Introduction to Communicating Sequential Process (CSP) (Lecture 13)

Mannheim, September 2007

Contents

- Introduction to JCSP (CSP for Java)

Introduction

- The implementation of a CSP specification can be done using other environments
  - UML-RT
  - CTJ (libray similar to JCSP)
  - occam (language that implements CSP)
  - ...
- We will see a brief overview of JCSP

JCSP: General Issues

- Java library that implements the communication and concurrency model of CSP/occam (with restrictions)
- Gives support to the development of process-oriented projects.
- Implementation is based on the threads/monitor Java mechanism.
- Versions
  - *Base Edition*
  - *Network Edition* (provides support to distribution)

JCSP: General Issues

- CSP features available
  - Prefix
  - Channels
  - Comunication (including buffers)
  - Sequential Composition
  - Parallelism (but not the alphabetized)
  - External Choice (with restrictions and extensions)
- CSP features not available
  - Hiding, relabelling, alphabetized parallel composition, internal choice, index, ...

JCSP Process

- A process is an autonomous entity (the execution flow is independent)
- Encapsulates states(attributes) and methods
  - Constructors are public
- Communicates with the environment through channels (as in CSP)
- Active behaviour (flow) implemented by method `run()` (public)

JCSP Process

- A ***process*** is an object of a class that implements a interface **`CSProcess`**

```
interface CSProcess {
  public void run();
}
```

- Every class that implements **`CSProcess`** must provide an implementation of **`run()`**

Process Structure

```
class Example implements CSProcess {

  ...  private shared synchronisation objects
       (channels etc.)
  ...  private state information

  ...  public constructors
  ...  public accessors(gets)/mutators(sets)
       (only to be used when not running)

  ...  private support methods (part of a run)
  ...  public void run() (process starts here)

}
```

JCSP Channels

- A ***channel*** is an object of a class that implements:

```
interface Channel {
}
```

JCSP Channels

- There are four types of interfaces:
  - **`ChannelInput`**
  - **`ChannelOutput`**
  - **`ChannelInputInt`**
  - **`ChannelOutputInt`**

JCSP Channels

- A channel holds a data of some type.
- Channels can either send data to a process (*output channels*), or receive data from a process (*input channels*)

Processes and channels

- When a process dispatch an event through a channel, it stays blocked until the synchronisation with another process occurs.

```
class P implements CSProcess{
 ChannelOutput a;
 public void run() {
  a.write(...);
 }
}
```

```
class Q implements CSProcess{
 ChannelInput a;
 public void run() {
  x = a.read();
 }
}
```

Interfaces for integer channels and objects

```
interface ChannelOutput {
  public void write (Object o);
}

interface ChannelInput {
  public Object read ();
}
```

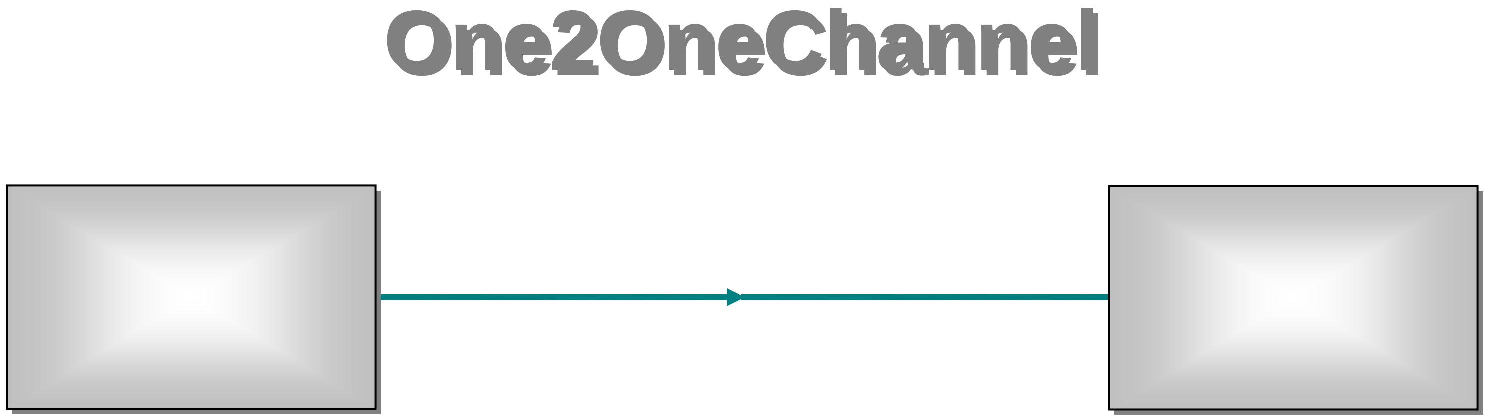
```
interface ChannelOutputInt {
  public void write (int o);
}

interface ChannelInputInt {
  public int read ();
}
```

