Tuple-based Coordination: From Linda to ReSpecT & TuCSoN

Multiagent Systems LS
Sistemi Multiagente LS

Andrea Omicini
after Matteo Casadei, Elena Nardini, Alessandro Ricci
andrea.omicini@unibo.it

Ingegneria Due
Alma Mater Studiorum—Università di Bologna a Cesena

Academic Year 2009/2010

Our Running Example: The Dining Philosophers Problem

Dining Philosophers [Dijkstra, 2002]

- In the classical Dining Philosopher problem, N philosopher agents share N chopsticks and a spaghetti bowl
- Each philosopher either eats or thinks
- Each philosopher needs a pair of chopsticks to eat—and can access the two chopsticks on his left and on his right
- Each chopstick is shared by two adjacent philosophers
- When a philosopher needs to think, he gets rid of chopsticks

Our Running Example: The Dining Philosophers Problem

Dining Philosophers [Dijkstra, 2002]

- In the classical Dining Philosopher problem, N philosopher agents share N chopsticks and a spaghetti bowl
- Each philosopher either eats or thinks
- Each philosopher needs a pair of chopsticks to eat—and can access the two chopsticks on his left and on his right
- Each chopstick is shared by two adjacent philosophers
- When a philosopher needs to think, he gets rid of chopsticks

Our Running Example: The Dining Philosophers Problem

Dining Philosophers [Dijkstra, 2002]

- In the classical Dining Philosopher problem, N philosopher agents share N chopsticks and a spaghetti bowl
- Each philosopher either eats or thinks
- Each philosopher needs a pair of chopsticks to eat—and can access the two chopsticks on his left and on his right
- Each chopstick is shared by two adjacent philosophers
- When a philosopher needs to think, he gets rid of chopsticks

Our Running Example: The Dining Philosophers Problem

Dining Philosophers [Dijkstra, 2002]

- In the classical Dining Philosopher problem, N philosopher agents share N chopsticks and a spaghetti bowl
- Each philosopher either eats or thinks
- Each philosopher needs a pair of chopsticks to eat—and can access the two chopsticks on his left and on his right
- Each chopstick is shared by two adjacent philosophers
- When a philosopher needs to think, he gets rid of chopsticks

Our Running Example: The Dining Philosophers Problem

Dining Philosophers [Dijkstra, 2002]

- In the classical Dining Philosopher problem, N philosopher agents share N chopsticks and a spaghetti bowl
- Each philosopher either eats or thinks
- Each philosopher needs a pair of chopsticks to eat—and can access the two chopsticks on his left and on his right
- Each chopstick is shared by two adjacent philosophers
- When a philosopher needs to think, he gets rid of chopsticks

Our Running Example: The Dining Philosophers Problem

Dining Philosophers [Dijkstra, 2002]

- In the classical Dining Philosopher problem, N philosopher agents share N chopsticks and a spaghetti bowl
- Each philosopher either eats or thinks
- Each philosopher needs a pair of chopsticks to eat—and can access the two chopsticks on his left and on his right
- Each chopstick is shared by two adjacent philosophers
- When a philosopher needs to think, he gets rid of chopsticks

Concurrency issues in the Dining Philosophers Problem

shared resources Two adjacent philosophers cannot eat simultaneously

starvation If one philosopher eats all the time, the two adjacent philosophers will starve

deadlock If every philosopher picks up the same (say, the left) chopstick at the same time, all of them may wait indefinitely for the other (say, the right) chopstick so as to eat

fairness If a philosopher releases one chopstick before the other one, it favours one of his adjacent philosophers over the other one

Concurrency issues in the Dining Philosophers Problem

shared resources Two adjacent philosophers cannot eat simultaneously

starvation If one philosopher eats all the time, the two adjacent philosophers will starve

deadlock If every philosopher picks up the same (say, the left) chopstick at the same time, all of them may wait indefinitely for the other (say, the right) chopstick so as to eat

fairness If a philosopher releases one chopstick before the other one, it favours one of his adjacent philosophers over the other one

Concurrency issues in the Dining Philosophers Problem

shared resources Two adjacent philosophers cannot eat simultaneously

starvation If one philosopher eats all the time, the two adjacent philosophers will starve

deadlock If every philosopher picks up the same (say, the left) chopstick at the same time, all of them may wait indefinitely for the other (say, the right) chopstick so as to eat

fairness If a philosopher releases one chopstick before the other one, it favours one of his adjacent philosophers over the other one

Concurrency issues in the Dining Philosophers Problem

shared resources Two adjacent philosophers cannot eat simultaneously

starvation If one philosopher eats all the time, the two adjacent philosophers will starve

deadlock If every philosopher picks up the same (say, the left) chopstick at the same time, all of them may wait indefinitely for the other (say, the right) chopstick so as to eat

fairness If a philosopher releases one chopstick before the other one, it favours one of his adjacent philosophers over the other one

Dining Philosophers in Linda

- The spaghetti bowl, or, more easily, the table where the bowl and the chopstick are, and the philosophers are seated, are represented by the tuple space
- Chopsticks are represented as tuples `chop(`i`)`, that represents the left chopstick for the $i-th$ philosopher
  - philosopher i needs chopsticks i (left) and $(i+1) mod N$ (right)
- Philosophers try to eat by getting their chopstick pairs from the tuple space as a pair of tuples `chop(`i`)` `chop(`i`+1 mod N)`
- Philosophers start to think by releasing their own chopstick pairs to the tuple space as a pair of tuples `chop(`i`)` `chop(`i`+1 mod N)`

Dining Philosophers in Linda

- The spaghetti bowl, or, more easily, the table where the bowl and the chopstick are, and the philosophers are seated, are represented by the tuple space
- Chopsticks are represented as tuples `chop(`i`)`, that represents the left chopstick for the $i-th$ philosopher
  - philosopher i needs chopsticks i (left) and $(i+1) mod N$ (right)
- Philosophers try to eat by getting their chopstick pairs from the tuple space as a pair of tuples `chop(`i`)` `chop(`i`+1 mod N)`
- Philosophers start to think by releasing their own chopstick pairs to the tuple space as a pair of tuples `chop(`i`)` `chop(`i`+1 mod N)`

Dining Philosophers in Linda

- The spaghetti bowl, or, more easily, the table where the bowl and the chopstick are, and the philosophers are seated, are represented by the tuple space
- Chopsticks are represented as tuples `chop(`i`)`, that represents the left chopstick for the $i-th$ philosopher
  - philosopher i needs chopsticks i (left) and $(i+1) mod N$ (right)
- Philosophers try to eat by getting their chopstick pairs from the tuple space as a pair of tuples `chop(`i`) chop(`$i+1\ mod\ N$`)`
- Philosophers start to think by releasing their own chopstick pairs to the tuple space as a pair of tuples `chop(`i`) chop(`$i+1\ mod\ N$`)`

Dining Philosophers in Linda

- The spaghetti bowl, or, more easily, the table where the bowl and the chopstick are, and the philosophers are seated, are represented by the tuple space
- Chopsticks are represented as tuples `chop(`i`)`, that represents the left chopstick for the $i-th$ philosopher
  - philosopher i needs chopsticks i (left) and $(i+1) mod N$ (right)
- Philosophers try to eat by getting their chopstick pairs from the tuple space as a pair of tuples `chop(`i`)` `chop(`i`+1 mod N)`
- Philosophers start to think by releasing their own chopstick pairs to the tuple space as a pair of tuples `chop(`i`)` `chop(`i`+1 mod N)`

Dining Philosophers in Linda

- The spaghetti bowl, or, more easily, the table where the bowl and the chopstick are, and the philosophers are seated, are represented by the tuple space
- Chopsticks are represented as tuples `chop(`i`)`, that represents the left chopstick for the $i-th$ philosopher
  - philosopher i needs chopsticks i (left) and $(i+1) mod N$ (right)
- Philosophers try to eat by getting their chopstick pairs from the tuple space as a pair of tuples `chop(`i`)` `chop(`i`+1 mod N)`
- Philosophers start to think by releasing their own chopstick pairs to the tuple space as a pair of tuples `chop(`i`)` `chop(`i`+1 mod N)`

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                          % thinking
    in(chop(I)), in(chop(J)),       % waiting to eat
    eat,                            % eating
    out(chop(I)), out(chop(J)),     % waiting to think
!,  philosopher(I,J).
```

Issues

- \+ shared resources handled correctly
- – starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                          % thinking
    in(chop(I)), in(chop(J)),       % waiting to eat
    eat,                            % eating
    out(chop(I)), out(chop(J)),     % waiting to think
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly
− starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                          % thinking
    in(chop(I)), in(chop(J)),       % waiting to eat
    eat,                            % eating
    out(chop(I)), out(chop(J)),     % waiting to think
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly
− starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)), in(chop(J)),           % waiting to eat
    eat,                                % eating
    out(chop(I)), out(chop(J)),         % waiting to think
!,  philosopher(I,J).
```

Issues

- \+ shared resources handled correctly
- – starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                          % thinking
    in(chop(I)), in(chop(J)),       % waiting to eat
    eat,                            % eating
    out(chop(I)), out(chop(J)),     % waiting to think
!,  philosopher(I,J).
```

Issues

- \+ shared resources handled correctly
- – starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                          % thinking
    in(chop(I)), in(chop(J)),       % waiting to eat
    eat,                            % eating
    out(chop(I)), out(chop(J)),     % waiting to think
!,  philosopher(I,J).
```

Issues

- \+ shared resources handled correctly
- – starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                          % thinking
    in(chop(I)), in(chop(J)),       % waiting to eat
    eat,                            % eating
    out(chop(I)), out(chop(J)),     % waiting to think
!,  philosopher(I,J).
```

Issues

- \+ shared resources handled correctly
- – starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                          % thinking
    in(chop(I)), in(chop(J)),       % waiting to eat
    eat,                            % eating
    out(chop(I)), out(chop(J)),     % waiting to think
!,  philosopher(I,J).
```

