100.00  |
  19    90.000     0    0.00  100.00  |
                                          **rest of table empty**
                                      |----------------------------------------
```

4.3 Chemical reactor system

This simulation of a chemical reaction process is an example used by Hurst and Pritsker [4] to illustrate GASP IV's capability for combined simulation.

A chemical reactor system consists of a compressor, a surge tank, and four reactors (see Figure 4.2).

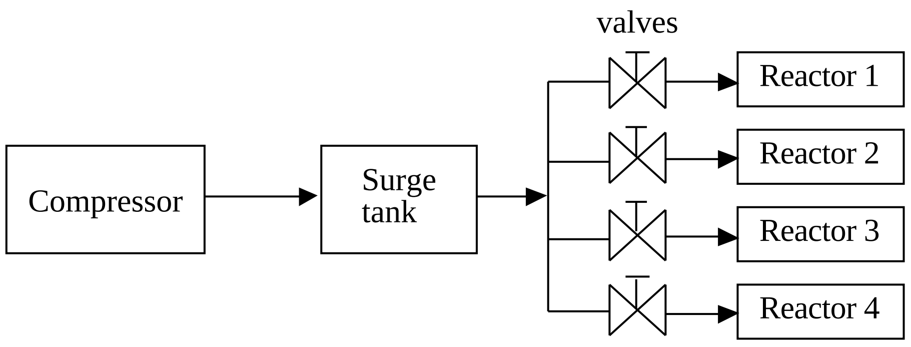


Figure 4.2 – *Schematic diagram of the reactor system*

The reactors are charged with reactants, which are supplied with hydrogen from the compressor through the surge tank. The reactants react under pressure with the hydrogen, which causes the concentration of the reactants to decrease. The effective pressure in each reactor is automatically adjusted to the minimum of the surge tank pressure and the critical pressure (100 psia).

When the concentration of a reactant decreases to 10% of its initial value, the reaction is considered complete and the reactor is turned off and made ready for a fresh batch of reactant.

Initially the surge tank pressure is 500 psia. If the pressure falls below the critical value of 100 psia, the last reactor started will be turned off immediately. The other reactors will continue, but no reactor will be started as long as surge tank pressure is below a nominal pressure of 150 psia.

If two or more reactors can start at the same time, the reactor with the highest value of accumulated batch processing time will be started first.

The reactor system involves both discrete events and continuous state variables. The starting and stopping of reactors are discrete events. Between these events concentration and pressure vary continuously.

The program that follows gives a precise description of the system.

The continuous processes of the system are described in the class `Reactions` (lines 28-44). The class is used to define a single continuous process having a variable number of differential equations. The process computes the active reactors' rate of change in concentration (lines 33-36) and the rate of change in surge tank pressure (lines 40-41).

The discrete processes of the system are described in the classes `Reactor` and `Interrupter`.

Class `Reactor` (lines 44-85) describes the reactors and associated events. Starting a reactor is allowed only if the surge tank pressure is above the nominal pressure. When the reaction process is completed, the reactor is stopped, cleaned (line 77) and recharged (line 78).

Class `Interrupter` (lines 86-106) describes the discrete events occurring whenever surge tank pressure drops to the critical pressure. An `Interrupter`-object sees that the last reactor started is turned off when the pressure falls below the critical pressure. Further, the object ensures that the effective pressure in the reactors is the minimum of surge tank pressure and the critical pressure.

The main process (lines 16-27) specifies the initial conditions, the step-size and accuracy requirements, and the length of the simulation period. Initially the four reactors are scheduled to be turned on at intervals of half an hour (lines 20-23). The simulation period is 150 hours (line 26).

```
import jDisco.*;
import jDisco.Process;

public class ReactorSystem extends Process {
    static final double flowFromCompressor = 4.3523;
    static final double factor = 107.03;
    static final double nominalPressure = 150;
    static final double criticalPressure = 100;
    Reactor[] reactor =
        { new Reactor(0.03466, 10, 0.1),
          new Reactor(0.00866, 15, 0.4),
          new Reactor(0.01155, 20, 0.2),
          new Reactor(0.00770, 25, 0.5) };
    Variable pressure;
    double effectivePressure;
    Head started = new Head();