JCSP Channels

- An input and output channel may be of type:
  - **`One2OneChannel`**
  - **`Any2OneChannel`**
  - **`One2AnyChannel`**
  - **`Any2AnyChannel`**
- The two first types can be used inside ALT constructs.

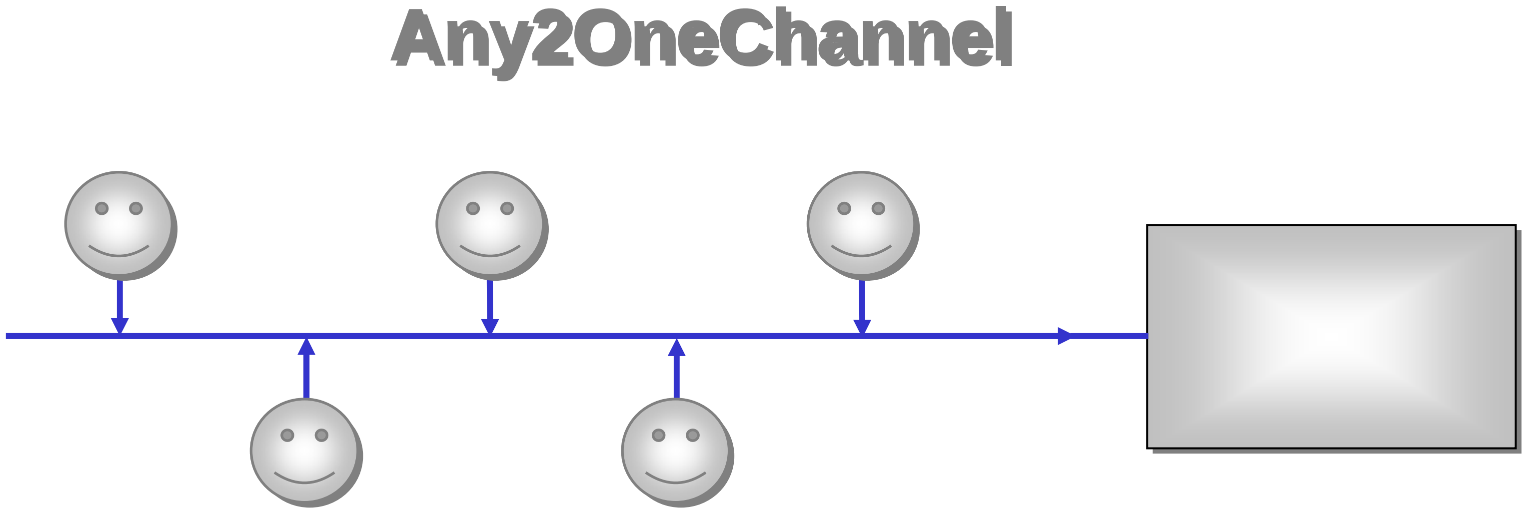
JCSP Channels

- One to one object channel. Allows only one writer and one reader.