Issues

- \+ shared resources handled correctly
- – starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                          % thinking
    in(chop(I)), in(chop(J)),       % waiting to eat
    eat,                            % eating
    out(chop(I)), out(chop(J)),     % waiting to think
!,  philosopher(I,J).
```

Issues

- \+ shared resources handled correctly
- – starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                          % thinking
    in(chop(I)), in(chop(J)),       % waiting to eat
    eat,                            % eating
    out(chop(I)), out(chop(J)),     % waiting to think
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly
− starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)), in(chop(J)),           % waiting to eat
    eat,                                % eating
    out(chop(I)), out(chop(J)),         % waiting to think
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly
− starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)), in(chop(J)),           % waiting to eat
    eat,                                % eating
    out(chop(I)), out(chop(J)),         % waiting to think
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly
– starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)),                        % waiting to eat
    (  inp(chop(J)),                    % if other chop available
       eat,                             % eating
       out(chop(I)), out(chop(J)),      % waiting to think
       ;                                % otherwise
       out(chop(I))                     % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

- \+ shared resources handled correctly, deadlock possibly avoided
- – starvation and unfairness still possible
- – not-so-trivial philosopher's interaction protocol
- – part of the coordination logic is on the coordinables rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)),                        % waiting to eat
    (  inp(chop(J)),                    % if other chop available
       eat,                             % eating
       out(chop(I)), out(chop(J)),      % waiting to think
       ;                                % otherwise
       out(chop(I))                     % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

- \+ shared resources handled correctly, deadlock possibly avoided
- − starvation and unfairness still possible
- − not-so-trivial philosopher's interaction protocol
- − part of the coordination logic in the coordinables
- − rather than in the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)),                        % waiting to eat
    (  inp(chop(J)),                    % if other chop available
       eat,                             % eating
       out(chop(I)), out(chop(J)),      % waiting to think
       ;                                % otherwise
       out(chop(I))                     % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

- \+ shared resources handled correctly, deadlock possibly avoided
- – starvation and unfairness still possible
- – not-so-trivial philosopher's interaction protocol
- – part of the coordination logic is on the coordinables
- – rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                                % thinking
    in(chop(I)),                          % waiting to eat
    ( inp(chop(J)),                       % if other chop available
      eat,                                % eating
      out(chop(I)), out(chop(J)),         % waiting to think
      ;                                   % otherwise
      out(chop(I))                        % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

- \+ shared resources handled correctly, deadlock possibly avoided
- − starvation and unfairness still possible
- − not-so-trivial philosopher's interaction protocol
- − part of the coordination logic is on the coordinables rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                                % thinking
    in(chop(I)),                          % waiting to eat
    (  inp(chop(J)),                      % if other chop available
       eat,                               % eating
       out(chop(I)), out(chop(J)),        % waiting to think
       ;                                  % otherwise
       out(chop(I))                       % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

- \+ shared resources handled correctly, deadlock possibly avoided
- – starvation and unfairness still possible
- – not-so-trivial philosopher's interaction protocol
- – part of the coordination logic in the coordinables rather than in the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)),                        % waiting to eat
    (  inp(chop(J)),                    % if other chop available
       eat,                             % eating
       out(chop(I)), out(chop(J)),      % waiting to think
       ;                                % otherwise
       out(chop(I))                     % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

- \+ shared resources handled correctly, deadlock possibly avoided
- − starvation and unfairness still possible
- − not-so-trivial philosopher's interaction protocol
- − part of the coordination logic in the coordinables rather than in the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)),                        % waiting to eat
    (  inp(chop(J)),                    % if other chop available
       eat,                             % eating
       out(chop(I)), out(chop(J)),      % waiting to think
       ;                                % otherwise
       out(chop(I))                     % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

- \+ shared resources handled correctly, deadlock possibly avoided
- − starvation and unfairness still possible
- − not-so-trivial philosopher's interaction protocol
- − part of the coordination logic in the coordinables rather than in the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)),                        % waiting to eat
    (  inp(chop(J)),                    % if other chop available
       eat,                             % eating
       out(chop(I)), out(chop(J)),      % waiting to think
       ;                                % otherwise
       out(chop(I))                     % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

- \+ shared resources handled correctly, deadlock possibly avoided
- − starvation and unfairness still possible
- − not-so-trivial philosopher's interaction protocol
- − part of the coordination logic in the coordinables
- rather than in the coordination media

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                                % thinking
    in(chop(I)),                          % waiting to eat
    (  inp(chop(J)),                      % if other chop available
       eat,                               % eating
       out(chop(I)), out(chop(J)),        % waiting to think
       ;                                  % otherwise
       out(chop(I))                       % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

- \+ shared resources handled correctly, deadlock possibly avoided
- − starvation and unfairness still possible
- − not-so-trivial philosopher's interaction protocol
  - part of the coordination load is on the coordinables
  - rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                                % thinking
    in(chop(I)),                          % waiting to eat
    (  inp(chop(J)),                      % if other chop available
       eat,                               % eating
       out(chop(I)), out(chop(J)),        % waiting to think
       ;                                  % otherwise
       out(chop(I))                       % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

- \+ shared resources handled correctly, deadlock possibly avoided
- – starvation and unfairness still possible
- – not-so-trivial philosopher's interaction protocol
  - part of the coordination load is on the coordinables
  - rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                                % thinking
    in(chop(I)),                          % waiting to eat
    (  inp(chop(J)),                      % if other chop available
       eat,                               % eating
       out(chop(I)), out(chop(J)),        % waiting to think
       ;                                  % otherwise
       out(chop(I))                       % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

- \+ shared resources handled correctly, deadlock possibly avoided
- − starvation and unfairness still possible
- − not-so-trivial philosopher's interaction protocol
  - part of the coordination load is on the coordinables
  - rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                                % thinking
    in(chop(I)),                          % waiting to eat
    (  inp(chop(J)),                      % if other chop available
       eat,                               % eating
       out(chop(I)), out(chop(J)),        % waiting to think
       ;                                  % otherwise
       out(chop(I))                       % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

- \+ shared resources handled correctly, deadlock possibly avoided
- − starvation and unfairness still possible
- − not-so-trivial philosopher's interaction protocol
  - part of the coordination load is on the coordinables
  - rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                                % thinking
    in(chop(I)),                          % waiting to eat
    (  inp(chop(J)),                      % if other chop available
       eat,                               % eating
       out(chop(I)), out(chop(J)),        % waiting to think
       ;                                  % otherwise
       out(chop(I))                       % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

- \+ shared resources handled correctly, deadlock possibly avoided
- − starvation and unfairness still possible
- − not-so-trivial philosopher's interaction protocol
  - part of the coordination load is on the coordinables
  - rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                                % thinking
    in(chop(I)),                          % waiting to eat
    (  inp(chop(J)),                      % if other chop available
       eat,                               % eating
       out(chop(I)), out(chop(J)),        % waiting to think
       ;                                  % otherwise
       out(chop(I))                       % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

- \+ shared resources handled correctly, deadlock possibly avoided
- − starvation and unfairness still possible
- − not-so-trivial philosopher's interaction protocol
  - part of the coordination load is on the coordinables
  - rather than on the coordination medium

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                  % thinking
    in(chops(I,J)),         % waiting to eat
    eat,                    % eating
    out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Issues

- \+ fairness, no deadlock
- \+ trivial philosopher's interaction protocol
- – shared resources not handled properly
- – starvation still possible

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                  % thinking
    in(chops(I,J)),         % waiting to eat
    eat,                    % eating
    out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Issues

- \+ fairness, no deadlock
- \+ trivial philosopher's interaction protocol
- – shared resources not handled properly
- – starvation still possible

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                  % thinking
    in(chops(I,J)),         % waiting to eat
    eat,                    % eating
    out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Issues

- \+ fairness, no deadlock
- \+ trivial philosopher's interaction protocol
- – shared resources not handled properly
- – starvation still possible

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                  % thinking
    in(chops(I,J)),         % waiting to eat
    eat,                    % eating
    out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Issues

- \+ fairness, no deadlock
- \+ trivial philosopher's interaction protocol
- – shared resources not handled properly
- – starvation still possible

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                  % thinking
    in(chops(I,J)),         % waiting to eat
    eat,                    % eating
    out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Issues

- \+ fairness, no deadlock
- \+ trivial philosopher's interaction protocol
- − shared resources not handled properly
- − starvation still possible

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                  % thinking
    in(chops(I,J)),         % waiting to eat
    eat,                    % eating
    out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Issues

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
- shared resources not handled properly
- starvation still possible

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                  % thinking
    in(chops(I,J)),         % waiting to eat
    eat,                    % eating
    out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Issues

- \+ fairness, no deadlock
- \+ trivial philosopher's interaction protocol
- − shared resources not handled properly
- − starvation still possible

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                  % thinking
    in(chops(I,J)),         % waiting to eat
    eat,                    % eating
    out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Issues

- \+ fairness, no deadlock
- \+ trivial philosopher's interaction protocol
- – shared resources not handled properly
- – starvation still possible

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                  % thinking
    in(chops(I,J)),         % waiting to eat
    eat,                    % eating
    out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Issues

- \+ fairness, no deadlock
- \+ trivial philosopher's interaction protocol
- – shared resources not handled properly
- – starvation still possible

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                  % thinking
    in(chops(I,J)),         % waiting to eat
    eat,                    % eating
    out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Issues

- \+ fairness, no deadlock
- \+ trivial philosopher's interaction protocol
- – shared resources not handled properly
- – starvation still possible

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                  % thinking
    in(chops(I,J)),         % waiting to eat
    eat,                    % eating
    out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Issues

- \+ fairness, no deadlock
- \+ trivial philosopher's interaction protocol
- – shared resources not handled properly
- – starvation still possible

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                  % thinking
    in(chops(I,J)),         % waiting to eat
    eat,                    % eating
    out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Issues

- \+ fairness, no deadlock
- \+ trivial philosopher's interaction protocol
- – shared resources not handled properly
- – starvation still possible

Dining Philosophers in Linda: Where is the Problem?

- Coordination is limited to writing, reading, consuming, suspending on one tuple at a time
  - the behaviour of the coordination medium is fixed once and for all
  - coordination problems that fits it are solved satisfactorily, those that do not fit are not
- Bulk primitives are not a general-purpose solution
  - adding ad hoc primitives does not solve the problem in general
  - and does not fit open scenarios—where instead a limited number of well-known primitives are the perfect solution
- As a result, the coordination load is typically charged upon coordination entities
  - this does not fit open scenarios
  - neither it does follow basic software engineering principles, like encapsulation and locality

Dining Philosophers in Linda: Where is the Problem?

- Coordination is limited to writing, reading, consuming, suspending on one tuple at a time
  - the behaviour of the coordination medium is fixed once and for all
  - coordination problems that fits it are solved satisfactorily, those that do not fit are not
- Bulk primitives are not a general-purpose solution
  - adding ad hoc primitives does not solve the problem in general
  - and does not fit open scenarios—where instead a limited number of well-known primitives are the perfect solution
- As a result, the coordination load is typically charged upon coordination entities
  - this does not fit open scenarios
  - neither it does follow basic software engineering principles, like encapsulation and locality

Dining Philosophers in Linda: Where is the Problem?

- Coordination is limited to writing, reading, consuming, suspending on one tuple at a time
  - the behaviour of the coordination medium is fixed once and for all
  - coordination problems that fits it are solved satisfactorily, those that do not fit are not
- Bulk primitives are not a general-purpose solution
  - adding ad hoc primitives does not solve the problem in general
  - and does not fit open scenarios—where instead a limited number of well-known primitives are the perfect solution
- As a result, the coordination load is typically charged upon coordination entities
  - this does not fit open scenarios
  - neither it does follow basic software engineering principles, like encapsulation and locality

Dining Philosophers in Linda: Where is the Problem?

- Coordination is limited to writing, reading, consuming, suspending on one tuple at a time
  - the behaviour of the coordination medium is fixed once and for all
  - coordination problems that fits it are solved satisfactorily, those that do not fit are not
- Bulk primitives are not a general-purpose solution
  - adding ad hoc primitives does not solve the problem in general
  - and does not fit open scenarios—where instead a limited number of well-known primitives are the perfect solution
- As a result, the coordination load is typically charged upon coordination entities
  - this does not fit open scenarios
  - neither it does follow basic software engineering principles, like encapsulation and locality

Dining Philosophers in Linda: Where is the Problem?

- Coordination is limited to writing, reading, consuming, suspending on one tuple at a time
  - the behaviour of the coordination medium is fixed once and for all
  - coordination problems that fits it are solved satisfactorily, those that do not fit are not
- Bulk primitives are not a general-purpose solution
  - adding ad hoc primitives does not solve the problem in general
  - and does not fit open scenarios—where instead a limited number of well-known primitives are the perfect solution
- As a result, the coordination load is typically charged upon coordination entities
  - this does not fit open scenarios
  - neither it does follow basic software engineering principles, like encapsulation and locality

Dining Philosophers in Linda: Where is the Problem?

- Coordination is limited to writing, reading, consuming, suspending on one tuple at a time
  - the behaviour of the coordination medium is fixed once and for all
  - coordination problems that fits it are solved satisfactorily, those that do not fit are not
- Bulk primitives are not a general-purpose solution
  - adding ad hoc primitives does not solve the problem in general
  - and does not fit open scenarios—where instead a limited number of well-known primitives are the perfect solution
- As a result, the coordination load is typically charged upon coordination entities
  - this does not fit open scenarios
  - neither it does follow basic software engineering principles, like encapsulation and locality

Dining Philosophers in Linda: Where is the Problem?

- Coordination is limited to writing, reading, consuming, suspending on one tuple at a time
  - the behaviour of the coordination medium is fixed once and for all
  - coordination problems that fits it are solved satisfactorily, those that do not fit are not
- Bulk primitives are not a general-purpose solution
  - adding ad hoc primitives does not solve the problem in general
  - and does not fit open scenarios—where instead a limited number of well-known primitives are the perfect solution
- As a result, the coordination load is typically charged upon coordination entities
  - this does not fit open scenarios
  - neither it does follow basic software engineering principles, like encapsulation and locality

Dining Philosophers in Linda: Where is the Problem?

- Coordination is limited to writing, reading, consuming, suspending on one tuple at a time
  - the behaviour of the coordination medium is fixed once and for all