    public void actions() {
        dtMin = 0.001; dtMax = 1;
        maxRelError = maxAbsError = 0.00001;
        pressure = new Variable(500).start();
        activate(reactor[0], at, 0.0);
        activate(reactor[1], at, 0.5);
        activate(reactor[2], at, 1.0);
        activate(reactor[3], at, 1.5);
        activate(new Interrupter());
        new Reactions().start();
        hold(150);
    }

    class Reactions extends Continuous {
        public void derivatives() {
            double flowToReactors = 0;

            for (Reactor r = (Reactor) started.first();
                 r != null; r = (Reactor) r.suc()) {
                r.concentration.rate =
                    -r.reactionConstant *
                    r.concentration.state *
                    effectivePressure;
                flowToReactors +=
                    -r.concentration.rate * r.volume;
            }
            pressure.rate = factor *
                (flowFromCompressor - flowToReactors);
        }
    }
    class Reactor extends Process {
        Reactor(double rC, double v, double iC) {
            reactionConstant = rC;
            volume = v;
            initialConcentration = iC;
            concentration = new Variable(iC);
        }

        public void actions() {
            Random rand = new Random(7913);
            while (true) {
                double processingTime = 0;
                while (concentration.state >
                       0.10 * initialConcentration) {
                    waitUntil(new Condition() {
                        public boolean test() {
                            return pressure.state >=
                                   nominalPressure;
                        }
                    }, processingTime);
                    concentration.start();
                    into(started);
                    double startTime = time();
                    waitUntil(new Condition() {
                        public boolean test() {
                            return concentration.state <=
                             0.10 * initialConcentration;
                        }
                    });
                    concentration.stop();
                    out();
                    processingTime += (time() - startTime);
                }
                concentration.state = initialConcentration;
                hold(rand.negexp(1));
                hold(Math.min(rand.normal(1, 0.5), 2));
            }
        }

        Variable concentration;
        double reactionConstant,
               volume,
               initialConcentration;
    }
    class Interrupter extends Process {
       public void actions() {
           while (true) {
               effectivePressure = criticalPressure;
               waitUntil(new Condition() {
                   public boolean test() {
                       return pressure.state <=
                              criticalPressure;
                   }
               });
               activate((Process) started.last());
               effectivePressure = pressure.state;
               waitUntil(new Condition() {
                   public boolean test() {
                       return pressure.state >
                              criticalPressure;
                   }
                });
            }
        }
    }

    public static void main(String[] args) {
        activate(new ReactorSystem());
    }
}
```

4.4 Domino game

This example was suggested by F. E. Cellier [1] as a be benchmark problem that can be used to test the capability of a variable-structure simulation, that is, a simulation in which the number of differential equations varies with time.

Fifty-five identical stones of the Domino game are placed vertically upright in a sequence with the same distance between any two stones. If the first stone is pushed, a chain reaction is started and all stones fall flat.

The aim of the simulation is to determine the distance between successive stones that maximizes the velocity of the chain.

A precise description of the model and the experiment is given in the following program. The program illustrates how the specification of the model (`DominoGame`, lines 3-94) can be separated from the experiment (`main`, lines 76-93).

A `Stone`-object (lines 38-70) represents a falling stone and the associated discrete events: the pushing of the next stone (line 54) and the termination of the fall (lines 62-65).

The fall is governed by Newton’s law. For each stone we have the equation

$$\theta \cdot \varphi'' = m \cdot g \cdot r \cdot \sin \varphi$$

where θ is the moment of inertia, φ is the inclination, m the mass, g the acceleration, and r is half the diagonal (see Figure 4.3).

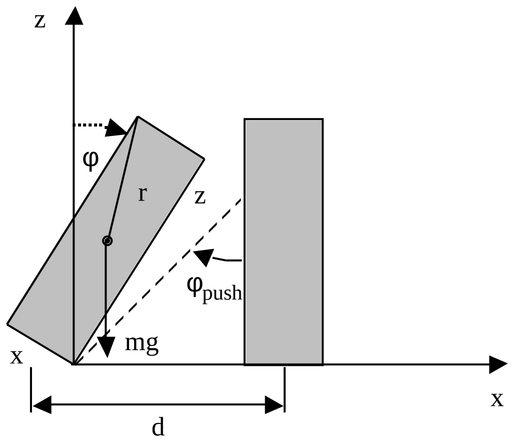


Figure 4.3 – *Graphical description of a falling domino stone*

In class `StoneFall` (lines 27-37) this second-order differential equation is formulated by means of two first-order differential equations.

Each falling stone is associated with one `StoneFall`-object (line 47). A stone that is not moving - either because it has not yet been pushed, or because it has already fallen and lies still - is not associated with such an object. Thus, the number of differential equations will vary with time.

The method `maximum` (lines 97-113) determines a maximum of the function `y = f(x)` on the interval `[a, b]`. The maximum is found within the specified tolerance, `tol`, by using the golden section search method.

For the given parameters the program produces the following output:

```
The maximum chain velocity 0.628m/s
is reached with a distance of 0.0204m between stones
```

```
import jDisco.*;
import jDisco.Process;

public class DominoGame extends Process {
    public DominoGame(int nbrStones,
                      double d, double g, double m,
                      double x, double y, double z) {
        this.nbrStones = nbrStones;
        this.d = d; this.g = g; this.m = m;
        this.x = x; this.y = y; this.z = z;
    }

    public void actions() {
        dtMin = 1.0e-6; dtMax = 0.2;
        maxAbsError = maxRelError = 1.0e-4;
        ko = 1 - (d - x) * (d – x) / (z * z);
        km = m * g * 0.5 * Math.sqrt(x * x + z * z);
        kr = 1 - Math.sqrt(ko);
        phiPush = Math.asin((d - x) / z);
        theta = m * (x * x + z * z) / 3;
        activate(new Stone(z / 2 * 1.0e-5 / theta));
        waitUntil(new Condition() {
            public boolean test() {
                return fallingStones.empty();
            }
        });
        v = d * (stones - 1) / time();
    }

    class StoneFall extends Continuous {
        StoneFall(Stone s) {
            this.s = s;
        }

        public void derivatives() {
            s.omega.rate =
                km * Math.sin(s.phi.state) / theta;
            s.phi.rate = s.omega.state;
        }

        Stone s;
    }
```

```
class Stone extends Process {
    Stone(double initialOmega) {
        this.initialOmega = initialOmega;
    }

    public void actions() {
        into(fallingStones);
        stones++;
        phi = new Variable(0).start();
        omega = new Variable(initialOmega).start();
        fall = new StoneFall(this).start();
        if (stones < nbrStones) {
            waitUntil(new Condition() {
                public boolean test() {
                    return phi.state >= phiPush;
                }
            });
            activate(new Stone(ko * omega.state));
            omega.state *= kr;
        }
        waitUntil(new Condition() {
            public boolean test() {
                return phi.state >= Math.PI/2;
            }
        });
        out();
        fall.stop();
        phi.stop();
        omega.stop();
    }

    double initialOmega;
    Variable phi, omega;
    Continuous fall;
}

int nbrStones;
double d, g, m, x, y, z;

double v, ko, km, kr, phiPush, theta;
int stones;
Head fallingStones = new Head();
```

```
   public static void main(String[] args) {
       Function velocity = new Function() {
           public double f(double x) {
               DominoGame game =
                  new DominoGame(55, x, 9.81, 0.01,
                                 0.008, 0.024, 0.046);
               activate(game);
               return game.v;
           }
       };
       Function.Pair p =
           velocity.maximum(0.008, 0.008 + 0.046, 0.001);
       Format.print(System.out,
          "The maximum chain velocity %5.3f m/s\n", p.x);
       Format.print(System.out,
           "is reached with a distance of " +
           "%6.4f m between stones\n", p.y);
   }
}

public abstract class Function {
    public abstract double f(double x);

    Pair maximum(double a, double b, double tol) {
        double f = (Math.sqrt(5) - 1) / 2;
        double x1 = b - f * (b - a), y1 = f(x1);
        double x2 = a + f * (b - a), y2 = f(x2);

        while (b - x1 > tol) {
            if (y1 >= y2) {
                b = x2; x2 = x1; y2 = y1;
                x1 = b - f * (b - a); y1 = f(x1);
            } else {
                a = x1; x1 = x2; y1 = y2;
                x2 = a + f * (b - a); y2 = f(x2);
            }
        }
        if (y1 >= y2)
            return new Pair(x1, y1);
        return new Pair(x2, y2);
    }