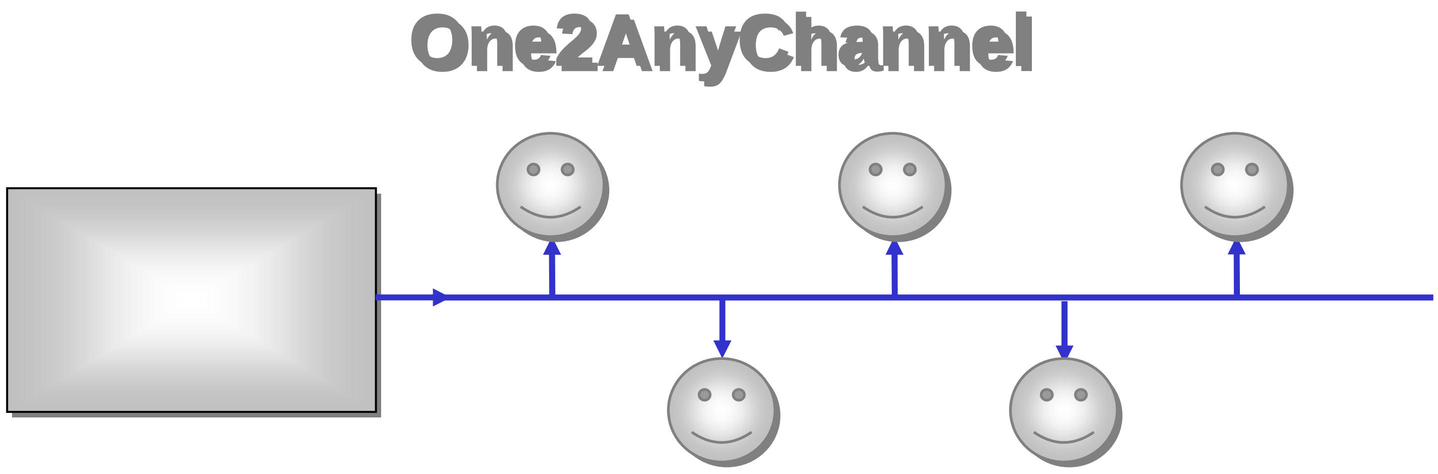
JCSP Channels

- Any2OneChannel. Allows many writers and one reader.



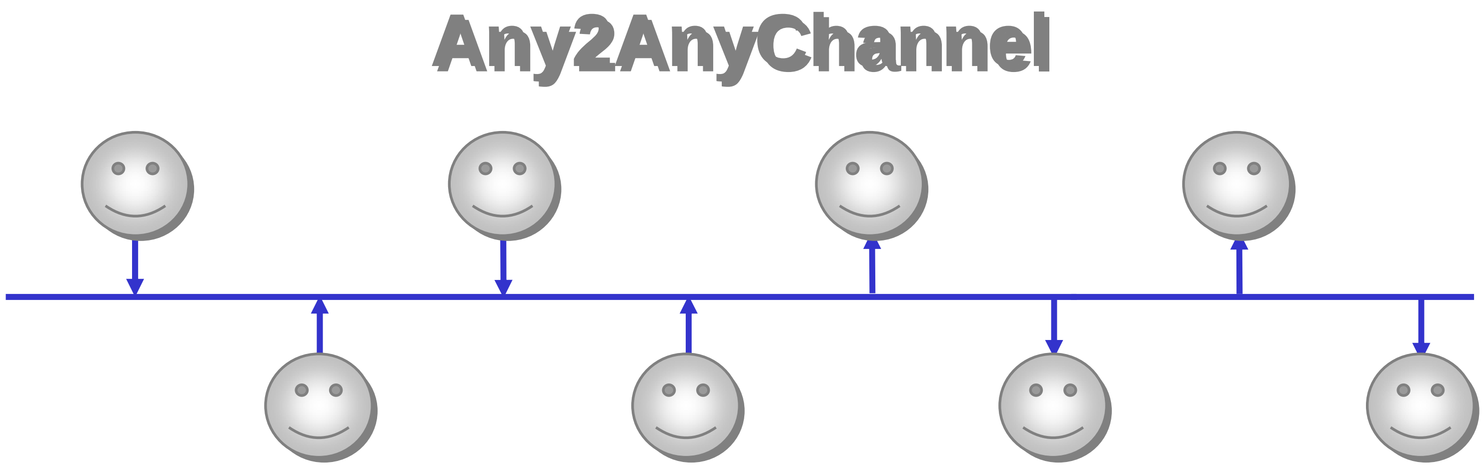
JCSP Channels

- One2AnyChannel. Allows one writer and many readers.



JCSP Channels

- Any2AnyChannel. Allows many writers and many readers.



Channels of objects

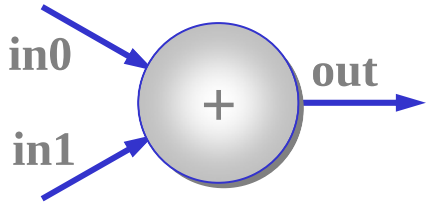
- By default, channels are fully synchronised. At a given time only one reader and only one writer can use the channel.
- **JCSP** ofers a set of plugins of channels that provides several buffering mechanisms.

