The SugarCubes Tool Box
Nets of Reactive Processes Implementation

FRÉDÉRIC BOUSSINOT, JEAN-FERDY SUSINI
INRIA EMP-CMA/Meije
2004 route des lucioles
F-06902 Sophia-Antipolis
`fb@sophia.inria.fr, jfsusini@sophia.inria.fr`

April 18, 1997

Contents

1 Introduction

Nets of Reactive Processes (NRP) is a computing model defined in the following way:

- A net is made of processes which execute in parallel and communicate through channels.
- Channels are unbounded first-in/first-out files.
- To communicate, processes can take informations from their input channels or can put informations into their output channels.
- Processes are reactive programs which share the same notion of an instant: a run of a net means a run of each of the processes belonging to it.
- There is no possibility to know that a channel is empty at a given instant, before the end of this instant; thus *reaction to emptyness is always postponed to the next instant.* This is the exact counterpart for channels of the basic principle of SugarCubes for events.

In this paper we define channels, the notion of a reactive process, and the notion of a reactive machine for reactive nets. We give the definition of some basic processes and use them to code two small examples.

All these definitions are in SugarCubes presented in the paper:

- The SugarCubes Tool Box - Definition

.

2 Channels

The `Channel` class defines unbounded fifo files.

```
class Channel
{
  protected Vector fifo = new Vector();

  public void put(Object obj){ fifo.addElement(obj); }

  public Object get(){
    Object res = fifo.firstElement();
    fifo.removeElementAt(0);
    return res;
  }

  public boolean isEmpty() { return fifo.isEmpty(); }
}
```

ChannelEnv associates channels to names. The newChannel method always return a new channel with a unique name.

```
public class ChannelEnv
{
  private Hashtable channelEnv  = new Hashtable();
  private int localNum = 0;

  public Channel getChannel(String name)
  {
     Channel c = (Channel)channelEnv.get(name);
     if (c==null){
       c = new Channel();
       channelEnv.put(name,c);
     }
     return c;
  }

  /* Strange name, is'nt it ? */
  public Channel newChannel(){
    return getChannel("@@@local@@@" + (localNum++));
  }
}
```

3 NrpMachine

A NrpMachine is a machine that executes a net of reactive processes. Class NrpMachine extends Machine by adding a channel environment.

```
public class NrpMachine extends Machine
{
  public ChannelEnv channelEnv = new ChannelEnv();

  final public Channel getChannel(String name){
    return channelEnv.getChannel(name);
  }

  final public Channel newChannel(){
    return channelEnv.newChannel();
  }
}
```

4 Reactive Processes

We first define the class of reactive processes then give some basic processes.

4.1 ReactiveProcess Class

The abstract class `ReactiveProcess` of reactive processes extends `Instruction`. It defines a `fix` method which returns `TERM` is its channel parameter is not empty, `STOP` it it is empty (which is known only at the end of the current instant), and `SUSP` otherwise.

The `newMove` method of `Machine` is called when an object is put into a channel.

```
abstract public class ReactiveProcess extends Instruction
{
  protected byte fix(Channel chan,Machine machine)
  {
    if (!chan.isEmpty()) return TERM;
    if (machine.isEndOfInstant()) return STOP;
    return SUSP;
  }

  protected void put(Channel chan,Object obj,Machine machine)
  {
    chan.put(obj);
    machine.newMove();
  }
}
```

4.2 Constant Process

`Const` puts an integer constant on its output channel, at each instant.

```
public class Const extends ReactiveProcess
{
  private Channel out;
  private int val;

  public Const(Channel out,int val){
    this.out = out; this.val = val;
  }

  public Const(NrpMachine machine,String out,int val) {
    this.out = machine.getChannel(out);
    this.val = val;
    machine.add(this);
  }

  final public String toString(){
    return "const(" + val + "," + out + ")";
  }

  protected byte activation(Machine machine)
  {
    put(out,new Integer(val),machine);
    return STOP;
  }
}
```