    public class Pair {
        Pair(double x, double y) {
            this.x = x;
            this.y = y;
        }

        public double x, y;
    }
}
```

4.5 Pilot ejection

This example has been taken from [5]. A pilot ejection system, when activated, causes the pilot and his seat to travel along rails at a specified exit velocity (v_E) at an angle (θ_E) backward from vertical. After travelling a vertical distance (Y_1), the seat becomes disengaged from its mounting rail. At this point a second phase begins during which the pilot's trajectory is influenced by the force of gravity and atmospheric drag.

The two phases are continuous. The first phase is described by two differential equations. The second phase is described by four differential equations. A discrete event separates the two phases.

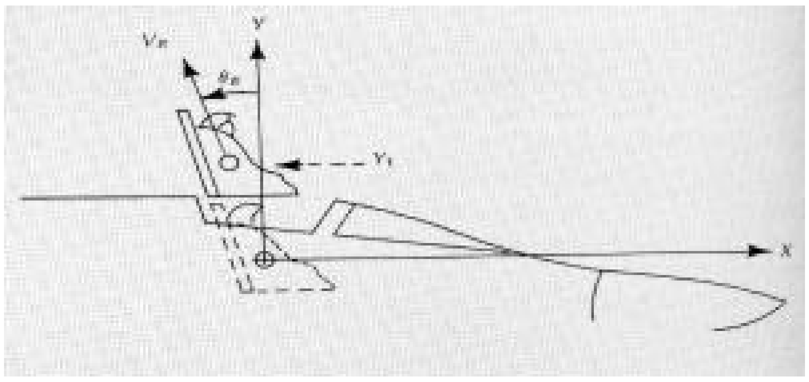


Figure 4.4 - *First phase of ejection*

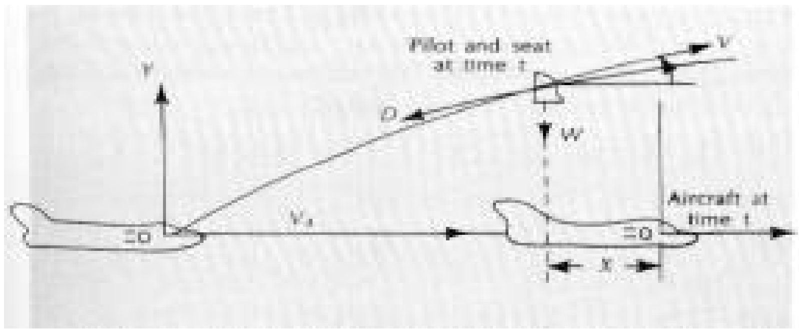


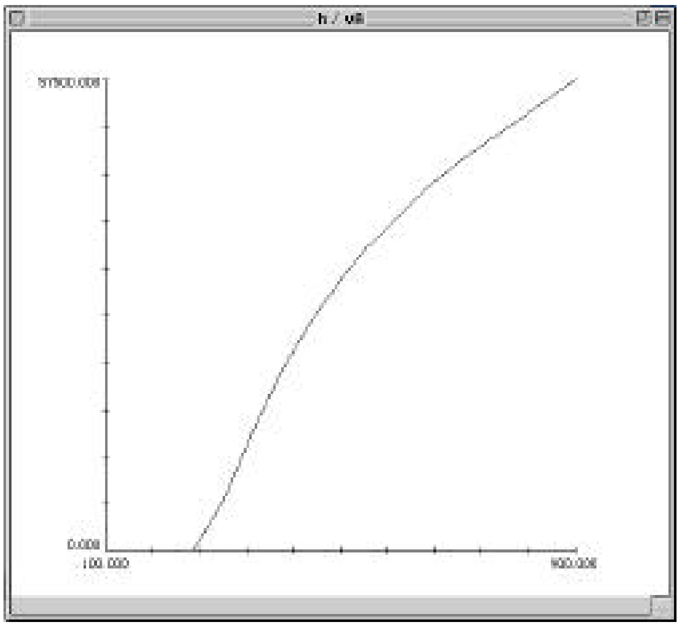
Figure 4.5 - *Second phase of ejection*

The following program determines how large the maximum velocity of the aircraft (v_A) may be, as a function of the height above sea level (h), in order to allow for a safe ejection. An ejection is said to be "safe" if the ejection seat clears the vertical stabilizer of the aircraft, which is 30 feet behind the cockpit, at a distance of at least 20 feet.