  - coordination problems that fits it are solved satisfactorily, those that do not fit are not
- Bulk primitives are not a general-purpose solution
  - adding ad hoc primitives does not solve the problem in general
  - and does not fit open scenarios—where instead a limited number of well-known primitives are the perfect solution
- As a result, the coordination load is typically charged upon coordination entities
  - this does not fit open scenarios
  - neither it does follow basic software engineering principles, like encapsulation and locality

Dining Philosophers in Linda: Where is the Problem?

- Coordination is limited to writing, reading, consuming, suspending on one tuple at a time
  - the behaviour of the coordination medium is fixed once and for all
  - coordination problems that fits it are solved satisfactorily, those that do not fit are not
- Bulk primitives are not a general-purpose solution
  - adding ad hoc primitives does not solve the problem in general
  - and does not fit open scenarios—where instead a limited number of well-known primitives are the perfect solution
- As a result, the coordination load is typically charged upon coordination entities
  - this does not fit open scenarios
  - neither it does follow basic software engineering principles, like encapsulation and locality

Dining Philosophers in Tuple-based Models: Solution?

- The behaviour of the coordination medium should be *expressive* enough to capture the issues posed by the coordination problem
  - the behaviour of the coordination medium should *not* be fixed once and for all
  - all coordination problems should fit some admissible behaviour of the coordination medium
  - with no need to either add new *ad hoc* primitives, or change the semantics of the old ones
- In this way, coordination media could *encapsulate* solutions to coordination problems
  - represented in terms of coordination policies
  - enacted in terms of coordinative behaviour of the coordination media
- What is needed is a way to *define the behaviour* of a coordination medium according to the specific coordination issues
  - a general *computational model for coordination media*
  - along with a suitably expressive *programming language* to define the behaviour of coordination media

Dining Philosophers in Tuple-based Models: Solution?

- The behaviour of the coordination medium should be *expressive* enough to capture the issues posed by the coordination problem
  - the behaviour of the coordination medium should *not* be fixed once and for all
  - all coordination problems should fit some admissible behaviour of the coordination medium
  - with no need to either add new *ad hoc* primitives, or change the semantics of the old ones
- In this way, coordination media could *encapsulate* solutions to coordination problems
  - represented in terms of coordination policies
  - enacted in terms of coordinative behaviour of the coordination media
- What is needed is a way to *define the behaviour* of a coordination medium according to the specific coordination issues
  - a general *computational model for coordination media*
  - along with a suitably expressive *programming language* to define the behaviour of coordination media

Dining Philosophers in Tuple-based Models: Solution?

- The behaviour of the coordination medium should be *expressive* enough to capture the issues posed by the coordination problem
  - the behaviour of the coordination medium should *not* be fixed once and for all
  - all coordination problems should fit some admissible behaviour of the coordination medium
  - with no need to either add new *ad hoc* primitives, or change the semantics of the old ones
- In this way, coordination media could *encapsulate* solutions to coordination problems
  - represented in terms of coordination policies
  - enacted in terms of coordinative behaviour of the coordination media
- What is needed is a way to *define the behaviour* of a coordination medium according to the specific coordination issues
  - a general *computational model for coordination media*
  - along with a suitably expressive *programming language* to define the behaviour of coordination media

Dining Philosophers in Tuple-based Models: Solution?

- The behaviour of the coordination medium should be *expressive* enough to capture the issues posed by the coordination problem
  - the behaviour of the coordination medium should *not* be fixed once and for all
  - all coordination problems should fit some admissible behaviour of the coordination medium
  - with no need to either add new *ad hoc* primitives, or change the semantics of the old ones
- In this way, coordination media could *encapsulate* solutions to coordination problems
  - represented in terms of coordination policies
  - enacted in terms of coordinative behaviour of the coordination media
- What is needed is a way to *define the behaviour* of a coordination medium according to the specific coordination issues
  - a general *computational model for coordination media*
  - along with a suitably expressive *programming language* to define the behaviour of coordination media

Dining Philosophers in Tuple-based Models: Solution?

- The behaviour of the coordination medium should be *expressive* enough to capture the issues posed by the coordination problem
  - the behaviour of the coordination medium should *not* be fixed once and for all
  - all coordination problems should fit some admissible behaviour of the coordination medium
  - with no need to either add new *ad hoc* primitives, or change the semantics of the old ones
- In this way, coordination media could *encapsulate* solutions to coordination problems
  - represented in terms of coordination policies
  - enacted in terms of coordinative behaviour of the coordination media
- What is needed is a way to *define the behaviour* of a coordination medium according to the specific coordination issues
  - a general *computational model for coordination media*
  - along with a suitably expressive *programming language* to define the behaviour of coordination media

Dining Philosophers in Tuple-based Models: Solution?

- The behaviour of the coordination medium should be *expressive* enough to capture the issues posed by the coordination problem
  - the behaviour of the coordination medium should *not* be fixed once and for all
  - all coordination problems should fit some admissible behaviour of the coordination medium
  - with no need to either add new *ad hoc* primitives, or change the semantics of the old ones
- In this way, coordination media could *encapsulate* solutions to coordination problems
  - represented in terms of coordination policies
  - enacted in terms of coordinative behaviour of the coordination media
- What is needed is a way to *define the behaviour* of a coordination medium according to the specific coordination issues
  - a general *computational model for coordination media*
  - along with a suitably expressive *programming language* to define the behaviour of coordination media

Dining Philosophers in Tuple-based Models: Solution?

- The behaviour of the coordination medium should be *expressive* enough to capture the issues posed by the coordination problem
  - the behaviour of the coordination medium should *not* be fixed once and for all
  - all coordination problems should fit some admissible behaviour of the coordination medium
  - with no need to either add new *ad hoc* primitives, or change the semantics of the old ones
- In this way, coordination media could *encapsulate* solutions to coordination problems
  - represented in terms of coordination policies
  - enacted in terms of coordinative behaviour of the coordination media
- What is needed is a way to *define the behaviour* of a coordination medium according to the specific coordination issues
  - a general *computational model for coordination media*
  - along with a suitably expressive *programming language* to define the behaviour of coordination media

Dining Philosophers in Tuple-based Models: Solution?

- The behaviour of the coordination medium should be *expressive* enough to capture the issues posed by the coordination problem
  - the behaviour of the coordination medium should *not* be fixed once and for all
  - all coordination problems should fit some admissible behaviour of the coordination medium
  - with no need to either add new *ad hoc* primitives, or change the semantics of the old ones
- In this way, coordination media could *encapsulate* solutions to coordination problems
  - represented in terms of coordination policies
  - enacted in terms of coordinative behaviour of the coordination media
- What is needed is a way to *define the behaviour* of a coordination medium according to the specific coordination issues
  - a general *computational model for coordination media*
  - along with a suitably expressive *programming language* to define the behaviour of coordination media

Dining Philosophers in Tuple-based Models: Solution?

- The behaviour of the coordination medium should be *expressive* enough to capture the issues posed by the coordination problem
  - the behaviour of the coordination medium should *not* be fixed once and for all
  - all coordination problems should fit some admissible behaviour of the coordination medium
  - with no need to either add new *ad hoc* primitives, or change the semantics of the old ones
- In this way, coordination media could *encapsulate* solutions to coordination problems
  - represented in terms of coordination policies
  - enacted in terms of coordinative behaviour of the coordination media
- What is needed is a way to *define the behaviour* of a coordination medium according to the specific coordination issues
  - a general *computational model for coordination media*
  - along with a suitably expressive *programming language* to define the behaviour of coordination media

Dining Philosophers in Tuple-based Models: Solution?

- The behaviour of the coordination medium should be *expressive* enough to capture the issues posed by the coordination problem
  - the behaviour of the coordination medium should *not* be fixed once and for all
  - all coordination problems should fit some admissible behaviour of the coordination medium
  - with no need to either add new *ad hoc* primitives, or change the semantics of the old ones
- In this way, coordination media could *encapsulate* solutions to coordination problems
  - represented in terms of coordination policies
  - enacted in terms of coordinative behaviour of the coordination media
- What is needed is a way to *define the behaviour* of a coordination medium according to the specific coordination issues
  - a general *computational model for coordination media*
  - along with a suitably expressive *programming language* to define the behaviour of coordination media

Outline

Data- vs. Control-driven Coordination

- What if we need to start an activity after, say, at least N agents have asked for a resource?
  - More generally, what if we need, in general, to coordinate based on the coordinable actions, rather than on the information available / exchanged?
- Classical distinction in the coordination community
  - data-driven coordination vs. control-driven coordination
- Of course, this does not fit our agent / A&A framework, where (passage of) control is blacklisted
  - *information-driven* coordination vs. *action-driven* coordination clearly fits better
  - but we might as well use the old terms, while we understand their limitations

Data- vs. Control-driven Coordination

- What if we need to start an activity after, say, at least N agents have asked for a resource?
  - More generally, what if we need, in general, to coordinate based on the coordinable actions, rather than on the information available / exchanged?
- Classical distinction in the coordination community
  - data-driven coordination vs. control-driven coordination
- Of course, this does not fit our agent / A&A framework, where (passage of) control is blacklisted
  - *information-driven* coordination vs. *action-driven* coordination clearly fits better
  - but we might as well use the old terms, while we understand their limitations

Data- vs. Control-driven Coordination

- What if we need to start an activity after, say, at least N agents have asked for a resource?
  - More generally, what if we need, in general, to coordinate based on the coordinable actions, rather than on the information available / exchanged?
- Classical distinction in the coordination community
  - data-driven coordination vs. control-driven coordination
- Of course, this does not fit our agent / A&A framework, where (passage of) control is blacklisted
  - *information-driven* coordination vs. *action-driven* coordination clearly fits better
  - but we might as well use the old terms, while we understand their limitations

Data- vs. Control-driven Coordination

- What if we need to start an activity after, say, at least N agents have asked for a resource?
  - More generally, what if we need, in general, to coordinate based on the coordinable actions, rather than on the information available / exchanged?
- Classical distinction in the coordination community
  - data-driven coordination vs. control-driven coordination
- Of course, this does not fit our agent / A&A framework, where (passage of) control is blacklisted
  - *information-driven* coordination vs. *action-driven* coordination clearly fits better
  - but we might as well use the old terms, while we understand their limitations

Data- vs. Control-driven Coordination

- What if we need to start an activity after, say, at least N agents have asked for a resource?
  - More generally, what if we need, in general, to coordinate based on the coordinable actions, rather than on the information available / exchanged?
- Classical distinction in the coordination community
  - data-driven coordination vs. control-driven coordination
- Of course, this does not fit our agent / A&A framework, where (passage of) control is blacklisted
  - *information-driven* coordination vs. *action-driven* coordination clearly fits better
  - but we might as well use the old terms, while we understand their limitations

Data- vs. Control-driven Coordination

- What if we need to start an activity after, say, at least N agents have asked for a resource?
  - More generally, what if we need, in general, to coordinate based on the coordinable actions, rather than on the information available / exchanged?
- Classical distinction in the coordination community
  - data-driven coordination vs. control-driven coordination
- Of course, this does not fit our agent / A&A framework, where (passage of) control is blacklisted
  - *information-driven* coordination vs. *action-driven* coordination clearly fits better
  - but we might as well use the old terms, while we understand their limitations

Data- vs. Control-driven Coordination

- What if we need to start an activity after, say, at least N agents have asked for a resource?
  - More generally, what if we need, in general, to coordinate based on the coordinable actions, rather than on the information available / exchanged?
- Classical distinction in the coordination community
  - data-driven coordination vs. control-driven coordination
- Of course, this does not fit our agent / A&A framework, where (passage of) control is blacklisted
  - *information-driven* coordination vs. *action-driven* coordination clearly fits better
  - but we might as well use the old terms, while we understand their limitations

Hybrid Coordination Models

- Generally speaking, control-driven coordination does not fit so well information-driven contexts, like agent-based ones
  - control-driven models like Reo [Arbab, 2004] need to be adapted to agent-based contexts, mainly to deal with the issue of agent autonomy [Dastani et al., 2005]
  - no coordination medium could say "do this, do that" to a coordinated entity, when a coordinable is an agent
- We need features of both approaches to coordination
  - *hybrid* coordination models
  - adding for instance a control-driven layer to a Linda-based one
- What should be added to a tuple-based model to make it hybrid, and how?

Hybrid Coordination Models

- Generally speaking, control-driven coordination does not fit so well information-driven contexts, like agent-based ones
  - control-driven models like Reo [Arbab, 2004] need to be adapted to agent-based contexts, mainly to deal with the issue of agent autonomy [Dastani et al., 2005]
  - no coordination medium could say "do this, do that" to a coordinated entity, when a coordinable is an agent
- We need features of both approaches to coordination
  - *hybrid* coordination models
  - adding for instance a control-driven layer to a Linda-based one
- What should be added to a tuple-based model to make it hybrid, and how?

Hybrid Coordination Models

- Generally speaking, control-driven coordination does not fit so well information-driven contexts, like agent-based ones
  - control-driven models like Reo [Arbab, 2004] need to be adapted to agent-based contexts, mainly to deal with the issue of agent autonomy [Dastani et al., 2005]
  - no coordination medium could say "do this, do that" to a coordinated entity, when a coordinable is an agent
- We need features of both approaches to coordination
  - *hybrid* coordination models
  - adding for instance a control-driven layer to a Linda-based one
- What should be added to a tuple-based model to make it hybrid, and how?

Hybrid Coordination Models

- Generally speaking, control-driven coordination does not fit so well information-driven contexts, like agent-based ones
  - control-driven models like Reo [Arbab, 2004] need to be adapted to agent-based contexts, mainly to deal with the issue of agent autonomy [Dastani et al., 2005]
  - no coordination medium could say "do this, do that" to a coordinated entity, when a coordinable is an agent
- We need features of both approaches to coordination
  - *hybrid* coordination models
  - adding for instance a control-driven layer to a Linda-based one
- What should be added to a tuple-based model to make it hybrid, and how?

Hybrid Coordination Models

- Generally speaking, control-driven coordination does not fit so well information-driven contexts, like agent-based ones
  - control-driven models like Reo [Arbab, 2004] need to be adapted to agent-based contexts, mainly to deal with the issue of agent autonomy [Dastani et al., 2005]
  - no coordination medium could say "do this, do that" to a coordinated entity, when a coordinable is an agent
- We need features of both approaches to coordination
  - *hybrid* coordination models
  - adding for instance a control-driven layer to a Linda-based one
- What should be added to a tuple-based model to make it hybrid, and how?

Hybrid Coordination Models

- Generally speaking, control-driven coordination does not fit so well information-driven contexts, like agent-based ones
  - control-driven models like Reo [Arbab, 2004] need to be adapted to agent-based contexts, mainly to deal with the issue of agent autonomy [Dastani et al., 2005]
  - no coordination medium could say "do this, do that" to a coordinated entity, when a coordinable is an agent
- We need features of both approaches to coordination
  - *hybrid* coordination models
  - adding for instance a control-driven layer to a Linda-based one
- What should be added to a tuple-based model to make it hybrid, and how?

Hybrid Coordination Models

- Generally speaking, control-driven coordination does not fit so well information-driven contexts, like agent-based ones
  - control-driven models like Reo [Arbab, 2004] need to be adapted to agent-based contexts, mainly to deal with the issue of agent autonomy [Dastani et al., 2005]
  - no coordination medium could say "do this, do that" to a coordinated entity, when a coordinable is an agent
- We need features of both approaches to coordination
  - *hybrid* coordination models
  - adding for instance a control-driven layer to a Linda-based one
- What should be added to a tuple-based model to make it hybrid, and how?

Towards Tuple Centres

- What should be left unchanged?
  - no new primitives
  - basic Linda primitives are preserved, both syntax and semantics
  - matching mechanism preserved, still depending on the communication language of choice
  - multiple tuple spaces, flat name space
- New features from the coordination side
  - ability to define new coordinative behaviours embodying required coordination policies
  - ability to associate coordinative behaviours to coordination events
- New features from the artifact side?
  - the list deriving from the interpretation of coordination media as coordination artifacts

Towards Tuple Centres

- What should be left unchanged?
  - no new primitives
  - basic Linda primitives are preserved, both syntax and semantics
  - matching mechanism preserved, still depending on the communication language of choice
  - multiple tuple spaces, flat name space
- New features from the coordination side
