Sémantique de JUNIOR

exec(t,E) = (t',E',b)

t : instruction de départ
E : environnement de départ
t' : instruction d'arrivée
E' : environnement d'arrivée
b : status de terminaison

Environnement :

ensemble d'événements présents + 2 booléens : eoi et move

Status de terminaison :

TERM, STOP ou SUSP

Séquence :

exec(`Seq(t,u)`,E) =
soit exec(`t`,E) = (`t'`,E',b) ;
si b ≠ TERM alors (`Seq(t',u)`,E',b) sinon exec(`u`,E')

Sémantique - 2

Stop et nothing :

exec(`Nothing`,E) = (`Nothing`,E,TERM)
exec(`Stop`,E) = (`Nothing`,E,STOP)

Loop :

exec(`Loop(t)`,E) = exec(`Seq(t,Loop(t))`,E)

Generate :

exec(`Generate(S)`,E) = (`Nothing`,E+{S}+[move = vrai] ,TERM)

Présence :

exec(`When(S,t,u)`,E) =
si S dans E alors exec(`t`,E),
sinon si eoi alors (`u`,E,STOP),
sinon (`When(S,t,u)` ,E,SUSP)

Sémantique - 3

Await :

exec(`Await(S)`,E) =
si S dans E alors (`Nothing`,E,TERM),
sinon si eoi alors (`Await(S)`,E,STOP),
sinon (`Await(S)`,E,SUSP)

Préemption :

exec(`Until(S,t)`,E) =
si exec(`t`,E) = (`t'`,E',b) et b = STOP alors exec(`Waiting (S,t')`,E')
sinon (`Until(S,t')`,E',b)

exec(`Waiting(S,t)`,E) =
si S dans E alors (`Nothing`,E,TERM),
sinon si eoi alors (`Until(S,t)`,E,STOP),
sinon (`Waiting(S,t)`,E,SUSP)

Sémantique - 4

Parallélisme :

exec(Par(t,u),E) = exec(Par(t,u,SUSP,SUSP),E)

exec(Par(t,u,b,SUSP),E) =
(aux(t,u',b,c),E', stat(b,c))
où exec(u,E) = (u',E',c)

stat	TERM	STOP	SUSP
TERM	TERM	STOP	SUSP
STOP	STOP	STOP	SUSP
SUSP	SUSP	SUSP	SUSP

exec(Par(t,u,SUSP,c),E) =
(aux(t',u,b,c),E',stat(b,c))
où exec(t,E) = (t',E',b)

exec(Par(t,u,SUSP,SUSP),E) = (aux(t',u',b,c),E'',stat(b,c))
où exec(t,E) = (t',E',b) et exec(u,E') = (u',E'',c)

aux(t,u,b,c) = Par(t,u) si b et c ≠ SUSP,
Par(t,u,b,c) sinon

Sémantique - 5

Evénements :

exec(`Event(S,t)`,E) = exec(`AbsentEvent(S,t)`,E)

E'' est égal à E', mais E''(S) = E(S)

exec(`AbsentEvent(S,t)`,E) =
soit exec(`t`,E-{S}) = (`t'`,E',b) ;
si b ≠ SUSP alors (`Event(S,t')`,E'',b)
sinon si S dans E' alors (`PresentEvent (S,t')`,E'',SUSP)
sinon (`AbsentEvent(S,t')` ,E'',SUSP)

exec(`PresentEvent(S,t)`,E) =
soit exec(`t`,E+{S}) = (`t'` ,E',b) ;
si b ≠ SUSP alors (`Event(S,t')`,E'',b)
sinon (`PresentEvent(S,t')` ,E'',SUSP)

Sémantique - 6

Instants :

exec(`instant(t)`,E) =
soit exec(`t`,E[move = faux]) = (`instant(t')`,E',b) ;
si b ≠ SUSP alors (`instant(t')`,E',b)
sinon si move = vrai alors exec(`instant(t')`,E')
sinon exec(`instant(t')`,E'[eoi = vrai])

Implémentations

- **REWRITE = exec**
- **REPLACE = REWRITE + optimisation des créations de structures intermédiaires**
- **SIMPLE = implémentation adaptée à un grand nombre d'événements**
- **…**

REWRITE

exec(t,E) = (t',E',b)

Instruction

```
MicroState s = t.rewrite(E);
b = s.flag
t' == s.term
E' est la nouvelle valeur de E
```

TERM, STOP, SUSP

```
abstract public class Instruction implements Flags, Program
{
   abstract public MicroState rewrite(EnvironmentImpl env);

   public MicroState unknown(){ return new MicroState(SUSP,this); }
}
```

```
public class Stop extends Instruction
{
   public MicroState rewrite(EnvironmentImpl env){
      return new MicroState(STOP,new Nothing());
   }
}
```

exec(Stop,E) =
(Nothing ,E,STOP)

Séquence

```
public abstract class BinaryInstruction extends Instruction
{
  final public Instruction left,right;

  public BinaryInstruction(Program left,Program right){
     this.left = (Instruction)left; this.right = (Instruction)right;
  }
}
```

```
public class Seq extends BinaryInstruction
{
  public Seq(Program left,Program right){ super(left, right); }

  public MicroState rewrite(EnvironmentImpl env){
     MicroState s = left.rewrite(env);
     if (TERM == s.flag) return right.rewrite(env);
     return new MicroState(s.flag,new Seq(s.term,right));
  }
}
```

création d'une nouvelle instruction

exec(`Seq(t,u)` ,E) =
soit exec(`t`,E) = (`t'` ,E',b) ;
si b ≠ TERM alors (`Seq(t',u)` ,E',b)
sinon exec(`u`,E')