```
import jDisco.*;
import jDisco.Process;

public class PilotEjection extends Process {
    public PilotEjection(double vA, double h) {
        this.vA = vA; this.h = h;
    }

    public void actions() {
        density = new Table();
        density.add(    0, 2377e-6);
        density.add( 1000, 2308e-6);
        density.add( 2000, 2241e-6);
        density.add( 4000, 2117e-6);
        density.add( 6000, 1987e-6);
        density.add(10000, 1755e-6);
        density.add(15000, 1497e-6);
        density.add(20000, 1267e-6);
        density.add(30000,  891e-6);
        density.add(40000,  587e-6);
        density.add(50000,  364e-6);
        density.add(60000, 2238e-7);
        rhoP = density.value(h) * cD * s;
        vx = vA - vE * Math.sin(thetaE);
        vy = vE * Math.cos(thetaE);
        x = new Variable(0).start();
        y = new Variable(0).start();
        v = new Variable(Math.sqrt(vx * vx + vy * vy));
        theta = new Variable(Math.atan(vy / vx));
        phase = new Phase1().start();
        waitUntil(new Condition() {
            public boolean test() {
                return y.state >= Y1;
            }
        });
        phase.stop();
        phase = new Phase2().start();
        v.start();
        theta.start();
        waitUntil(new Condition() {
            public boolean test() {
                return x.state <= -30;
           }
        });
        safe = y.state >= 20;
    }
```

```
     class Phase1 extends Continuous {
        public void derivatives() {
            x.rate = v.state * Math.cos(theta.state) - vA;
            y.rate = v.state * Math.sin(theta.state);
        }
    }

    class Phase2 extends Phase1 {
        public void derivatives() {
            super.derivatives();
            v.rate = (-0.5 * rhoP * (v.state * v.state)) / m
                     – g * Math.sin(theta.state);
            theta.rate = -g * Math.cos(theta.state) /
                         v.state;
        }
    }

    double va, h;
    boolean safe;
    static final double m = 7, g = 32.2, cD = 1, s = 10,
                        Y1 = 4, vE = 40, thetaD = 15,
                        thetaE = thetaD / 57.3;
    double vx, vy, rhoP;
    Variable x, y, v, theta;
    Continuous phase;
    Table density;

    public static void main(String[] args) {
        double h = 0;
        Graph graph = new Graph("h / vA");
        for (double vA = 100; vA <= 900; vA += 50) {
            while (true) {
                PilotEjection pe = new PilotEjection(vA, h);
                activate(pe);
                if (pe.safe)
                    break;
                h += 500;
            }
            graph.add(vA, h);
        }
        graph.show();
    }
}
```

The program plots the ejection level against the aircraft velocity using jDisco's class `Graph`.



4.6 Bouncing balls

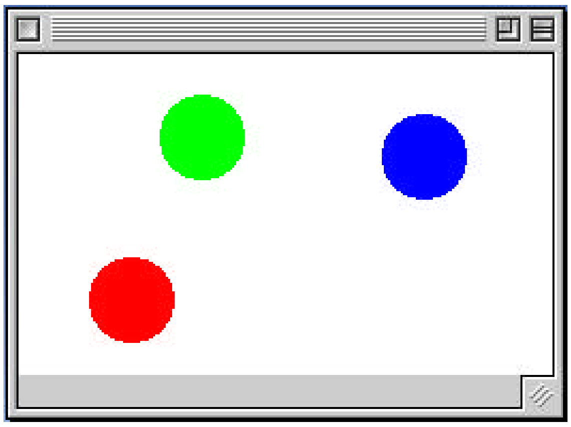
The following program animates a number of balls moving inside a rectangular box. A ball reverses direction if it touches any of the four sides of the box. If two balls collide they bounce off each other.

A `Ball`-object (line 9-44) represents a ball and its associated continuous motion. Each ball has a position (`x, y`), velocities in the x and y directions (`vx, vy`), a radius and a colour. The continuous motion of a ball is specified in its `derivatives` method (lines 22-25).

The classes `BounceHandler` (lines 45-87) and `CollisionHandler` (lines 88-121) handle the bouncing events.

For each ball there is a `BounceHandler`-object. Each time a ball touches any of the sides of the box its direction is reversed (line 69 and line 76).

An object of class `CollisionHandler` handles all collisions between balls. Anytime two balls collide their velocities are exchanged (lines 111-112).