  (FIFO blocking, overflowing, overwriting, infinite)
- These plugins can be find in **`jcsp.util`**.

Example

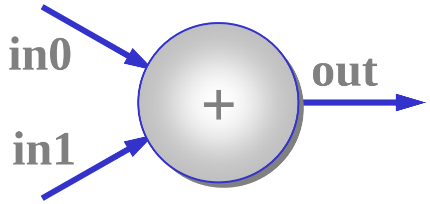

```
class SuccInt implements CSProcess {

  private final ChannelInputInt in;
  private final ChannelOutputInt out;

  public SuccInt (ChannelInputInt in,
                  ChannelOutputInt out) {
    this.in = in;
    this.out = out;
  }

  public void run () {
    while (true) {
      int n = in.read ();
      out.write (n + 1);
    }
  }

}
```

Example




```
class PlusInt implements CSProcess {

  private final ChannelInputInt in0;
  private final ChannelInputInt in1;
  private final ChannelOutputInt out;


  public PlusInt (ChannelInputInt in0,
                  ChannelInputInt in1,
                  ChannelOutputInt out) {
    this.in0 = in0;
    this.in1 = in1;
    this.out = out;
  }

  ...  public void run ()

}
```

Example




```
class PlusInt implements CSProcess {

  ...  private final channels (in0, in1, out)

  ...  public PlusInt (ChannelInputInt in0, ...)

  public void run () {
    while (true) {
      int n0 = in0.read ();
      int n1 = in1.read ();
      out.write (n0 + n1);
    }
  }

}
```

sequential

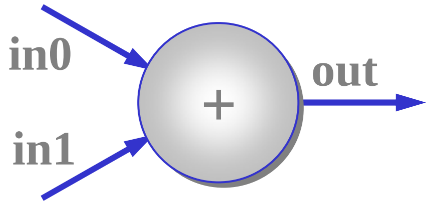
Process Networks

- Process instances (components) may be combined to form a network.
- The resulting network is also a process.
- Components are connected through conectors (channel instances)
- The components execute in ***parallel***.

The class Parallel

- **`Parallel`** is a CSProcess whose constructor has as argument an array of processes.
- The method **`run()`** implements the parallel composition of the argument processes.
- The semantics is the same as the interaction operator in CSP (||).
- The method **`run()`** finishes only when all arguments finish sucessfully.

Example



processes for reading the input in parallel

```
public void run () {

  ProcessReadInt readIn0 = new ProcessReadInt (in0);
  ProcessReadInt readIn1 = new ProcessReadInt (in1);

  CSProcess parRead =
    new Parallel (new CSProcess[] {readIn0, readIn1});

  while (true) {
    parRead.run ();
    out.write (readIn0.value + readIn1.value);
  }

}
```

The Class Parallel

- Ofers methods for adding (**`addProcess`**) and removing processes (**`removeProcess`**)
- However, these methods should be called only when the object is not running.
- If called during the execution, the effect occurs only after the ending of the execution.

Exercise

- Gives a JCSP implementation of the CSP process Main below

Main = Send (0) || Read

Send (i) = chan ! i -> Send (i+1)

Read = chan ? x -> Print(x); Read

Print(x) = ...

Process Send

```
public class Send implements CSProcess {
 private final ChannelOutputInt chan;
 private int i;

 public Send(ChannelOutputInt chan, int i) {
     this.chan = chan;
     this.i = i;
 }

 public void run() {
     while (true) {
         chan.write(i);
         i = i + 1;
     }
 }
}
```