  - ability to define new coordinative behaviours embodying required coordination policies
  - ability to associate coordinative behaviours to coordination events
- New features from the artifact side?
  - the list deriving from the interpretation of coordination media as coordination artifacts

Towards Tuple Centres

- What should be left unchanged?
  - no new primitives
  - basic Linda primitives are preserved, both syntax and semantics
  - matching mechanism preserved, still depending on the communication language of choice
  - multiple tuple spaces, flat name space
- New features from the coordination side
  - ability to define new coordinative behaviours embodying required coordination policies
  - ability to associate coordinative behaviours to coordination events
- New features from the artifact side?
  - the list deriving from the interpretation of coordination media as coordination artifacts

Towards Tuple Centres

- What should be left unchanged?
  - no new primitives
  - basic Linda primitives are preserved, both syntax and semantics
  - matching mechanism preserved, still depending on the communication language of choice
  - multiple tuple spaces, flat name space
- New features from the coordination side
  - ability to define new coordinative behaviours embodying required coordination policies
  - ability to associate coordinative behaviours to coordination events
- New features from the artifact side?
  - the list deriving from the interpretation of coordination media as coordination artifacts

Towards Tuple Centres

- What should be left unchanged?
  - no new primitives
  - basic Linda primitives are preserved, both syntax and semantics
  - matching mechanism preserved, still depending on the communication language of choice
  - multiple tuple spaces, flat name space
- New features from the coordination side
  - ability to define new coordinative behaviours embodying required coordination policies
  - ability to associate coordinative behaviours to coordination events
- New features from the artifact side?
  - the list deriving from the interpretation of coordination media as coordination artifacts

Towards Tuple Centres

- What should be left unchanged?
  - no new primitives
  - basic Linda primitives are preserved, both syntax and semantics
  - matching mechanism preserved, still depending on the communication language of choice
  - multiple tuple spaces, flat name space
- New features from the coordination side
  - ability to define new coordinative behaviours embodying required coordination policies
  - ability to associate coordinative behaviours to coordination events
- New features from the artifact side?
  - the list deriving from the interpretation of coordination media as coordination artifacts

Towards Tuple Centres

- What should be left unchanged?
  - no new primitives
  - basic Linda primitives are preserved, both syntax and semantics
  - matching mechanism preserved, still depending on the communication language of choice
  - multiple tuple spaces, flat name space
- New features from the coordination side
  - ability to define new coordinative behaviours embodying required coordination policies
  - ability to associate coordinative behaviours to coordination events
- New features from the artifact side?
  - the list deriving from the interpretation of coordination media as coordination artifacts

Towards Tuple Centres

- What should be left unchanged?
  - no new primitives
  - basic Linda primitives are preserved, both syntax and semantics
  - matching mechanism preserved, still depending on the communication language of choice
  - multiple tuple spaces, flat name space
- New features from the coordination side
  - ability to define new coordinative behaviours embodying required coordination policies
  - ability to associate coordinative behaviours to coordination events
- New features from the artifact side?
  - the list deriving from the interpretation of coordination media as coordination artifacts

Towards Tuple Centres

- What should be left unchanged?
  - no new primitives
  - basic Linda primitives are preserved, both syntax and semantics
  - matching mechanism preserved, still depending on the communication language of choice
  - multiple tuple spaces, flat name space
- New features from the coordination side
  - ability to define new coordinative behaviours embodying required coordination policies
  - ability to associate coordinative behaviours to coordination events
- New features from the artifact side?
  - the list deriving from the interpretation of coordination media as coordination artifacts

Towards Tuple Centres

- What should be left unchanged?
  - no new primitives
  - basic Linda primitives are preserved, both syntax and semantics
  - matching mechanism preserved, still depending on the communication language of choice
  - multiple tuple spaces, flat name space
- New features from the coordination side
  - ability to define new coordinative behaviours embodying required coordination policies
  - ability to associate coordinative behaviours to coordination events
- New features from the artifact side?
  - the list deriving from the interpretation of coordination media as coordination artifacts

Outline

Feature List: From A&A to Tuple-based Coordination

- Coordinable are agents
  - tuple-space coordination primitives are (communication / pragmatical) actions
- Coordination abstractions are artifacts
  - tuple spaces as specialised artifacts for agent coordination
- Some relevant features of (coordination) artifacts

inspectability & controllability observing / controlling tuple space structure, state & behaviour
- e.g. for monitoring / debugging purposes

malleability / forgeability adapting / changing tuple space function / state & behaviour
- e.g. for incremental development, but also for runtime adaptation & change

linkability & distribution composing distributed tuple spaces
- e.g. for separation of concerns, encapsulation & scalability

situation reacting to environment events & changes
- e.g. reacting to observation that triggers coordination primitives

Feature List: From A&A to Tuple-based Coordination

- Coordinable are agents
  - tuple-space coordination primitives are (communication / pragmatical) actions
- Coordination abstractions are artifacts
  - tuple spaces as specialised artifacts for agent coordination
- Some relevant features of (coordination) artifacts

  inspectability & controllability observing / controlling tuple space structure, state & behaviour
  - e.g. for monitoring / debugging purposes

  malleability / forgeability adapting / changing tuple space function / state & behaviour
  - e.g. for incremental development, but also for runtime adaptation & change

  linkability & distribution composing distributed tuple spaces
  - e.g. for separation of concerns, encapsulation & scalability

  situation reacting to environment events & changes
  - e.g. reacting to observation that involve tuple space coordination primitives

Feature List: From A&A to Tuple-based Coordination

- Coordinable are agents
  - tuple-space coordination primitives are (communication / pragmatical) actions
- Coordination abstractions are artifacts
  - tuple spaces as specialised artifacts for agent coordination
- Some relevant features of (coordination) artifacts

inspectability & controllability observing / controlling tuple space structure, state & behaviour
- e.g. for monitoring / debugging purposes

malleability / forgeability adapting / changing tuple space function / state & behaviour
- e.g. for incremental development, but also for runtime adaptation & change

linkability & distribution composing distributed tuple spaces
- e.g. for separation of concerns, encapsulation & scalability

situation reacting to environment events & changes
- e.g. reacting to observation that involve tuple space coordination primitives

Feature List: From A&A to Tuple-based Coordination

- Coordinable are agents
  - tuple-space coordination primitives are (communication / pragmatical) actions
- Coordination abstractions are artifacts
  - tuple spaces as specialised artifacts for agent coordination
- Some relevant features of (coordination) artifacts

inspectability & controllability observing / controlling tuple space structure, state & behaviour
- e.g. for monitoring / debugging purposes

malleability / forgeability adapting / changing tuple space function / state & behaviour
- e.g. for incremental development, but also for runtime adaptation & change

linkability & distribution composing distributed tuple spaces
- e.g. for separation of concerns, encapsulation & scalability

situation reacting to environment events & changes
- e.g. reacting to observation that triggers coordination primitives

Feature List: From A&A to Tuple-based Coordination

- Coordinable are agents
  - tuple-space coordination primitives are (communication / pragmatical) actions
- Coordination abstractions are artifacts
  - tuple spaces as specialised artifacts for agent coordination
- Some relevant features of (coordination) artifacts

  inspectability & controllability observing / controlling tuple space structure, state & behaviour
  - for monitoring / debugging purposes

  malleability / forgeability adapting / changing tuple space function / state & behaviour
  - for incremental development, but also for run-time adaptation & change

  linkability & distribution composing distributed tuple spaces
  - for separation of concerns, encapsulation & scalability

  situation reacting to environment events & changes
  - reacting to other events rather than invocations of coordination primitives

Feature List: From A&A to Tuple-based Coordination

- Coordinable are agents
  - tuple-space coordination primitives are (communication / pragmatical) actions
- Coordination abstractions are artifacts
  - tuple spaces as specialised artifacts for agent coordination
- Some relevant features of (coordination) artifacts

  inspectability & controllability observing / controlling tuple space structure, state & behaviour
  - for monitoring / debugging purposes

  malleability / forgeability adapting / changing tuple space function / state & behaviour
  - for incremental development, but also for run-time adaptation & change

  linkability & distribution composing distributed tuple spaces
  - for separation of concerns, encapsulation & scalability

  situation reacting to environment events & changes
  - reacting to other events rather than invocations of coordination primitives

Feature List: From A&A to Tuple-based Coordination

- Coordinable are agents
  - tuple-space coordination primitives are (communication / pragmatical) actions
- Coordination abstractions are artifacts
  - tuple spaces as specialised artifacts for agent coordination
- Some relevant features of (coordination) artifacts

  inspectability & controllability observing / controlling tuple space structure, state & behaviour
    - for monitoring / debugging purposes

  malleability / forgeability adapting / changing tuple space function / state & behaviour
    - for incremental development, but also for run-time adaptation & change

  linkability & distribution composing distributed tuple spaces
    - for separation of concerns, encapsulation & scalability

  situation reacting to environment events & changes
    - reacting to other events rather than invocations of coordination primitives

Feature List: From A&A to Tuple-based Coordination

- Coordinable are agents
  - tuple-space coordination primitives are (communication / pragmatical) actions
- Coordination abstractions are artifacts
  - tuple spaces as specialised artifacts for agent coordination
- Some relevant features of (coordination) artifacts

  inspectability & controllability observing / controlling tuple space structure, state & behaviour
    - for monitoring / debugging purposes

  malleability / forgeability adapting / changing tuple space function / state & behaviour
    - for incremental development, but also for run-time adaptation & change

  linkability & distribution composing distributed tuple spaces
    - for separation of concerns, encapsulation & scalability

  situation reacting to environment events & changes
    - reacting to other events rather than invocations of coordination primitives

Feature List: From A&A to Tuple-based Coordination

- Coordinable are agents
  - tuple-space coordination primitives are (communication / pragmatical) actions
- Coordination abstractions are artifacts
  - tuple spaces as specialised artifacts for agent coordination
- Some relevant features of (coordination) artifacts

  inspectability & controllability observing / controlling tuple space structure, state & behaviour
  - for monitoring / debugging purposes

  malleability / forgeability adapting / changing tuple space function / state & behaviour
  - for incremental development, but also for run-time adaptation & change

  linkability & distribution composing distributed tuple spaces
  - for separation of concerns, encapsulation & scalability

  situation reacting to environment events & changes
  - reacting to other events rather than invocations of coordination primitives

Feature List: From A&A to Tuple-based Coordination

- Coordinable are agents
  - tuple-space coordination primitives are (communication / pragmatical) actions
- Coordination abstractions are artifacts
  - tuple spaces as specialised artifacts for agent coordination
- Some relevant features of (coordination) artifacts

  inspectability & controllability observing / controlling tuple space structure, state & behaviour
  - for monitoring / debugging purposes

  malleability / forgeability adapting / changing tuple space function / state & behaviour
  - for incremental development, but also for run-time adaptation & change

  linkability & distribution composing distributed tuple spaces
  - for separation of concerns, encapsulation & scalability

  situation reacting to environment events & changes
  - reacting to other events rather than invocations of coordination primitives

Feature List: From A&A to Tuple-based Coordination

- Coordinable are agents
  - tuple-space coordination primitives are (communication / pragmatical) actions
- Coordination abstractions are artifacts
  - tuple spaces as specialised artifacts for agent coordination
- Some relevant features of (coordination) artifacts

  inspectability & controllability observing / controlling tuple space structure, state & behaviour
  - for monitoring / debugging purposes

  malleability / forgeability adapting / changing tuple space function / state & behaviour
  - for incremental development, but also for run-time adaptation & change

  linkability & distribution composing distributed tuple spaces
  - for separation of concerns, encapsulation & scalability

  situation reacting to environment events & changes
  - reacting to other events rather than invocations of coordination primitives

Feature List: From A&A to Tuple-based Coordination

- Coordinable are agents
  - tuple-space coordination primitives are (communication / pragmatical) actions
- Coordination abstractions are artifacts
  - tuple spaces as specialised artifacts for agent coordination
- Some relevant features of (coordination) artifacts

  inspectability & controllability observing / controlling tuple space structure, state & behaviour
  - for monitoring / debugging purposes

  malleability / forgeability adapting / changing tuple space function / state & behaviour
  - for incremental development, but also for run-time adaptation & change

  linkability & distribution composing distributed tuple spaces
  - for separation of concerns, encapsulation & scalability

  situation reacting to environment events & changes
  - reacting to other events rather than invocations of coordination primitives

Feature List: From A&A to Tuple-based Coordination

- Coordinable are agents
  - tuple-space coordination primitives are (communication / pragmatical) actions
- Coordination abstractions are artifacts
  - tuple spaces as specialised artifacts for agent coordination
- Some relevant features of (coordination) artifacts

  inspectability & controllability observing / controlling tuple space structure, state & behaviour
  - for monitoring / debugging purposes

  malleability / forgeability adapting / changing tuple space function / state & behaviour
  - for incremental development, but also for run-time adaptation & change

  linkability & distribution composing distributed tuple spaces
  - for separation of concerns, encapsulation & scalability

  situation reacting to environment events & changes
  - reacting to other events rather than invocations of coordination primitives

Ideas from the Dining Philosophers

1. Keeping information representation and perception separated
   - in the tuple space
   - this would enable agent interaction protocols to be organised around the desired / required agent perception of the interaction space (tuple space), independently of its *actual* representation in terms of tuples
2. Properly relating information representation and perception through a suitably defined tuple-space behaviour
   - so, agents could get rid of the unnecessary burden of coordination, by embedding coordination laws into the coordination media

In the Dining Philosophers example...

- ...this would amount to
  - representing each chopstick as a single `chop(`i`)` tuple in the tuple space, while
  - enabling philosopher agents to perceive chopsticks as pairs (tuples `chops(`i`,`j`)`), thus
  - allowing agent to acquire / release two chopsticks by means of a single tuple space operation `in(chops(`i`,`j`))` / `out(chops(`i`,`j`))`
- How could we do that, in the example, and in general?

Ideas from the Dining Philosophers

1. Keeping information representation and perception separated
   - in the tuple space
   - this would enable agent interaction protocols to be organised around the desired / required agent perception of the interaction space (tuple space), independently of its *actual* representation in terms of tuples
2. Properly relating information representation and perception through a suitably defined tuple-space behaviour
   - so, agents could get rid of the unnecessary burden of coordination, by embedding coordination laws into the coordination media

In the Dining Philosophers example...

- ...this would amount to
  - representing each chopstick as a single `chop(`i`)` tuple in the tuple space, while
  - enabling philosopher agents to perceive chopsticks as pairs (tuples `chops(`i`,`j`)`), thus
  - allowing agent to acquire / release two chopsticks by means of a single tuple space operation `in(chops(`i`,`j`))` / `out(chops(`i`,`j`))`
- How could we do that, in the example, and in general?

Ideas from the Dining Philosophers

1. Keeping information representation and perception separated
   - in the tuple space
   - this would enable agent interaction protocols to be organised around the desired / required agent perception of the interaction space (tuple space), independently of its *actual* representation in terms of tuples
2. Properly relating information representation and perception through a suitably defined tuple-space behaviour
   - so, agents could get rid of the unnecessary burden of coordination, by embedding coordination laws into the coordination media

In the Dining Philosophers example...

- ...this would amount to
  - representing each chopstick as a single `chop(`i`)` tuple in the tuple space, while
  - enabling philosopher agents to perceive chopsticks as pairs (tuples `chops(`i`,`j`)`), thus
  - allowing agent to acquire / release two chopsticks by means of a single tuple space operation `in(chops(`i`,`j`))` / `out(chops(`i`,`j`))`
- How could we do that, in the example, and in general?

Ideas from the Dining Philosophers

1. Keeping information representation and perception separated
   - in the tuple space
   - this would enable agent interaction protocols to be organised around the desired / required agent perception of the interaction space (tuple space), independently of its *actual* representation in terms of tuples
2. Properly relating information representation and perception through a suitably defined tuple-space behaviour
   - so, agents could get rid of the unnecessary burden of coordination, by embedding coordination laws into the coordination media

In the Dining Philosophers example...

- ...this would amount to
  - representing each chopstick as a single `chop(`i`)` tuple in the tuple space, while
  - enabling philosopher agents to perceive chopsticks as pairs (tuples `chops(`i`,`j`)`), thus
  - allowing agent to acquire / release two chopsticks by means of a single tuple space operation `in(chops(`i`,`j`))` / `out(chops(`i`,`j`))`
- How could we do that, in the example, and in general?

Ideas from the Dining Philosophers

1. Keeping information representation and perception separated
   - in the tuple space