```java
import jDisco.*;
import jDisco.Process;
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class BouncingBalls extends Process {
    static Vector allBalls = new Vector();

    class Ball extends Continuous {
        Variable x, y;
        double radius, vx, vy;
        Color color;

        Ball(double x0, double y0, double radius,
             double vx, double vy, Color color) {
            x = new Variable(x0).start();
            y = new Variable(y0).start();
            this.radius = radius; this.color = color;
            this.vx = vx; this.vy = vy;
            allBalls.addElement(this);
            start();
        }

        public void derivatives() {
            x.rate = vx;
            y.rate = vy;
        }

        public double distance(Ball b) {
            double dx = x.state - b.x.state;
            double dy = y.state - b.y.state;
            return Math.sqrt(dx * dx + dy * dy);
        }

        public boolean collidesWith(Ball b) {
            return b != this &&
                   distance(b) < radius + b.radius;
        }

        public void draw(Graphics g) {
            Color oldColor = g.getColor();
            g.setColor(color);
            g.fillOval((int) (x.state - radius),
                       (int) (y.state - radius),
                       (int) (radius * 2),
                       (int) (radius * 2));
            g.setColor(oldColor);
        }
    }
```

```
   class BounceHandler extends Process {
       BounceHandler(Ball ball, Dimension d) {
           this.ball = ball;
           this.d = d;
       }

       boolean horizontalBounce() {
           return ball.x.state < ball.radius ||
                  ball.x.state > d.width - ball.radius;
       }

       boolean verticalBounce() {
           return ball.y.state < ball.radius ||
                  ball.y.state > d.height - ball.radius;
       }

       boolean bounce() {
           return horizontalBounce() || verticalBounce();
       }

       public void actions() {
           while (true) {
               waitUntil(new Condition() {
                   public boolean test() {
                       return bounce();
                   }
               });
               if (horizontalBounce()) {
                   ball.vx = -ball.vx;
                   waitUntil(new Condition() {
                       public boolean test() {
                           return !horizontalBounce();
                       }
                   });
               } else if (verticalBounce()) {
                   ball.vy = -ball.vy;
                   waitUntil(new Condition() {
                       public boolean test() {
                           return !verticalBounce();
                       }
                   });
               }
           }
     }

     Ball ball;
     Dimension d;
}
```

```
class CollisionHandler extends Process {
    boolean collision() {
        hit = null;
        for (int i = 0; i < allBalls.size(); i++) {
            ball = (Ball) allBalls.elementAt(i);
            for (int j = i + 1; j < allBalls.size();
                 j++) {
              hit = (Ball) allBalls.elementAt(j);
              if (ball.collidesWith(hit))
                  return true;
            }
        }
        ball = hit = null;
        return false;
    }

    public void actions() {
        while (true) {
            waitUntil(new Condition() {
                public boolean test() {
                    return collision();
                }
            });
            double t;
            t = ball.vx; ball.vx = hit.vx; hit.vx = t;
            t = ball.vy; ball.vy = hit.vy; hit.vy = t;
            waitUntil(new Condition() {
                public boolean test() {
                    return !collision();
                }
            });
        }
    }

    Ball ball, hit;
}

class Canvas extends JComponent {
    public void paintComponent(Graphics g) {
        g.setColor(Color.white);
        Dimension d = getSize();
        g.fillRect(0, 0, d.width, d.height);
        for (int i = 0; i < allBalls.size(); i++) {
            Ball b = (Ball) allBalls.elementAt(i);
            b.draw(g);
        }
    }
}
```