Process Read

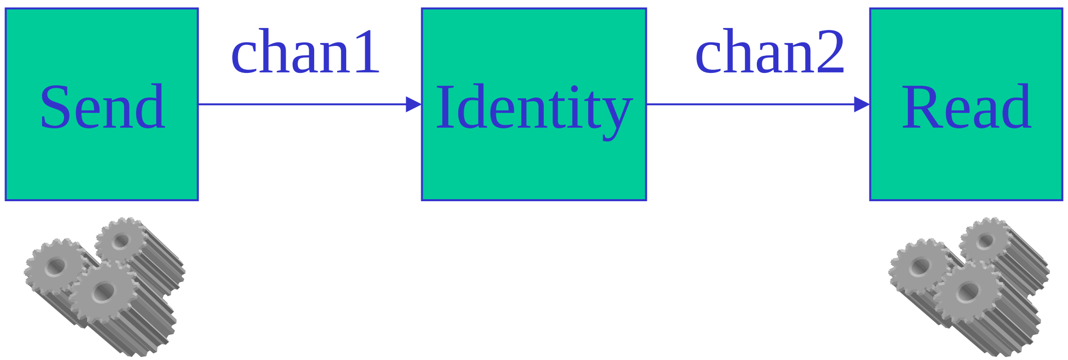
```
public class Read implements CSProcess {
 private final ChannelInputInt chan;

 public Read(ChannelInputInt chan) {
     this.chan = chan;
 }

 public void run() {
     while (true) {
       int i = chan.read();
       System.out.println(i);
     }
 }
}
```

Process Main

```
public class ExampleMain {
 public static void main (String[] args) {

      One2OneChannelInt chan =
           new One2OneChannelInt();

      Send send = new Send(chan);
      Read read = new Read (chan);
      CSProcess[] parArray = {send,read};
      Parallel par = new Parallel (parArray);
      par.run();
 }
}
```

Exercício 2 - Comunicação Assíncrona

```
public class ExampleMain {
 public static void main (String[] argv) {

      One2OneChannelInt chan1 =
                  new One2OneChannelInt ();
      One2OneChannelInt chan2 =
                  new One2OneChannelInt ();

      new Parallel ( new CSProcess[] {
            new Send (chan1),
            new IdentityInt(chan1, chan2),
            new Read (chan2)} ).run ();
 }
}
```



Sequential Composition

- class Sequence (implements CSProcess)
  - CSProcess whose constructor has an array of processes as argument.
  - The method run() implements a sequential composition of the processes in the argument.

Alternative

– Implements (external) choice

– Example:

Selection algorithm

```
ChannelInput ch1, ch2 = ...
Guard[] guards = new Guard[] {ch1, ch2};
boolean[] preconditions = new boolean[] {g1, g2};

Alternative alt = new Alternative(guards);
int indexGuard = alt.select(preconditions);
```

Channel Mapping

- One2OneChannel a = P(a) [|a|] Q(a)
- One2AnyChannel a = P(a) [|a|] (Q1(a) ||| Q2(a))
- Any2OneChannel a = (P1(a) ||| P2(a)) [|a|] Q(a)
- Any2AnyChannel a = (P1(a) ||| P2(a)) [|a|]
  (Q1(a) ||| Q2(a))

The communication is always point to point, because only two processes communicate at each time.

Processes Mapping

- What do we have in CSP?
  1. *P = pre & a -> P*
  2. *P = a?x:{restricao} -> P*
  3. *P = a!x?y -> P*
  4. *P = (a -> P) [] (b -> P)*
  5. *P = (a -> P) |~| (b -> P)*
  6. *P = a -> Q*
  7. *P = (a -> Q) [] (b -> R)*
  8. *P = Q ||| R*
  9. *P = Q || R*
  10. *P = Q |a| R*
  11. *....*

JCSP does not support all CSP constructions!

We will present some of them.

Processes Mapping

P = *pre* & *a* -> *P*

```
P implements CSProcess {
        AltingChannelInput a;
        public void run() {
                boolean pre = ...
                Guard[] guards = new Guard[]{a};
                bollean[] preconditions = new boolean[] {pre};
                Alternative alt = new Alternative(guards);

                while (true) {
                        switch (alt.select(preconditions))
                        case 0:
                                a.read();
                                break;
                }
        }
}
```

Processes Mapping

$P = a?x!y \text{ -> } P$

There are several forms for implementing channels of several data.

Processes Mapping

P = a?x!y -> P

```
P = a?x!y -> ...        // a channel for each data
Q = a!x?y -> ...
```

```
P {     AltingChannelInput ax;
        ChannelOutput ay;
        ...
        DataX valueX = (DataX)ax.read();
        ay.write(valueY);
        ...
}
Q {     ChannelOutput ax;
        AltingChannelInput ay;
        ...
        ax.write(valueX);
        DataY valueY = (DataY)ay.read();
        ...
}
```

Processes Mapping

$P = a!x?y \text{ -> } P$

```
P = a!x?y -> ...        // an input (output) channel
Q = a?x!y -> ...        // for a composite type

P {
        ChannelOutput a;
        ...
        a.write(valueXY);
        ...
}

Q {
        AltingChannelInput a;
        ...
        DataXY valueXY = (DataXY)a.read();
        ...
}
```