   - this would enable agent interaction protocols to be organised around the desired / required agent perception of the interaction space (tuple space), independently of its *actual* representation in terms of tuples
2. Properly relating information representation and perception through a suitably defined tuple-space behaviour
   - so, agents could get rid of the unnecessary burden of coordination, by embedding coordination laws into the coordination media

In the Dining Philosophers example...

- ...this would amount to
  - representing each chopstick as a single `chop(`i`)` tuple in the tuple space, while
  - enabling philosopher agents to perceive chopsticks as pairs (tuples `chops(`i`,`j`)`), thus
  - allowing agent to acquire / release two chopsticks by means of a single tuple space operation `in(chops(`i`,`j`))` / `out(chops(`i`,`j`))`
- How could we do that, in the example, and in general?

Ideas from the Dining Philosophers

1. Keeping information representation and perception separated
   - in the tuple space
   - this would enable agent interaction protocols to be organised around the desired / required agent perception of the interaction space (tuple space), independently of its *actual* representation in terms of tuples
2. Properly relating information representation and perception through a suitably defined tuple-space behaviour
   - so, agents could get rid of the unnecessary burden of coordination, by embedding coordination laws into the coordination media

In the Dining Philosophers example...

- ...this would amount to
  - representing each chopstick as a single `chop(`i`)` tuple in the tuple space, while
  - enabling philosopher agents to perceive chopsticks as pairs (tuples `chops(`i`,`j`)`), thus
  - allowing agent to acquire / release two chopsticks by means of a single tuple space operation `in(chops(`i`,`j`))` / `out(chops(`i`,`j`))`
- How could we do that, in the example, and in general?

Ideas from the Dining Philosophers

1. Keeping information representation and perception separated
   - in the tuple space
   - this would enable agent interaction protocols to be organised around the desired / required agent perception of the interaction space (tuple space), independently of its *actual* representation in terms of tuples
2. Properly relating information representation and perception through a suitably defined tuple-space behaviour
   - so, agents could get rid of the unnecessary burden of coordination, by embedding coordination laws into the coordination media

In the Dining Philosophers example...

- ...this would amount to
  - representing each chopstick as a single `chop(`i`)` tuple in the tuple space, while
  - enabling philosopher agents to perceive chopsticks as pairs (tuples `chops(`i`,`j`)`), thus
  - allowing agent to acquire / release two chopsticks by means of a single tuple space operation `in(chops(`i`,`j`))` / `out(chops(`i`,`j`))`
- How could we do that, in the example, and in general?

Ideas from the Dining Philosophers

1. Keeping information representation and perception separated
   - in the tuple space
   - this would enable agent interaction protocols to be organised around the desired / required agent perception of the interaction space (tuple space), independently of its *actual* representation in terms of tuples
2. Properly relating information representation and perception through a suitably defined tuple-space behaviour
   - so, agents could get rid of the unnecessary burden of coordination, by embedding coordination laws into the coordination media

In the Dining Philosophers example...

- ...this would amount to
  - representing each chopstick as a single `chop(`i`)` tuple in the tuple space, while
  - enabling philosopher agents to perceive chopsticks as pairs (tuples `chops(`i`,`j`)`), thus
  - allowing agent to acquire / release two chopsticks by means of a single tuple space operation `in(chops(`i`,`j`))` / `out(chops(`i`,`j`))`
- How could we do that, in the example, and in general?

A Possible Solution

- A twofold solution
  1. maintaining the standard tuple space interface
  2. making it possible to enrich the behaviour of a tuple space in terms of the state transitions performed in response to the occurrence of standard communication events
- This is the motivation behind the very notion of *tuple centre*
  - a tuple space whose behaviour in response to communication events is no longer fixed once and for all by the coordination model, but can be defined according to the required coordination policies

Consequences

- Since it has exactly the same interface, a tuple centre is perceived by agents as a standard tuple space
- However, since its behaviour can be specified so as to encapsulate the coordination rules governing agent interaction, a tuple centre may behave in a completely different way with respect to a tuple space

A Possible Solution

- A twofold solution
  1. maintaining the standard tuple space interface
  2. making it possible to enrich the behaviour of a tuple space in terms of the state transitions performed in response to the occurrence of standard communication events
- This is the motivation behind the very notion of *tuple centre*
  - a tuple space whose behaviour in response to communication events is no longer fixed once and for all by the coordination model, but can be defined according to the required coordination policies

Consequences

- Since it has exactly the same interface, a tuple centre is perceived by agents as a standard tuple space
- However, since its behaviour can be specified so as to encapsulate the coordination rules governing agent interaction, a tuple centre may behave in a completely different way with respect to a tuple space

A Possible Solution

- A twofold solution
  1. maintaining the standard tuple space interface
  2. making it possible to enrich the behaviour of a tuple space in terms of the state transitions performed in response to the occurrence of standard communication events
- This is the motivation behind the very notion of *tuple centre*
  - a tuple space whose behaviour in response to communication events is no longer fixed once and for all by the coordination model, but can be defined according to the required coordination policies

Consequences

- Since it has exactly the same interface, a tuple centre is perceived by agents as a standard tuple space
- However, since its behaviour can be specified so as to encapsulate the coordination rules governing agent interaction, a tuple centre may behave in a completely different way with respect to a tuple space

A Possible Solution

- A twofold solution
  1. maintaining the standard tuple space interface
  2. making it possible to enrich the behaviour of a tuple space in terms of the state transitions performed in response to the occurrence of standard communication events
- This is the motivation behind the very notion of *tuple centre*
  - a tuple space whose behaviour in response to communication events is no longer fixed once and for all by the coordination model, but can be defined according to the required coordination policies

Consequences

- Since it has exactly the same interface, a tuple centre is perceived by agents as a standard tuple space
- However, since its behaviour can be specified so as to encapsulate the coordination rules governing agent interaction, a tuple centre may behave in a completely different way with respect to a tuple space

A Possible Solution

- A twofold solution
  1. maintaining the standard tuple space interface
  2. making it possible to enrich the behaviour of a tuple space in terms of the state transitions performed in response to the occurrence of standard communication events
- This is the motivation behind the very notion of *tuple centre*
  - a tuple space whose behaviour in response to communication events is no longer fixed once and for all by the coordination model, but can be defined according to the required coordination policies

Consequences

- Since it has exactly the same interface, a tuple centre is perceived by agents as a standard tuple space
- However, since its behaviour can be specified so as to encapsulate the coordination rules governing agent interaction, a tuple centre may behave in a completely different way with respect to a tuple space

A Possible Solution

- A twofold solution
  1. maintaining the standard tuple space interface
  2. making it possible to enrich the behaviour of a tuple space in terms of the state transitions performed in response to the occurrence of standard communication events
- This is the motivation behind the very notion of *tuple centre*
  - a tuple space whose behaviour in response to communication events is no longer fixed once and for all by the coordination model, but can be defined according to the required coordination policies

Consequences

- Since it has exactly the same interface, a tuple centre is perceived by agents as a standard tuple space
- However, since its behaviour can be specified so as to encapsulate the coordination rules governing agent interaction, a tuple centre may behave in a completely different way with respect to a tuple space

A Possible Solution

- A twofold solution
  1. maintaining the standard tuple space interface
  2. making it possible to enrich the behaviour of a tuple space in terms of the state transitions performed in response to the occurrence of standard communication events
- This is the motivation behind the very notion of *tuple centre*
  - a tuple space whose behaviour in response to communication events is no longer fixed once and for all by the coordination model, but can be defined according to the required coordination policies

Consequences

- Since it has exactly the same interface, a tuple centre is perceived by agents as a standard tuple space
- However, since its behaviour can be specified so as to encapsulate the coordination rules governing agent interaction, a tuple centre may behave in a completely different way with respect to a tuple space

A Possible Solution

- A twofold solution
  1. maintaining the standard tuple space interface
  2. making it possible to enrich the behaviour of a tuple space in terms of the state transitions performed in response to the occurrence of standard communication events
- This is the motivation behind the very notion of *tuple centre*
  - a tuple space whose behaviour in response to communication events is no longer fixed once and for all by the coordination model, but can be defined according to the required coordination policies

Consequences

- Since it has exactly the same interface, a tuple centre is perceived by agents as a standard tuple space
- However, since its behaviour can be specified so as to encapsulate the coordination rules governing agent interaction, a tuple centre may behave in a completely different way with respect to a tuple space

Tuple Centres

Definition [Omicini and Denti, 2001]

- A tuple centre is a tuple space enhanced with a *behaviour specification*, defining the behaviour of a tuple centre in response to interaction events
- The *behaviour specification* of tuple centre
  - is expressed in terms of a *reaction specification language*, and
  - associates any tuple-centre event to a (possibly empty) set of computational activities, which are called *reactions*
- More precisely, a reaction specification language
  - enables the definitions of computational activities within a tuple centre, called reactions, and
  - makes it possible to associate reactions to the events that occur in a tuple centre

Tuple Centres

Definition [Omicini and Denti, 2001]

- A tuple centre is a tuple space enhanced with a *behaviour specification*, defining the behaviour of a tuple centre in response to interaction events
- The *behaviour specification* of tuple centre
  - is expressed in terms of a *reaction specification language*, and
  - associates any tuple-centre event to a (possibly empty) set of computational activities, which are called *reactions*
- More precisely, a reaction specification language
  - enables the definitions of computational activities within a tuple centre, called reactions, and
  - makes it possible to associate reactions to the events that occur in a tuple centre

Tuple Centres

Definition [Omicini and Denti, 2001]

- A tuple centre is a tuple space enhanced with a *behaviour specification*, defining the behaviour of a tuple centre in response to interaction events
- The *behaviour specification* of tuple centre
  - is expressed in terms of a *reaction specification language*, and
  - associates any tuple-centre event to a (possibly empty) set of computational activities, which are called *reactions*
- More precisely, a reaction specification language
  - enables the definitions of computational activities within a tuple centre, called reactions, and
  - makes it possible to associate reactions to the events that occur in a tuple centre

Tuple Centres

Definition [Omicini and Denti, 2001]

- A tuple centre is a tuple space enhanced with a *behaviour specification*, defining the behaviour of a tuple centre in response to interaction events
- The *behaviour specification* of tuple centre
  - is expressed in terms of a *reaction specification language*, and
  - associates any tuple-centre event to a (possibly empty) set of computational activities, which are called *reactions*
- More precisely, a reaction specification language
  - enables the definitions of computational activities within a tuple centre, called reactions, and
  - makes it possible to associate reactions to the events that occur in a tuple centre

Tuple Centres

Definition [Omicini and Denti, 2001]

- A tuple centre is a tuple space enhanced with a *behaviour specification*, defining the behaviour of a tuple centre in response to interaction events
- The *behaviour specification* of tuple centre
  - is expressed in terms of a *reaction specification language*, and
  - associates any tuple-centre event to a (possibly empty) set of computational activities, which are called *reactions*
- More precisely, a reaction specification language
  - enables the definitions of computational activities within a tuple centre, called reactions, and
  - makes it possible to associate reactions to the events that occur in a tuple centre

Tuple Centres

Definition [Omicini and Denti, 2001]

- A tuple centre is a tuple space enhanced with a *behaviour specification*, defining the behaviour of a tuple centre in response to interaction events
- The *behaviour specification* of tuple centre
  - is expressed in terms of a *reaction specification language*, and
  - associates any tuple-centre event to a (possibly empty) set of computational activities, which are called *reactions*
- More precisely, a reaction specification language
  - enables the definitions of computational activities within a tuple centre, called reactions, and
  - makes it possible to associate reactions to the events that occur in a tuple centre

Tuple Centres

Definition [Omicini and Denti, 2001]

- A tuple centre is a tuple space enhanced with a *behaviour specification*, defining the behaviour of a tuple centre in response to interaction events
- The *behaviour specification* of tuple centre
  - is expressed in terms of a *reaction specification language*, and
  - associates any tuple-centre event to a (possibly empty) set of computational activities, which are called *reactions*
- More precisely, a reaction specification language
  - enables the definitions of computational activities within a tuple centre, called reactions, and
  - makes it possible to associate reactions to the events that occur in a tuple centre

Tuple Centres

Definition [Omicini and Denti, 2001]

- A tuple centre is a tuple space enhanced with a *behaviour specification*, defining the behaviour of a tuple centre in response to interaction events
- The *behaviour specification* of tuple centre
  - is expressed in terms of a *reaction specification language*, and
  - associates any tuple-centre event to a (possibly empty) set of computational activities, which are called *reactions*
- More precisely, a reaction specification language
  - enables the definitions of computational activities within a tuple centre, called reactions, and
  - makes it possible to associate reactions to the events that occur in a tuple centre

Reactions

- Each reaction can in principle
  - access and modify the current tuple centre state—like adding or removing tuples)
  - access the information related to the triggering event—such as the performing agent, the primitive invoked, the tuple involved, etc.)—which is made completely observable
  - invoke link primitives upon other tuple centres
- As a result, the semantics of the standard tuple space communication primitives is no longer constrained to be as simple as in the Linda model—i.e., adding, reading, and removing tuples
  - instead, it can be made as complex as required by the specific application needs

Reactions

- Each reaction can in principle
  - access and modify the current tuple centre state—like adding or removing tuples)
  - access the information related to the triggering event—such as the performing agent, the primitive invoked, the tuple involved, etc.)—which is made completely observable
  - invoke link primitives upon other tuple centres
- As a result, the semantics of the standard tuple space communication primitives is no longer constrained to be as simple as in the Linda model—i.e., adding, reading, and removing tuples
  - instead, it can be made as complex as required by the specific application needs

Reactions

- Each reaction can in principle
  - access and modify the current tuple centre state—like adding or removing tuples)
  - access the information related to the triggering event—such as the performing agent, the primitive invoked, the tuple involved, etc.)—which is made completely observable
  - invoke link primitives upon other tuple centres
- As a result, the semantics of the standard tuple space communication primitives is no longer constrained to be as simple as in the Linda model—i.e., adding, reading, and removing tuples
  - instead, it can be made as complex as required by the specific application needs

Reactions

- Each reaction can in principle
  - access and modify the current tuple centre state—like adding or removing tuples)
  - access the information related to the triggering event—such as the performing agent, the primitive invoked, the tuple involved, etc.)—which is made completely observable
  - invoke link primitives upon other tuple centres
- As a result, the semantics of the standard tuple space communication primitives is no longer constrained to be as simple as in the Linda model—i.e., adding, reading, and removing tuples
  - instead, it can be made as complex as required by the specific application needs

Reactions

- Each reaction can in principle
  - access and modify the current tuple centre state—like adding or removing tuples)
  - access the information related to the triggering event—such as the performing agent, the primitive invoked, the tuple involved, etc.)—which is made completely observable
  - invoke link primitives upon other tuple centres
- As a result, the semantics of the standard tuple space communication primitives is no longer constrained to be as simple as in the Linda model—i.e., adding, reading, and removing tuples
  - instead, it can be made as complex as required by the specific application needs

Reactions

- Each reaction can in principle
  - access and modify the current tuple centre state—like adding or removing tuples)
  - access the information related to the triggering event—such as the performing agent, the primitive invoked, the tuple involved, etc.)—which is made completely observable
  - invoke link primitives upon other tuple centres
- As a result, the semantics of the standard tuple space communication primitives is no longer constrained to be as simple as in the Linda model—i.e., adding, reading, and removing tuples
  - instead, it can be made as complex as required by the specific application needs

Reaction Execution

- The main cycle of a tuple centre works as follows
  - when a primitive invocation reaches a tuple centre, all the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the primitive is served in the same way as in standard Linda
  - upon completion of the invocation, the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the main cycle of a tuple centre may go on possibly serving another invocation
- As a result, tuple centres exhibit a couple of fundamental features
  - since an empty behaviour specification brings no triggered reactions independently of the invocation, the behaviour of a tuple centre defaults to a tuple space when no behaviour specification is given
  - from the agent's viewpoint, the result of the invocation of a tuple centre primitive is the sum of the effects of the primitive itself and of all the reactions it triggers, perceived altogether as a single-step transition of the tuple centre state

Reaction Execution