```
    public void actions() {
        dtMin = 1.0e-5; dtMax = 1; maxRelError = 1.0e-5;
        final JFrame frame = new JFrame();
        frame.getContentPane().add(new Canvas());
        frame.setSize(250, 150);
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        Dimension d = frame.getSize();
        new Ball(d.width * 2 / 3, d.height - 20, 20, -2, -4,
                 Color.green);
        new Ball(    d.width / 4, d.height - 20, 20,  2, 10,
                 Color.red);
        new Ball(   d.width / 10, d.height - 40, 20, -5,  3,
                 Color.blue);
        for (int i = 0; i < allBalls.size(); i++)
            activate(new BounceHandler(
                         (Ball) allBalls.elementAt(i), d));
        activate(new CollisionHandler());
        frame.setVisible(true);
        activate(new Process() {
            public void actions() {
                while (true) {
                    frame.repaint();
                    hold(0.1);
                }
            }
        });
        hold(10000);
    }

    public static void main(String[] args) {
        activate(new BouncingBalls());
    }
}
```

5. Additional facilities

The examples in the preceding section illustrate the application of the essential facilities of jDisco. In this section some additional facilities are briefly mentioned.

5.1 Reporting

A class called `Reporter` is provided for gathering information about model behavior. Each `Reporter`-object may have its user-defined actions executed with a specified frequency, namely, at uniformly spaced intervals, at the end of each time step, or only at event times.

An outline of class Reporter is shown below.

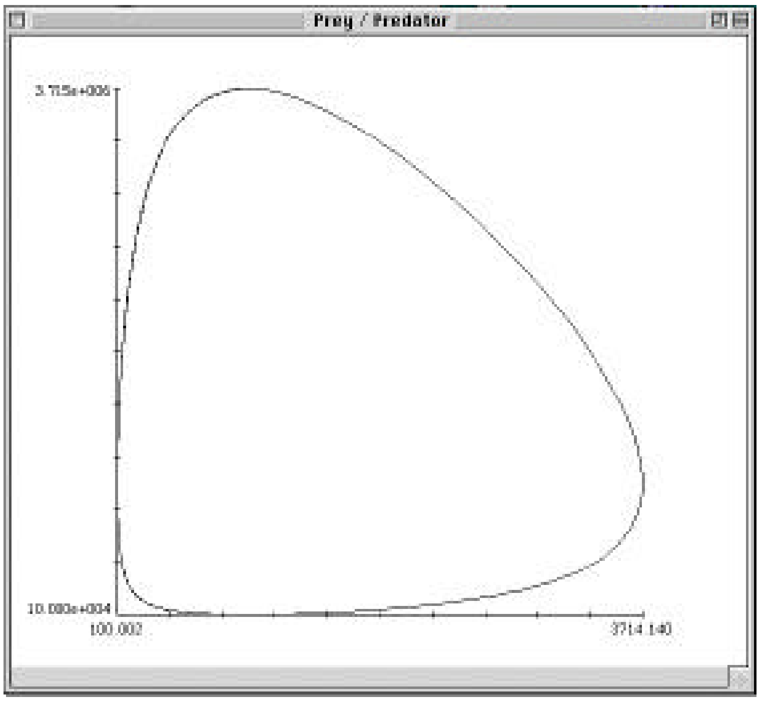
```
public abstract class Reporter extends Link {
    protected abstract void actions();

    public Reporter setFrequency(double f);
    public Reporter start();
    public void stop();
}
```

After its `start` method is called, a `Reporter`-object becomes *active*, that is to say, its `actions` are executed with the frequency set by `setFrequency`.

Facilities for producing curve plots (class `Graph`) and histograms (class `Histogram`) are available for displaying information gathered during a simulation. Below is shown how the predator-prey simulation program may be extended to plot the number of predators versus the number of preys.

```
public class PredatorPreySystem extends Process {
    Variable predator, prey;
    Graph graph = new Graph(" Prey / Predator ");