Evénements

```
abstract public class Config implements Configuration
{
  abstract public boolean fixed(EnvironmentImpl env);
  abstract public boolean eval(EnvironmentImpl env);
  abstract public void reset();
  public boolean sat(EnvironmentImpl env){ return fixed(env) && eval(env); }
  public boolean unsat(EnvironmentImpl env){ return fixed(env) && !eval(env); }
}


public class Presence extends Config
{
  final public IdentifierWrapper wrapper;
  public boolean evaluated = false;
  public Identifier event;

  public Presence(IdentifierWrapper wrapper){ this.wrapper = wrapper; }
  public void reset(){ evaluated = false; }

  public boolean fixed(EnvironmentImpl env){
     if(evaluated == false){ event = wrapper.evaluate(env); evaluated = true; }
     return env.isGenerated(event) || env.eoi;
  }

  public boolean eval(EnvironmentImpl env){ return env.isGenerated(event); }
}



public class Await extends Instruction
{
  final public Config config;

  public Await(Configuration config){ this.config = (Config)config; }

  public MicroState rewrite(EnvironmentImpl env){
      if (config.sat(env)) return new MicroState((env.eoi ? STOP : TERM),new Nothing());
      if (config.unsat(env)) return new MicroState(STOP,this);
      return unknown();
  }
}
```

Parallélisme

Par implémenté par la classe Merge

```
public class Merge extends BinaryInstruction
{
  final public byte leftFlag, rightFlag;

  public Merge(Program left, Program right, byte leftFlag, byte rightFlag){
     super(left,right); this.leftFlag = leftFlag; this.rightFlag = rightFlag;
  }
  public Merge(Program left,Program right){ this(left,right,SUSP,SUSP); }

  public Instruction newTerm(Instruction l, Instruction r, byte lf, byte rf){
     return new Merge(l,r,lf,rf);
  }

  public MicroState result(Instruction l, Instruction r, byte lf, byte rf){
     byte b = SUSP, nlf = lf, nrf = rf;
     if(lf != SUSP && rf != SUSP){
        b = (lf==TERM && rf==TERM) ? TERM : STOP;
        if (lf==STOP) nlf = SUSP;
        if (rf==STOP) nrf = SUSP;
     }
     return new MicroState(b,newTerm(l,r,nlf,nrf));
  }
  public MicroState rewrite(EnvironmentImpl env){
     if (leftFlag == SUSP && rightFlag != SUSP){
        MicroState s = left.rewrite(env);
        return result(s.term,right,s.flag,rightFlag);
     }
     if (leftFlag != SUSP && rightFlag == SUSP){
        MicroState s = right.rewrite(env);
        return result(left,s.term,leftFlag,s.flag);
     }
     MicroState ls = left.rewrite(env), rs = right.rewrite(env);
     return result(ls.term,rs.term,ls.flag,rs.flag);
  }
}
```

stat	TERM	STOP	SUSP
TERM	TERM	STOP	SUSP
STOP	STOP	STOP	SUSP
SUSP	SUSP	SUSP	SUSP

Instants

```
public class Instant extends UnaryInstruction
{
   public Instant(Program body){ super(body); }

   public MicroState rewrite(EnvironmentImpl env){
      MicroState s = body.rewrite(env);
      if (s.flag != SUSP) return new MicroState(s.flag,new Instant(s.term));
      if (env.move) env.move = false; else env.eoi = true;
      return new Instant(s.term).rewrite(env);
   }
}
```

exec(instant(t),E) =
  soit exec(t,E[move = faux]) = (instant(t'),E',b) ;
  si b ≠ SUSP alors (instant(t') ,E',b)
  sinon si move = vrai alors exec(instant(t'),E')
  sinon exec(instant(t') ,E'[eoi = vrai])

Machines non synchronisées

```
public class BasicContext implements Flags, UnSyncMachine
{
   public Instant instant;
   public EnvironmentImpl env;
   public Program toAdd = Jr.Nothing();
   public boolean somethingToAdd = false;

   public BasicContext(Program program){
      this.instant = new Instant(program);
      buildEnvironment();
   }

   public void buildEnvironment(){ env = new EnvironmentImpl(); }

   public void add(Program inst){
      toAdd = Jr.Par(toAdd,inst);
      somethingToAdd = true;
   }
   protected void performAddings(){
      if (somethingToAdd == false) return;
      instant = new Instant(Jr.Par(toAdd,instant.body));
      toAdd = Jr.Nothing();
      somethingToAdd = false;
   }

   public void generate(Identifier event, Object obj){ env.generate(event,obj); }

   public Program getFrozen (Identifier event){ return env.getFrozen(event); }

   public boolean react(){
      performAddings();
      MicroState s = instant.rewrite(env);
      instant = (Instant)s.term;
      env.newInstant();
      return (TERM == s.flag);
   }
}
```

Machines synchronisées

```
public class ExecContext extends BasicContext implements Machine, SyncMachine
{
   public Vector toGenerate = new Vector();

   public ExecContext(Program program){ super(program); }

   public void add(Program inst){
      synchronized(toAdd){ super.add(inst); }
   }
   protected void performAddings(){
      synchronized(toAdd){ super.performAddings(); }
   }

   public void generate(Identifier event, Object obj){
      synchronized(toGenerate){
         toGenerate.addElement(new GenerateOrder(event,obj));
      }
   }

   protected void performGenerations(){
      synchronized(toGenerate){
         if (toGenerate.size() > 0){
            Enumeration list = toGenerate.elements();
            while (list.hasMoreElements()){
                GenerateOrder order = (GenerateOrder)list.nextElement();
                super.generate(order.identifier,order.value);
            }
            toGenerate.removeAllElements();
         }
      }
   }

   public synchronized boolean react(){
      performGenerations();
      return super.react();
   }
}
```