- The main cycle of a tuple centre works as follows
  - when a primitive invocation reaches a tuple centre, all the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the primitive is served in the same way as in standard Linda
  - upon completion of the invocation, the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the main cycle of a tuple centre may go on possibly serving another invocation
- As a result, tuple centres exhibit a couple of fundamental features
  - since an empty behaviour specification brings no triggered reactions independently of the invocation, the behaviour of a tuple centre defaults to a tuple space when no behaviour specification is given
  - from the agent's viewpoint, the result of the invocation of a tuple centre primitive is the sum of the effects of the primitive itself and of all the reactions it triggers, perceived altogether as a single-step transition of the tuple centre state

Reaction Execution

- The main cycle of a tuple centre works as follows
  - when a primitive invocation reaches a tuple centre, all the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the primitive is served in the same way as in standard Linda
  - upon completion of the invocation, the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the main cycle of a tuple centre may go on possibly serving another invocation
- As a result, tuple centres exhibit a couple of fundamental features
  - since an empty behaviour specification brings no triggered reactions independently of the invocation, the behaviour of a tuple centre defaults to a tuple space when no behaviour specification is given
  - from the agent's viewpoint, the result of the invocation of a tuple centre primitive is the sum of the effects of the primitive itself and of all the reactions it triggers, perceived altogether as a single-step transition of the tuple centre state

Reaction Execution

- The main cycle of a tuple centre works as follows
  - when a primitive invocation reaches a tuple centre, all the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the primitive is served in the same way as in standard Linda
  - upon completion of the invocation, the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the main cycle of a tuple centre may go on possibly serving another invocation
- As a result, tuple centres exhibit a couple of fundamental features
  - since an empty behaviour specification brings no triggered reactions independently of the invocation, the behaviour of a tuple centre defaults to a tuple space when no behaviour specification is given
  - from the agent's viewpoint, the result of the invocation of a tuple centre primitive is the sum of the effects of the primitive itself and of all the reactions it triggers, perceived altogether as a single-step transition of the tuple centre state

Reaction Execution

- The main cycle of a tuple centre works as follows
  - when a primitive invocation reaches a tuple centre, all the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the primitive is served in the same way as in standard Linda
  - upon completion of the invocation, the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the main cycle of a tuple centre may go on possibly serving another invocation
- As a result, tuple centres exhibit a couple of fundamental features
  - since an empty behaviour specification brings no triggered reactions independently of the invocation, the behaviour of a tuple centre defaults to a tuple space when no behaviour specification is given
  - from the agent's viewpoint, the result of the invocation of a tuple centre primitive is the sum of the effects of the primitive itself and of all the reactions it triggers, perceived altogether as a single-step transition of the tuple centre state

Reaction Execution

- The main cycle of a tuple centre works as follows
  - when a primitive invocation reaches a tuple centre, all the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the primitive is served in the same way as in standard Linda
  - upon completion of the invocation, the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the main cycle of a tuple centre may go on possibly serving another invocation
- As a result, tuple centres exhibit a couple of fundamental features
  - since an empty behaviour specification brings no triggered reactions independently of the invocation, the behaviour of a tuple centre defaults to a tuple space when no behaviour specification is given
  - from the agent's viewpoint, the result of the invocation of a tuple centre primitive is the sum of the effects of the primitive itself and of all the reactions it triggers, perceived altogether as a single-step transition of the tuple centre state

Reaction Execution

- The main cycle of a tuple centre works as follows
  - when a primitive invocation reaches a tuple centre, all the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the primitive is served in the same way as in standard Linda
  - upon completion of the invocation, the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the main cycle of a tuple centre may go on possibly serving another invocation
- As a result, tuple centres exhibit a couple of fundamental features
  - since an empty behaviour specification brings no triggered reactions independently of the invocation, the behaviour of a tuple centre defaults to a tuple space when no behaviour specification is given
  - from the agent's viewpoint, the result of the invocation of a tuple centre primitive is the sum of the effects of the primitive itself and of all the reactions it triggers, perceived altogether as a single-step transition of the tuple centre state

Reaction Execution

- The main cycle of a tuple centre works as follows
  - when a primitive invocation reaches a tuple centre, all the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the primitive is served in the same way as in standard Linda
  - upon completion of the invocation, the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the main cycle of a tuple centre may go on possibly serving another invocation
- As a result, tuple centres exhibit a couple of fundamental features
  - since an empty behaviour specification brings no triggered reactions independently of the invocation, the behaviour of a tuple centre defaults to a tuple space when no behaviour specification is given
  - from the agent's viewpoint, the result of the invocation of a tuple centre primitive is the sum of the effects of the primitive itself and of all the reactions it triggers, perceived altogether as a single-step transition of the tuple centre state

Tuple Centre's State vs. Agent's Perception

- Reactions are executed in such a way that the observable behaviour of a tuple centre in response to a communication event is still perceived by agents as a single-step transition of the tuple-centre state
  - as in the case of tuple spaces
  - so tuple centres are perceived as tuple spaces by agents
- Unlike a standard tuple space, whose state transitions are constrained to adding, reading or deleting one single tuple, the perceived transition of a tuple centre state can be made as complex as needed
  - this makes it possible to decouple the agent's view of the tuple centre (perceived as a standard tuple space) from the actual state of a tuple centre, and to relate them so as to embed the coordination laws governing the multiagent system

Tuple Centre's State vs. Agent's Perception

- Reactions are executed in such a way that the observable behaviour of a tuple centre in response to a communication event is still perceived by agents as a single-step transition of the tuple-centre state
  - as in the case of tuple spaces
  - so tuple centres are perceived as tuple spaces by agents
- Unlike a standard tuple space, whose state transitions are constrained to adding, reading or deleting one single tuple, the perceived transition of a tuple centre state can be made as complex as needed
  - this makes it possible to decouple the agent's view of the tuple centre (perceived as a standard tuple space) from the actual state of a tuple centre, and to relate them so as to embed the coordination laws governing the multiagent system

Tuple Centre's State vs. Agent's Perception

- Reactions are executed in such a way that the observable behaviour of a tuple centre in response to a communication event is still perceived by agents as a single-step transition of the tuple-centre state
  - as in the case of tuple spaces
  - so tuple centres are perceived as tuple spaces by agents
- Unlike a standard tuple space, whose state transitions are constrained to adding, reading or deleting one single tuple, the perceived transition of a tuple centre state can be made as complex as needed
  - this makes it possible to decouple the agent's view of the tuple centre (perceived as a standard tuple space) from the actual state of a tuple centre, and to relate them so as to embed the coordination laws governing the multiagent system

Tuple Centre's State vs. Agent's Perception

- Reactions are executed in such a way that the observable behaviour of a tuple centre in response to a communication event is still perceived by agents as a single-step transition of the tuple-centre state
  - as in the case of tuple spaces
  - so tuple centres are perceived as tuple spaces by agents
- Unlike a standard tuple space, whose state transitions are constrained to adding, reading or deleting one single tuple, the perceived transition of a tuple centre state can be made as complex as needed
  - this makes it possible to decouple the agent's view of the tuple centre (perceived as a standard tuple space) from the actual state of a tuple centre, and to relate them so as to embed the coordination laws governing the multiagent system

Tuple Centre's State vs. Agent's Perception

- Reactions are executed in such a way that the observable behaviour of a tuple centre in response to a communication event is still perceived by agents as a single-step transition of the tuple-centre state
  - as in the case of tuple spaces
  - so tuple centres are perceived as tuple spaces by agents
- Unlike a standard tuple space, whose state transitions are constrained to adding, reading or deleting one single tuple, the perceived transition of a tuple centre state can be made as complex as needed
  - this makes it possible to decouple the agent's view of the tuple centre (perceived as a standard tuple space) from the actual state of a tuple centre, and to relate them so as to embed the coordination laws governing the multiagent system

Tuple Centres & Hybrid Coordination

- Tuple centres promote a form of hybrid coordination
  - aimed at preserving the advantages of data-driven models
  - while addressing their limitations in terms of control capabilities
- On the one hand, a tuple centre is basically an information-driven coordination medium, which is perceived as such by agents
- On the other hand, a tuple centre also features some capabilities which are typical of action-driven models, like
  - the full observability of events
  - the ability to selectively react to events
  - the ability to implement coordination rules by manipulating the interaction space

Tuple Centres & Hybrid Coordination

- Tuple centres promote a form of hybrid coordination
  - aimed at preserving the advantages of data-driven models
  - while addressing their limitations in terms of control capabilities
- On the one hand, a tuple centre is basically an information-driven coordination medium, which is perceived as such by agents
- On the other hand, a tuple centre also features some capabilities which are typical of action-driven models, like
  - the full observability of events
  - the ability to selectively react to events
  - the ability to implement coordination rules by manipulating the interaction space

Tuple Centres & Hybrid Coordination

- Tuple centres promote a form of hybrid coordination
  - aimed at preserving the advantages of data-driven models
  - while addressing their limitations in terms of control capabilities
- On the one hand, a tuple centre is basically an information-driven coordination medium, which is perceived as such by agents
- On the other hand, a tuple centre also features some capabilities which are typical of action-driven models, like
  - the full observability of events
  - the ability to selectively react to events
  - the ability to implement coordination rules by manipulating the interaction space

Tuple Centres & Hybrid Coordination

- Tuple centres promote a form of hybrid coordination
  - aimed at preserving the advantages of data-driven models
  - while addressing their limitations in terms of control capabilities
- On the one hand, a tuple centre is basically an information-driven coordination medium, which is perceived as such by agents
- On the other hand, a tuple centre also features some capabilities which are typical of action-driven models, like
  - the full observability of events
  - the ability to selectively react to events
  - the ability to implement coordination rules by manipulating the interaction space

Tuple Centres & Hybrid Coordination

- Tuple centres promote a form of hybrid coordination
  - aimed at preserving the advantages of data-driven models
  - while addressing their limitations in terms of control capabilities
- On the one hand, a tuple centre is basically an information-driven coordination medium, which is perceived as such by agents
- On the other hand, a tuple centre also features some capabilities which are typical of action-driven models, like
  - the full observability of events
  - the ability to selectively react to events
  - the ability to implement coordination rules by manipulating the interaction space

Tuple Centres & Hybrid Coordination

- Tuple centres promote a form of hybrid coordination
  - aimed at preserving the advantages of data-driven models
  - while addressing their limitations in terms of control capabilities
- On the one hand, a tuple centre is basically an information-driven coordination medium, which is perceived as such by agents
- On the other hand, a tuple centre also features some capabilities which are typical of action-driven models, like
  - the full observability of events
  - the ability to selectively react to events
  - the ability to implement coordination rules by manipulating the interaction space

Tuple Centres & Hybrid Coordination

- Tuple centres promote a form of hybrid coordination
  - aimed at preserving the advantages of data-driven models
  - while addressing their limitations in terms of control capabilities
- On the one hand, a tuple centre is basically an information-driven coordination medium, which is perceived as such by agents
- On the other hand, a tuple centre also features some capabilities which are typical of action-driven models, like
  - the full observability of events
  - the ability to selectively react to events
  - the ability to implement coordination rules by manipulating the interaction space

Tuple Centres & Hybrid Coordination

- Tuple centres promote a form of hybrid coordination
  - aimed at preserving the advantages of data-driven models
  - while addressing their limitations in terms of control capabilities
- On the one hand, a tuple centre is basically an information-driven coordination medium, which is perceived as such by agents
- On the other hand, a tuple centre also features some capabilities which are typical of action-driven models, like
  - the full observability of events
  - the ability to selectively react to events
  - the ability to implement coordination rules by manipulating the interaction space

Outline

Dining Philosophers in ReSpecT

- The spaghetti bowl, or, more easily, the table where the bowl and the chopstick are, and the philosophers are seated, are represented by tuple centre `table`
- Chopsticks are represented as tuples `chop(`i`)`, that represents the left chopstick for the $i-th$ philosopher
  - philosopher i needs chopsticks i (left) and $(i+1) mod N$ (right)
- An agent philosopher tries to eat by getting his chopstick pair from the tuple centre by means of a `in(chops(`i`,`$i+1$ `mod` N`)` invocation
- A philosopher starts to think by releasing his own chopstick pair to the tuple centre by means of a `out(chops(`i`,`$i+1$ `mod` N`)` invocation

Dining Philosophers in ReSpecT

- The spaghetti bowl, or, more easily, the table where the bowl and the chopstick are, and the philosophers are seated, are represented by tuple centre `table`
- Chopsticks are represented as tuples `chop(`i`)`, that represents the left chopstick for the $i-th$ philosopher
  - philosopher i needs chopsticks i (left) and $(i+1) mod N$ (right)
- An agent philosopher tries to eat by getting his chopstick pair from the tuple centre by means of a `in(chops(`i`,`i`+1 mod N)` invocation
- A philosopher starts to think by releasing his own chopstick pair to the tuple centre by means of a `out(chops(`i`,`i`+1 mod N)` invocation

Dining Philosophers in ReSpecT

- The spaghetti bowl, or, more easily, the table where the bowl and the chopstick are, and the philosophers are seated, are represented by tuple centre `table`
- Chopsticks are represented as tuples `chop(`i`)`, that represents the left chopstick for the $i-th$ philosopher
  - philosopher i needs chopsticks i (left) and $(i+1) mod N$ (right)
- An agent philosopher tries to eat by getting his chopstick pair from the tuple centre by means of a `in(chops(`i`,`i`+1 mod N)` invocation
- A philosopher starts to think by releasing his own chopstick pair to the tuple centre by means of a `out(chops(`i`,`i`+1 mod N)` invocation

Dining Philosophers in ReSpecT

- The spaghetti bowl, or, more easily, the table where the bowl and the chopstick are, and the philosophers are seated, are represented by tuple centre `table`
- Chopsticks are represented as tuples `chop(`i`)`, that represents the left chopstick for the $i-th$ philosopher
  - philosopher i needs chopsticks i (left) and $(i+1) mod N$ (right)
- An agent philosopher tries to eat by getting his chopstick pair from the tuple centre by means of a `in(chops(`i`,`i`+1 mod N)` invocation
- A philosopher starts to think by releasing his own chopstick pair to the tuple centre by means of a `out(chops(`i`,`i`+1 mod N)` invocation

Dining Philosophers in ReSpecT

- The spaghetti bowl, or, more easily, the table where the bowl and the chopstick are, and the philosophers are seated, are represented by tuple centre `table`
- Chopsticks are represented as tuples `chop(`i`)`, that represents the left chopstick for the $i-th$ philosopher
  - philosopher i needs chopsticks i (left) and $(i+1) mod N$ (right)
- An agent philosopher tries to eat by getting his chopstick pair from the tuple centre by means of a `in(chops(`i`,`i`+1 mod N)` invocation
- A philosopher starts to think by releasing his own chopstick pair to the tuple centre by means of a `out(chops(`i`,`i`+1 mod N)` invocation

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Results

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
- ? shared resources handled properly?
- ? starvation still possible?

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Results

- \+ fairness, no deadlock
- \+ trivial philosopher's interaction protocol
- ? shared resources handled properly?
- ? starvation still possible?

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Results

- \+ fairness, no deadlock
- \+ trivial philosopher's interaction protocol
- ? shared resources handled properly?
- ? starvation still possible?

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Results

- \+ fairness, no deadlock
- \+ trivial philosopher's interaction protocol
- ? shared resources handled properly?
- ? starvation still possible?

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Results

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
? shared resources handled properly?
? starvation still possible?

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Results

- \+ fairness, no deadlock
- \+ trivial philosopher's interaction protocol
- ? shared resources handled properly?
- ? starvation still possible?

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Results

- \+ fairness, no deadlock
- \+ trivial philosopher's interaction protocol
- ? shared resources handled properly?
- ? starvation still possible?

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Results

- \+ fairness, no deadlock
- \+ trivial philosopher's interaction protocol
- ? shared resources handled properly?
- ? starvation still possible?

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Results

- \+ fairness, no deadlock
- \+ trivial philosopher's interaction protocol
- ? shared resources handled properly?
- ? starvation still possible?

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Results

- \+ fairness, no deadlock
- \+ trivial philosopher's interaction protocol
- ? shared resources handled properly?
- ? starvation still possible?

Dining Philosophers in ReSpecT: `table` Behaviour Specification