The values of x and y must be known

Processes Mapping

P = (a -> P) [] (b -> P)

- The input channels are directly used as guards
- Output channels cannot be used as guards.
  - option 1: create a new input channel to precede the output channel

    ```
    b!out -> ....        user -> b!out -> ...
    ```

  - option 2: use a timeout before the output channel

Processes Mapping

P = (a?x -> P) [] (b!y -> P)

```
P implements CSProcess {
        AltingChannelInput a, user;
        OutputChannel b;
        public void run() {
                Guard[] guards = new Guard[]{a, user};
                Alternative alt = new Alternative(guards);

                while (true) {
                        switch (alt.select())
                        case 0: a.read();
                                break;
                        case 1: user.read();
                                b.write(..);
                                break;
                }
        }
}
```

Processes Mapping

P = (a -> P) |~| (b -> P)

- Notion not very clear
- Non determinism may be implementes through the selection method of Alternatives

  select() – selecs arbitrarily from the list of active guards

  priSelect() – selects the first guard from the list of active guards

  fairSelect() – selects the less active visited guard

Processes Mapping

P = *a* -> *Q*

```
P implements CSProcess {
      ChannelInput a;
      public void run() {
            a.read();
            new Q().run();
      }
}
```

Processes Mapping

P = (a -> Q) [] (b -> R)

```
P implements CSProcess {
        AltingChannelInput a, b;
        public void run() {
                Guard[] guards = new Guard[]{a, b};
                Alternative alt = new Alternative(guards);
                while (true) {
                        switch (alt.select())
                        case 0: a.read();
                                new Q().run();
                                break;
                        case 1: b.read();
                                new R().run();
                                break;
                }
        }
}
```

Processes Mapping

$P = Q \mid\mid\mid Q$

Channels Any2One, One2Any e Any2Any

```
class Q implements CSProcess
{
        ChannelInput a;
        ChannelOutput b;
        a.read();
        ...
        b.write(...);
}
```

```
class Example {
  One2AnyChannel a = new One2AnyChannel();
  Any2OneChannel b = new Any2OneChannel();
  ...
  new Parallel (new CSProcess[] {
                new Q(a,b),
                new Q(a,b),
                new CSProcess () {
                    public void run () {
                       a.write(...);
                       b.read();
                    }
                }
            }).run ();
}
```

Processes Mapping

$P = Q \,||\, R$ and $P = Q \,|a|\, R$

- Channels One2One must be used to assure point to point synchronization (interaction).
- The other channels (One2Any, Any2One, Any2Any) do not guarantee synchronism among all participants.
- The synchronization is achieved by referring the same channel in both Q and R

Processes Mapping

$P = Q \mid\mid R$ and $P = Q \mid a \mid R$

- For each two processes in parallel one channel One2One is used for each event synchronization.

```
// P |a| Q
One2OneChannel a = new One2OneChannel();

new Parallel (
    new CSProcess[] {
        new P(a),
        new Q(a) }
).run ();
```

Processes Mapping

$P = Q \mid\mid R$ and $P = Q \mid a \mid R$

- For three or more processes in parallel na array of channels One2One is used for each event synchronization.

```
// (P |a| Q) |a| R
One2OneChannel[] a = One2OneChannel.create(3);

new Parallel (
    new CSProcess[] {
        new P(a[1], a[2]),
        new Q(a[0], a[2]),
        new R(a[0], a[1]),
    }
).run ();
```

channel with P Channel with Q Channel with R

Other JCSP constructions for communication

- JCSP provides
  - Barriers
  - Buckets
  - Both may synchronize any given number of processes, but do not transfer information.

Exercises

- Implements in JCSP the following processes:

```
VM(c,t) = c > 0 & coffee -> VM(c-1,t)
          []
          t > 0 & tea -> VM(c,t-1)

CLIENT = coffee -> CLIENT
         |~|
         tea -> CLIENT

SYSTEM = VM(10,10) [|{|coffee, tea|}|] CLIENT
```

Useful links

http://www.cs.kent.ac.uk/projects/ofa/jcsp/jcsp1-0-rc7/jcsp-docs/

http://www.cs.kent.ac.uk/projects/ofa/jcsp/