4.3 Integer List

IntValues produces the list of integers (starting from an initial value) on its output.

```
public class IntValues extends ReactiveProcess
{
  private Channel out;
  private int from = 0;

  public IntValues(Channel out,int from)
  {
    this.out  = out;
    this.from = from;
  }

  public IntValues(NrpMachine machine,String out,int from)
  {
    this.out  = machine.getChannel(out);
    this.from = from;
    machine.add(this);
  }

  final public String toString(){
    return "integers(" + out + "," + from + ")";
  }

  protected byte activation(Machine machine)
  {
    put(out,new Integer(from++),machine);
    return STOP;
  }
}
```

4.4 Out

Out prints its input (considered as a channel of long values).

```
public class Out extends ReactiveProcess
{
  private Channel in;

  public Out(Channel in){ this.in = in; }

  public Out(NrpMachine machine,String in){
    this.in = machine.getChannel(in);
    machine.add(this);
  }

  final public String toString(){ return "out(" + in + ")"; }

  protected byte activation(Machine machine)
  {
    byte b = fix(in,machine);
    if (b == TERM){
      long i = ((Number)in.get()).longValue();
      System.out.println(i);
      return STOP;
    }
    return b;
  }
```

4.5 Follow Process

Follow outputs the first item of its left input channel, then duplicates its right input on the output.

```
public class Follow extends ReactiveProcess
{
  private Channel in, left, right, out;
  private boolean first = true;

  public Follow(Channel left,Channel right,Channel out)
  {
    this.left = left;
    this.right = right;
    this.out = out;
  }

  public Follow(NrpMachine machine,
                    String left,String right,String out)
  {
    this.left = machine.getChannel(left);
    this.right = machine.getChannel(right);
    this.out = machine.getChannel(out);
    machine.add(this);
  }

 final public String toString(){
    return "follow(" + out + "," + left + "," + right + ")";
  }

  protected byte activation(Machine machine)
  {
    Channel in = first ? left : right;
    byte b = fix(in,machine);
    if (b == TERM){
      put(out,in.get(),machine);
      first = false;
      return STOP;
    }
    return b;
  }
}
```

4.6 The Funnel Process

`Funnel` merges its left and right inputs on its output.

```
public class Funnel extends ReactiveProcess
{
  private Channel in, left, right, out;
  private boolean fromLeft = true;

  public Funnel(Channel left,Channel right,Channel out)
  {
    this.left = left;
    this.right = right;
    this.out = out;
  }

  public Funnel(NrpMachine machine,String left,
                              String right,String out)
  {
    this.left = machine.getChannel(left);
    this.right = machine.getChannel(right);
    this.out = machine.getChannel(out);
    machine.add(this);
  }

 final public String toString(){
    return "funnel(" + out + "," + left + "," + right + ")";
  }

  protected byte activation(Machine machine)
  {
    Channel in = fromLeft ? left : right;
    byte b = fix(in,machine);
    if (b == TERM){
      put(out,in.get(),machine);
      b = STOP;
    }
    if (b != SUSP) fromLeft = !fromLeft;
    return b;
  }
}
```

Two points are important to note:

- `Funnel` does not remain for ever stuck on an empty input channel;
- emptyness of an input channel (when `fix` returns `STOP`) cannot be decided before the end of the current instant, so switching to the opposite input channel can only take place at next instant.

4.7 Binary Functions

The abstract class `BinaryFunction` is the class of processes that take an item on each of their two input channel, then apply a function `fun` to these idems and put the result on their output.

```
public abstract class BinaryFunction extends ReactiveProcess
{
  protected Channel left, right, out;
  protected boolean getLeft,getRight;

  public BinaryFunction(Channel left,Channel right,Channel out)
  {
    this.left  = left;
    this.right = right;
    this.out   = out;
    getLeft = getRight = false;
  }

  public BinaryFunction(NrpMachine machine,String left,
                                      String right,String out)
  {
    this.left  = machine.getChannel(left);
    this.right = machine.getChannel(right);
    this.out   = machine.getChannel(out);
    getLeft = getRight = false;
    machine.add(this);
  }

  protected abstract Object fun(Object arg1,Object arg2);

  protected byte activation(Machine machine)
  {
    if (fix(left,machine) == TERM ){ getLeft = true; }
    if (fix(right,machine) == TERM ){ getRight = true; }
    if (getLeft && getRight){
      getLeft = getRight = false;
      put(out,fun(left.get(),right.get()),machine);
      return STOP;
    }
    return SUSP;
  }
}
```