```
reaction( out(chops(C1,C2)), (operation, completion), (     % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (      % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (      % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                 % (4)
    in(chop(C1)), in(chop(C2)),  out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                         % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)),
    out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                         % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)),
    out(chops(C1,C)) )).
```

Dining Philosophers in ReSpecT: `table` Behaviour Specification

```
reaction( out(chops(C1,C2)), (operation, completion), (     % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (      % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (      % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                 % (4)
    in(chop(C1)), in(chop(C2)),  out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                         % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)),
    out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                         % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)),
    out(chops(C1,C)) )).
```

Dining Philosophers in ReSpecT: `table` Behaviour Specification

```
reaction( out(chops(C1,C2)), (operation, completion), (     % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (      % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (      % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                 % (4)
    in(chop(C1)), in(chop(C2)),  out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                         % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)),
    out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                         % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)),
    out(chops(C1,C)) )).
```

Dining Philosophers in ReSpecT: `table` Behaviour Specification

```
reaction( out(chops(C1,C2)), (operation, completion), (     % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (      % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (      % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                 % (4)
    in(chop(C1)), in(chop(C2)),  out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                         % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)),
    out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                         % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)),
    out(chops(C1,C)) )).
```

Dining Philosophers in ReSpecT: `table` Behaviour Specification

```
reaction( out(chops(C1,C2)), (operation, completion), (     % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (      % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (      % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                 % (4)
    in(chop(C1)), in(chop(C2)),  out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                         % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)),
    out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                         % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)),
    out(chops(C1,C)) )).
```

Dining Philosophers in ReSpecT: `table` Behaviour Specification

```
reaction( out(chops(C1,C2)), (operation, completion), (     % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (      % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (      % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                 % (4)
    in(chop(C1)), in(chop(C2)),  out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                         % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)),
    out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                         % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)),
    out(chops(C1,C)) )).
```

Dining Philosophers in ReSpecT: Results

Results

protocol	no deadlock
protocol	fairness
protocol	trivial philosopher's interaction protocol
tuple centre	shared resources handled properly
-	starvation still possible

Dining Philosophers in ReSpecT: Results

Results

protocol	no deadlock
protocol	fairness
protocol	trivial philosopher's interaction protocol
tuple centre	shared resources handled properly
-	starvation still possible

Dining Philosophers in ReSpecT: Results

Results

protocol	no deadlock
protocol	fairness
protocol	trivial philosopher's interaction protocol
tuple centre	shared resources handled properly
-	starvation still possible

Dining Philosophers in ReSpecT: Results

Results

- protocol: no deadlock
- protocol: fairness
- protocol: trivial philosopher's interaction protocol
- tuple centre: shared resources handled properly
- starvation still possible

Dining Philosophers in ReSpecT: Results

Results

protocol	no deadlock
protocol	fairness
protocol	trivial philosopher's interaction protocol
tuple centre	shared resources handled properly
	- starvation still possible

Dining Philosophers in ReSpecT: Results

Results

protocol	no deadlock
protocol	fairness
protocol	trivial philosopher's interaction protocol
tuple centre	shared resources handled properly
-	starvation still possible

Distributed Dining Philosophers

Dining Philosophers in a distributed setting

- N philosopher agents are distributed along the network
  - each philosopher is assigned a seat, represented by the tuple centre `seat(i,j)`
  - `seat(i,j)` denotes that the associated philosopher needs chopstick pair chops(i,j) so as to eat
- each chopstick `i` is represented as a tuple `chop(i)` in the `table` tuple centre
- each philosopher expresses his intention to eat / think by emitting a tuple `wanna_eat` / `wanna_think` in his `seat(i,j)` tuple centre
  - everything else is handled automatically in ReSpecT, embedded in the tuple centre / artifact behaviour
- *N* individual artifacts (`seat(i,j)`) + 1 social artifact (`table`) connected in a star network

Distributed Dining Philosophers

Dining Philosophers in a distributed setting

- N philosopher agents are distributed along the network
  - each philosopher is assigned a seat, represented by the tuple centre `seat(i,j)`
  - `seat(i,j)` denotes that the associated philosopher needs chopstick pair chops(i,j) so as to eat
- each chopstick `i` is represented as a tuple `chop(i)` in the `table` tuple centre
- each philosopher expresses his intention to eat / think by emitting a tuple `wanna_eat` / `wanna_think` in his `seat(i,j)` tuple centre
  - everything else is handled automatically in ReSpecT, embedded in the tuple centre / artifact behaviour
- N individual artifacts (`seat(i,j)`) + 1 social artifact (`table`) connected in a star network

Distributed Dining Philosophers

Dining Philosophers in a distributed setting

- N philosopher agents are distributed along the network
  - each philosopher is assigned a seat, represented by the tuple centre `seat(i,j)`
  - `seat(i,j)` denotes that the associated philosopher needs chopstick pair chops(i,j) so as to eat
- each chopstick `i` is represented as a tuple `chop(i)` in the `table` tuple centre
- each philosopher expresses his intention to eat / think by emitting a tuple `wanna_eat` / `wanna_think` in his `seat(i,j)` tuple centre
  - everything else is handled automatically in ReSpecT, embedded in the tuple centre / artifact behaviour
- *N* individual artifacts (`seat(i,j)`) + 1 social artifact (`table`) connected in a star network

Distributed Dining Philosophers

Dining Philosophers in a distributed setting

- N philosopher agents are distributed along the network
  - each philosopher is assigned a seat, represented by the tuple centre `seat(i,j)`
  - `seat(i,j)` denotes that the associated philosopher needs chopstick pair chops(i,j) so as to eat
- each chopstick `i` is represented as a tuple `chop(i)` in the `table` tuple centre
- each philosopher expresses his intention to eat / think by emitting a tuple `wanna_eat` / `wanna_think` in his `seat(i,j)` tuple centre
  - everything else is handled automatically in ReSpecT, embedded in the tuple centre / artifact behaviour
- *N* individual artifacts (`seat(i,j)`) + 1 social artifact (`table`) connected in a star network

Distributed Dining Philosophers

Dining Philosophers in a distributed setting

- N philosopher agents are distributed along the network
  - each philosopher is assigned a seat, represented by the tuple centre `seat(i,j)`
  - `seat(i,j)` denotes that the associated philosopher needs chopstick pair chops(i,j) so as to eat
- each chopstick `i` is represented as a tuple `chop(i)` in the `table` tuple centre
- each philosopher expresses his intention to eat / think by emitting a tuple `wanna_eat` / `wanna_think` in his `seat(i,j)` tuple centre
  - everything else is handled automatically in ReSpecT, embedded in the tuple centre / artifact behaviour
- *N* individual artifacts (`seat(i,j)`) + 1 social artifact (`table`) connected in a star network

Distributed Dining Philosophers

Dining Philosophers in a distributed setting

- N philosopher agents are distributed along the network
  - each philosopher is assigned a seat, represented by the tuple centre `seat(i,j)`
  - `seat(i,j)` denotes that the associated philosopher needs chopstick pair chops(i,j) so as to eat
- each chopstick `i` is represented as a tuple `chop(i)` in the `table` tuple centre
- each philosopher expresses his intention to eat / think by emitting a tuple `wanna_eat` / `wanna_think` in his `seat(i,j)` tuple centre
  - everything else is handled automatically in ReSpecT, embedded in the tuple centre / artifact behaviour
- *N* individual artifacts (`seat(i,j)`) + 1 social artifact (`table`) connected in a star network

Distributed Dining Philosophers

Dining Philosophers in a distributed setting

- N philosopher agents are distributed along the network
  - each philosopher is assigned a seat, represented by the tuple centre `seat(i,j)`
  - `seat(i,j)` denotes that the associated philosopher needs chopstick pair chops(i,j) so as to eat
- each chopstick `i` is represented as a tuple `chop(i)` in the `table` tuple centre
- each philosopher expresses his intention to eat / think by emitting a tuple `wanna_eat` / `wanna_think` in his `seat(i,j)` tuple centre
  - everything else is handled automatically in ReSpecT, embedded in the tuple centre / artifact behaviour
- *N* individual artifacts (`seat(i,j)`) + 1 social artifact (`table`) connected in a star network

Distributed Dining Philosophers

Dining Philosophers in a distributed setting

- N philosopher agents are distributed along the network
  - each philosopher is assigned a seat, represented by the tuple centre `seat(i,j)`
  - `seat(i,j)` denotes that the associated philosopher needs chopstick pair chops(i,j) so as to eat
- each chopstick `i` is represented as a tuple `chop(i)` in the `table` tuple centre
- each philosopher expresses his intention to eat / think by emitting a tuple `wanna_eat` / `wanna_think` in his `seat(i,j)` tuple centre
  - everything else is handled automatically in ReSpecT, embedded in the tuple centre / artifact behaviour
- *N* individual artifacts (`seat(i,j)`) + 1 social artifact (`table`) connected in a star network

Distributed Dining Philosophers: Individual Interaction

Philosopher–seat interaction (*use*)

- four states, represented by tuple `philosopher(_)`
  - `thinking`, `waiting_to_eat`, `eating`, `waiting_to_think`
- determined by
  - the `out(wanna_eat)` / `out(wanna_think)` invocations, expressing the philosopher's intentions
  - the interaction with the `table` tuple centre, expressing the availability of chop resources
- tuple `chops(i,j)` only occurs in tuple centre `seat(i,j)` in the `philosopher(eating)` state
- state transitions only occur when they are safe
  - from `waiting_to_think` to `thinking` only when chopsticks are safely back on the `table`
  - from `waiting_to_eat` to `eating` only when chopsticks are actually at the seat

Distributed Dining Philosophers: Individual Interaction

Philosopher–seat interaction (*use*)

- four states, represented by tuple `philosopher(_)`
  - `thinking`, `waiting_to_eat`, `eating`, `waiting_to_think`
- determined by
  - the `out(wanna_eat)` / `out(wanna_think)` invocations, expressing the philosopher's intentions
  - the interaction with the `table` tuple centre, expressing the availability of chop resources
- tuple `chops(i,j)` only occurs in tuple centre `seat(i,j)` in the `philosopher(eating)` state
- state transitions only occur when they are safe
  - from `waiting_to_think` to `thinking` only when chopsticks are safely back on the `table`
  - from `waiting_to_eat` to `eating` only when chopsticks are actually at the seat

Distributed Dining Philosophers: Individual Interaction

Philosopher–seat interaction (*use*)

- four states, represented by tuple `philosopher(_)`
  - `thinking`, `waiting_to_eat`, `eating`, `waiting_to_think`
- determined by
  - the `out(wanna_eat)` / `out(wanna_think)` invocations, expressing the philosopher's intentions
  - the interaction with the `table` tuple centre, expressing the availability of chop resources
- tuple `chops(i,j)` only occurs in tuple centre `seat(i,j)` in the `philosopher(eating)` state
- state transitions only occur when they are safe
  - from `waiting_to_think` to `thinking` only when chopsticks are safely back on the table
  - from `waiting_to_eat` to `eating` only when chopsticks are actually at the seat

Distributed Dining Philosophers: Individual Interaction

Philosopher–seat interaction (*use*)

- four states, represented by tuple `philosopher(_)`
  - `thinking`, `waiting_to_eat`, `eating`, `waiting_to_think`
- determined by
  - the `out(wanna_eat)` / `out(wanna_think)` invocations, expressing the philosopher's intentions
  - the interaction with the `table` tuple centre, expressing the availability of `chop` resources
- tuple `chops(i,j)` only occurs in tuple centre `seat(i,j)` in the `philosopher(eating)` state
- state transitions only occur when they are safe
  - from `waiting_to_think` to `thinking` only when chopsticks are safely back on the `table`
  - from `waiting_to_eat` to `eating` only when chopsticks are actually at the `seat`

Distributed Dining Philosophers: Individual Interaction

Philosopher–seat interaction (*use*)

- four states, represented by tuple `philosopher(_)`
  - `thinking`, `waiting_to_eat`, `eating`, `waiting_to_think`
- determined by
  - the `out(wanna_eat)` / `out(wanna_think)` invocations, expressing the philosopher's intentions
  - the interaction with the `table` tuple centre, expressing the availability of `chop` resources
- tuple `chops(i,j)` only occurs in tuple centre `seat(i,j)` in the `philosopher(eating)` state
- state transitions only occur when they are safe
  - from `waiting_to_think` to `thinking` only when chopsticks are safely back on the `table`
  - from `waiting_to_eat` to `eating` only when chopsticks are actually at the `seat`

Distributed Dining Philosophers: Individual Interaction

Philosopher–seat interaction (*use*)

- four states, represented by tuple `philosopher(_)`
  - `thinking`, `waiting_to_eat`, `eating`, `waiting_to_think`
- determined by
  - the `out(wanna_eat)` / `out(wanna_think)` invocations, expressing the philosopher's intentions
  - the interaction with the `table` tuple centre, expressing the availability of `chop` resources
- tuple `chops(i,j)` only occurs in tuple centre `seat(i,j)` in the `philosopher(eating)` state
- state transitions only occur when they are safe
  - from `waiting_to_think` to `thinking` only when chopsticks are safely back on the `table`
  - from `waiting_to_eat` to `eating` only when chopsticks are actually at the `seat`

Distributed Dining Philosophers: Individual Interaction

Philosopher–seat interaction (*use*)

- four states, represented by tuple `philosopher(_)`
  - `thinking`, `waiting_to_eat`, `eating`, `waiting_to_think`
- determined by
  - the `out(wanna_eat)` / `out(wanna_think)` invocations, expressing the philosopher's intentions
  - the interaction with the `table` tuple centre, expressing the availability of `chop` resources
- tuple `chops(i,j)` only occurs in tuple centre `seat(i,j)` in the `philosopher(eating)` state
- state transitions only occur when they are safe
  - from `waiting_to_think` to `thinking` only when chopsticks are safely back on the `table`
  - from `waiting_to_eat` to `eating` only when chopsticks are actually at the `seat`

Distributed Dining Philosophers: Individual Interaction

Philosopher–seat interaction (*use*)

- four states, represented by tuple `philosopher(_)`
  - `thinking`, `waiting_to_eat`, `eating`, `waiting_to_think`
- determined by
  - the `out(wanna_eat)` / `out(wanna_think)` invocations, expressing the philosopher's intentions
  - the interaction with the `table` tuple centre, expressing the availability of `chop` resources
- tuple `chops(i,j)` only occurs in tuple centre `seat(i,j)` in the `philosopher(eating)` state
- state transitions only occur when they are safe
  - from `waiting_to_think` to `thinking` only when chopsticks are safely back on the `table`
  - from `waiting_to_eat` to `eating` only when chopsticks are actually at the `seat`

Distributed Dining Philosophers: Individual Interaction

Philosopher–seat interaction (*use*)

- four states, represented by tuple `philosopher(_)`
  - `thinking`, `waiting_to_eat`, `eating`, `waiting_to_think`
- determined by
  - the `out(wanna_eat)` / `out(wanna_think)` invocations, expressing the philosopher's intentions
  - the interaction with the `table` tuple centre, expressing the availability of `chop` resources
- tuple `chops(i,j)` only occurs in tuple centre `seat(i,j)` in the `philosopher(eating)` state
- state transitions only occur when they are safe
  - from `waiting_to_think` to `thinking` only when chopsticks are safely back on the `table`
  - from `waiting_to_eat` to `eating` only when chopsticks are actually at the `seat`

Distributed Dining Philosophers: Individual Interaction

Philosopher–seat interaction (*use*)

- four states, represented by tuple `philosopher(_)`
  - `thinking`, `waiting_to_eat`, `eating`, `waiting_to_think`
- determined by
  - the `out(wanna_eat)` / `out(wanna_think)` invocations, expressing the philosopher's intentions
  - the interaction with the `table` tuple centre, expressing the availability of `chop` resources
- tuple `chops(i,j)` only occurs in tuple centre `seat(i,j)` in the `philosopher(eating)` state
- state transitions only occur when they are safe
  - from `waiting_to_think` to `thinking` only when chopsticks are safely back on the `table`
  - from `waiting_to_eat` to `eating` only when chopsticks are actually at the `seat`

ReSpecT code for seat(i, j) tuple centres