    public void actions() {
        ...
        class StateReporter extends Reporter {
            public void actions() {
                graph.add(predator.state, prey.state);
            }
        }
        new StateReporter().setFrequency(0.1).start();
        hold(100);
        graph.show();
    }
    ...
}
```

A run of the program produces the following output:



5.2 Numerical integration

The user is offered the choice of a number of numerical integration methods. Currently the following 11 methods are available: Runge-Kutta-England (RKE), Runge-Kutta-Fehlberg (RKF45), Runge-Kutta-Nørsett (RKN34), Runge-Kutta-Dormand-Price (RKDP45), Runge-Kutta-Verner (RKV56), Adams-Bashforth variable order predictor-corrector, Fowler-Warten, Trapez, Simpson and Euler. If the user does not specify an integration method, the program uses the Runge-Kutta-Fehlberg method (RKF45). The integration step size is variable and is automatically adjusted to meet the specified accuracy requirements.

In a continuous process the order in which the equations are written is left to the user. Because jDisco does not change the execution sequence of the equations, a correct sequencing is the responsibility of the user. To prevent unintentional delays from being introduced into the model dynamics, the user must make sure that the variables occurring on the right-hand side of an equation have values that reflect the current state of the system. The user can determine the order of evaluation within each continuous process, and the continuous processes themselves may be ranked by giving each a priority. Usually a correct evaluation order can be achieved by these means. An implicit function facility can be used to circumvent algebraic loops in the system of equations.

Additionally, it is possible to describe systems using *difference* equations. This capability may, for example, be used to specify models of the systems dynamics type.

5.3 Event sequencing

Discrete processes operate in *quasi-parallel*, which means that concurrent events are executed in a certain order. Since the ordering may be important the user must be able to determine their sequence. Time-events are ordered by using the keywords `before`, `after` and `prior`. State-events, that is events projected by the `waitUntil`-method, can be ordered by giving each a priority. An example of the use of this feature is illustrated in class `Reactor` (page 21, lines 58-63).

5.4 Utility software

The utility software includes among other things facilities for handling higher-order differential equations, ideal and exponential delays, tabulated functions, collection of statistics, and partial differential equations.

6. Implementation

A simulation is controlled behind the scenes, so to speak, by an object called the *monitor*.

It is the monitor's responsibility to see that

(1) The model state varies "continuously" between events

(2) Discrete events take place at the right time

(3) Information about the model's behaviour is gathered.

The monitor ensures that all continuous parts of the model operate in full parallel and fully synchronized with the quasi-parallel discrete processes.

Time in the model is advanced by steps of varying size. The monitor adjusts step-size so that no event occurs within a step and so that desired accuracy in updating state variables is maintained.

The monitor causes the events to take place at the right time. The event times of state-events are determined with an accuracy of `dtMin` using bisection and interpolation.

The monitor controls object of class `Reporter`. Each `Reporter`-object has its user-defined actions executed with a specified frequency. Interpolation is also used here to provide an efficient and accurate determination of the model's state at the reporting times.

The working cycle of the monitor is outlined below.

```
while (more projected events) {
        Execute all active Continuous-objects;
        Execute all active Reporter-objects;

    while (no event now) {
            Take an integration step fulfilling accuracy requirements;
            if (a state-event was passed)
                    Determine the event time and reduce step accordingly;
            Execute active Reporter-objects when requested;
    }

    Let an event take place now;
}
```

7. Conclusion

The jDisco package is a convenient tool for simulation. Even though the package is relatively small, it contains useful utilities for easily construction of complex simulation models.

Few concepts together with a convenient notation make jDisco easy to learn and use. The distinction between continuous and discrete processes aids in the conceptualisation of combined systems.

The package permits general interaction between processes and ensures their synchronous operation. Model structures can be changed by the addition, substitution, or deletion of any type of process, thus allowing simulation of systems with a variable structure.

References

[1] Cellier, C. F.
Combined Continuous Discrete Simulation by Use of Digital Computers: Techniques and Tools
PhD thesis, Swiss Federal Institute of Technology, Zürich 1979.

[2] Helsgaun, K.
DISCO – a SIMULA-based language for combined continuous and discrete simulation
SIMULATION, July 1980 (pp. 1-12).

[3] Helsgaun, K.
Discrete Event Simulation in Java
Datalogiske skrifter, No. 89, Roskilde University, 2000.

[4] Pritkser, A. A. B, Hurst, N. R.
GAPP IV: a Combined Continuous-Discrete FORTRAN-Based Simulation Language
Wiley, New York, 1975.

[5] SCi Simulation Software Committee
The SCi Continuous System Simulation Language (CSSL)
SIMULATION, Vol. 9, December 1967, pp. 281-303.

[6] Sierenberg, R. W.
Combined Discrete and Continuous Simulation with PROSIM
Proceedings Simulation ’77 (M. H. Hamza, editor).

[7] Spechart, F. H., Green, W. L.
A Guide to Using CSMP – the Continuous System Modelling Program
Prentice Hall, Englewood Cliffs, New Jersey 1976 (pp. 35-39).