4.8 Sum Process

The `Sum` extends `BinaryFunction` with the addition function defined for `long` parameters.

```
public class Sum extends BinaryFunction
{
  public Sum(Channel left,Channel right,Channel out){
     super(left,right,out);
  }

  public Sum(NrpMachine machine,
                String left,String right,String out){
     super(machine,left,right,out);
  }

  final public String toString(){
    return "sum(" + out + "," + left + "," + right + ")";
  }

  final protected Object fun(Object arg1,Object arg2){
     return new Long(
         ((Number)arg1).longValue()+((Number)arg2).longValue());
  }
}
```

5 Examples

We give two examples of nets of reactive processes: a net which computes the Fibonnacci list and a net which computes prime numbers.

5.1 Fibonnacci

The `Fibonnacci` machine prints the 50 first values of the Fibonnacci list.

```
public class Fibonnacci extends NrpMachine
{
  public Fibonnacci()
  {
    new Sum(this,"c0","c1","c2");
    new Const(this,"c3",1);
    new Const(this,"c4",0);
    new Follow(this,"c3","c2","c5");
    new Dup(this,"c5","c8","c6");
    new Dup(this,"c6","c7","c1");
    new Follow(this,"c4","c7","c0");
    new Out(this,"c8");
  }

  public static void main (String argv[])
  {
    Fibonnacci mach = new Fibonnacci();
    for (int i=0;i<50;i++){ mach.activation(mach); }
  }

}
```

5.2 Eratosthenes Sieve

The sieve of Eratosthenes is made of two reactive processes: `Filter` and `Sift`.

The Filter Process

`Filter` filters all multiples of the first value it receives.

```
public class Filter extends ReactiveProcess
{
  private Channel in, out;
  private int prime;


  public Filter(Channel in, Channel out,int prime)
  {
    this.in    = in;
    this.out   = out;
    this.prime = prime;
  }

  public Filter(NrpMachine machine,String in,
                             String out,int prime)
  {
    this.in    = machine.getChannel(in);
    this.out   = machine.getChannel(out);
    this.prime = prime;
    machine.add(this);
  }

  final public String toString(){
    return "filter(" + prime + "," + out + "," + in + ")";
  }

  protected byte activation(Machine machine)
  {
    byte b = fix(in,machine);
    if (b == TERM){
      int v = ((Integer)in.get()).intValue();
      if (0 != v % prime){ put(out,new Integer(v),machine); }
      return STOP;
    }
    return b;
  }
}
```

The Sift Process

The `Sift` reactive process is recursively defined. First, it outputs the first value `v` it receives, then it adds in front of itself a process which filters all multiple of `v`.

```
public class Sift extends ReactiveProcess
{
  private Channel in, out, intern;
  private int prime;
  private boolean first = true;

  public Sift(Channel in, Channel out)
  {
    this.in    = in;
    this.out   = out;
  }

  public Sift(NrpMachine machine,String in, String out)
  {
    this.in = machine.getChannel(in);
    this.out= machine.getChannel(out);
    machine.add(this);
  }

  final public String toString(){
    return "sift(" + prime + "," + out + "," + in + ")";
  }

  protected byte activation(Machine machine)
  {
    if (first){
      byte b = fix(in,machine);
      if (b == TERM){
        prime = ((Integer)in.get()).intValue();
        put(out,new Integer(prime),machine);
        first = false;
        return STOP;
      }
      return b;
    }else{
      intern = ((NrpMachine)machine).newChannel();
      machine.add(new Filter(in,intern,prime));
      machine.add(new Sift(intern,out));
      return TERM;
    }
  }
}
```

The Sieve Process

`Sieve` is the machine that computes prime numbers using the Eratosthenes sieve.

```
public class Sieve extends NrpMachine
{
  public Sieve()
  {
    new IntValues(this,"c0",2);
    new Sift(this,"c0","c1");
    new Out(this,"c1");
  }

  public static void main (String argv[])
  {
    Sieve mach = new Sieve();
    for (;;){ mach.activation(mach); }
  }
}
```

6 Conclusion

We have used SugarCubes to implements the model of nets of reactive processes. This shows that SugarCubes are useful when applying the reactive approach to the dataflow model.

Index