```
reaction( out(wanna_eat), (operation, invocation), (          % (1)
    in(philosopher(thinking)), out(philosopher(waiting_to_eat)),
    current_target(seat(C1,C2)), table@node ? in(chops(C1,C2)) )).
reaction( out(wanna_eat), (operation, completion),            % (2)
    in(wanna_eat)).
reaction( in(chops(C1,C2)), (link_out, completion), (         % (3)
    in(philosopher(waiting_to_eat)), out(philosopher(eating)),
    out(chops(C1,C2)) )).
reaction( out(wanna_think), (operation, invocation), (        % (4)
    in(philosopher(eating)), out(philosopher(waiting_to_think)),
    current_target(seat(C1,C2)), in(chops(C1,C2)),
    table@node ? out(chops(C1,C2)) )).
reaction( out(wanna_think), (operation, completion),          % (5)
    in(wanna_think) ).
reaction( out(chops(C1,C2)), (link_out, completion), (        % (6)
    in(philosopher(waiting_to_think)), out(philosopher(thinking)) )).
```

Distributed Dining Philosophers: Social Interaction

Seat–table interaction (*link*)

- tuple centre `seat(i,j)` requires / returns tuple `chops(i,j)` from / to `table` tuple centre
- tuple centre `table` transforms tuple `chops(i,j)` into a tuple pair `chop(i)`, `chop(j)` whenever required, and back `chop(i)`, `chop(j)` into `chops(i,j)` whenever required and possible

Distributed Dining Philosophers: Social Interaction

Seat–table interaction (*link*)

- tuple centre `seat(i,j)` requires / returns tuple `chops(i,j)` from / to `table` tuple centre
- tuple centre `table` transforms tuple `chops(i,j)` into a tuple pair `chop(i)`, `chop(j)` whenever required, and back `chop(i)`, `chop(j)` into `chops(i,j)` whenever required and possible

Distributed Dining Philosophers: Social Interaction

Seat–table interaction (*link*)

- tuple centre `seat(i,j)` requires / returns tuple `chops(i,j)` from / to `table` tuple centre
- tuple centre `table` transforms tuple `chops(i,j)` into a tuple pair `chop(i)`, `chop(j)` whenever required, and back `chop(i)`, `chop(j)` into `chops(i,j)` whenever required and possible

ReSpecT code for table tuple centre

```
reaction( out(chops(C1,C2)), (link_in, completion), (          %
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (link_in, invocation), (           %
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (link_in, completion), (           %
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                    %
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                            %
    rd(required(C,C2)), in(chop(C)), in(chop(C2)),
    out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                            %
    rd(required(C1,C)), in(chop(C1)), in(chop(C)),
    out(chops(C1,C)) )).
```

Distributed Dining Philosophers: Features

- Full separation of concerns
  - philosopher agents just express their intentions, in terms of simple tuples
  - individual artifacts (`seat(i,j)` tuple centres) handle individual behaviours and state, and mediate interaction of individuals with social artifacts (`table` tuple centre)
  - the social artifact (`table` tuple centre) deals with shared resources (`chop` tuples) and ensures global system properties, like fairness and deadlock avoidance
- At any time, one could look at the coordination artifacts, and find exactly the consistent representation of the current distributed state
  - properly distributed, suitably encapsulated
    - the state of shared resources is in the shared centre, and is updated
    - the state of single agents is into individual local interactions
  - accessible, represented in a declarative way
    - the state of individual philosophers is exposed through observable artifacts as far as the portion representing their respective interests is concerned

Distributed Dining Philosophers: Features

- Full separation of concerns
  - philosopher agents just express their intentions, in terms of simple tuples
  - individual artifacts (`seat(i,j)` tuple centres) handle individual behaviours and state, and mediate interaction of individuals with social artifacts (`table` tuple centre)
  - the social artifact (`table` tuple centre) deals with shared resources (`chop` tuples) and ensures global system properties, like fairness and deadlock avoidance
- At any time, one could look at the coordination artifacts, and find exactly the consistent representation of the current distributed state
  - properly distributed, suitably encapsulated
    - the state of shared resources is in the shared centre, and is updated
    - the state of single agents is into individual local abstractions
  - accessible, represented in a declarative way
    - the state of individual philosophers is expressed in a declarative way
    - artifacts are free to provide representations concerned

Distributed Dining Philosophers: Features

- Full separation of concerns
  - philosopher agents just express their intentions, in terms of simple tuples
  - individual artifacts (`seat(i,j)` tuple centres) handle individual behaviours and state, and mediate interaction of individuals with social artifacts (`table` tuple centre)
  - the social artifact (`table` tuple centre) deals with shared resources (`chop` tuples) and ensures global system properties, like fairness and deadlock avoidance
- At any time, one could look at the coordination artifacts, and find exactly the consistent representation of the current distributed state
  - properly distributed, suitably encapsulated
    - the state of shared resources is in the shared coordination artifact
    - the state of single agents is into individual local interactions
  - accessible, represented in a declarative way
    - the state of individual philosophers is expressed in the coordination artifacts as far as the portion representing their interaction is concerned

Distributed Dining Philosophers: Features

- Full separation of concerns
  - philosopher agents just express their intentions, in terms of simple tuples
  - individual artifacts (`seat(i,j)` tuple centres) handle individual behaviours and state, and mediate interaction of individuals with social artifacts (`table` tuple centre)
  - the social artifact (`table` tuple centre) deals with shared resources (`chop` tuples) and ensures global system properties, like fairness and deadlock avoidance
- At any time, one could look at the coordination artifacts, and find exactly the consistent representation of the current distributed state
  - properly distributed, suitably encapsulated
    - the state of shared resources is in the shared distributed abstraction, the state of single agents is into individual local abstractions
  - accessible, represented in a declarative way
    - the state of individual philosophers is exposed through accessible artifacts as far as the portion representing their social interaction is concerned

Distributed Dining Philosophers: Features

- Full separation of concerns
  - philosopher agents just express their intentions, in terms of simple tuples
  - individual artifacts (`seat(i,j)` tuple centres) handle individual behaviours and state, and mediate interaction of individuals with social artifacts (`table` tuple centre)
  - the social artifact (`table` tuple centre) deals with shared resources (`chop` tuples) and ensures global system properties, like fairness and deadlock avoidance
- At any time, one could look at the coordination artifacts, and find exactly the consistent representation of the current distributed state
  - properly distributed, suitably encapsulated
    - the state of shared resources is in the shared distributed abstraction, the state of single agents is into individual local abstractions
  - accessible, represented in a declarative way
    - the state of individual philosophers is exposed through accessible artifacts as far as the portion representing their social interaction is concerned

Distributed Dining Philosophers: Features

- Full separation of concerns
  - philosopher agents just express their intentions, in terms of simple tuples
  - individual artifacts (`seat(i,j)` tuple centres) handle individual behaviours and state, and mediate interaction of individuals with social artifacts (`table` tuple centre)
  - the social artifact (`table` tuple centre) deals with shared resources (`chop` tuples) and ensures global system properties, like fairness and deadlock avoidance
- At any time, one could look at the coordination artifacts, and find exactly the consistent representation of the current distributed state
  - properly distributed, suitably encapsulated
    - the state of shared resources is in the shared distributed abstraction, the state of single agents is into individual local abstractions
  - accessible, represented in a declarative way
    - the state of individual philosophers is exposed through accessible artifacts as far as the portion representing their social interaction is concerned

Distributed Dining Philosophers: Features

- Full separation of concerns
  - philosopher agents just express their intentions, in terms of simple tuples
  - individual artifacts (`seat(i,j)` tuple centres) handle individual behaviours and state, and mediate interaction of individuals with social artifacts (`table` tuple centre)
  - the social artifact (`table` tuple centre) deals with shared resources (`chop` tuples) and ensures global system properties, like fairness and deadlock avoidance
- At any time, one could look at the coordination artifacts, and find exactly the consistent representation of the current distributed state
  - properly distributed, suitably encapsulated
    - the state of shared resources is in the shared distributed abstraction, the state of single agents is into individual local abstractions
  - accessible, represented in a declarative way
    - the state of individual philosophers is exposed through accessible artifacts as far as the portion representing their social interaction is concerned

Distributed Dining Philosophers: Features

- Full separation of concerns
  - philosopher agents just express their intentions, in terms of simple tuples
  - individual artifacts (`seat(i,j)` tuple centres) handle individual behaviours and state, and mediate interaction of individuals with social artifacts (`table` tuple centre)
  - the social artifact (`table` tuple centre) deals with shared resources (`chop` tuples) and ensures global system properties, like fairness and deadlock avoidance
- At any time, one could look at the coordination artifacts, and find exactly the consistent representation of the current distributed state
  - properly distributed, suitably encapsulated
    - the state of shared resources is in the shared distributed abstraction, the state of single agents is into individual local abstractions
  - accessible, represented in a declarative way
    - the state of individual philosophers is exposed through accessible artifacts as far as the portion representing their social interaction is concerned

Distributed Dining Philosophers: Features

- Full separation of concerns
  - philosopher agents just express their intentions, in terms of simple tuples
  - individual artifacts (`seat(i,j)` tuple centres) handle individual behaviours and state, and mediate interaction of individuals with social artifacts (`table` tuple centre)
  - the social artifact (`table` tuple centre) deals with shared resources (`chop` tuples) and ensures global system properties, like fairness and deadlock avoidance
- At any time, one could look at the coordination artifacts, and find exactly the consistent representation of the current distributed state
  - properly distributed, suitably encapsulated
    - the state of shared resources is in the shared distributed abstraction, the state of single agents is into individual local abstractions
  - accessible, represented in a declarative way
    - the state of individual philosophers is exposed through accessible artifacts as far as the portion representing their social interaction is concerned

Timed Dining Philosophers

- An example for situatedness in the spatio-temporal fabric
- `table` tuple centre stores the maximum amount of time for any agent (philosopher) to use the resource (to eat using chops)
  - in terms of a tuple `max_eating_time(@Time)`
  - if this time expires the locks are automatically released—chopsticks are re-inserted by the table tuple centre
  - late releases (by agents through `seat` tuple centres) are to be ignored—linkability used to make `seat` tuple centres consistent
- With a very simple extension using timed reactions, Distributed Timed Dining Philosophers are done
  - see [Omicini et al., 2005b]

Timed Dining Philosophers

- An example for situatedness in the spatio-temporal fabric
- `table` tuple centre stores the maximum amount of time for any agent (philosopher) to use the resource (to eat using chops)
  - in terms of a tuple `max_eating_time(@Time)`
  - if this time expires the locks are automatically released—chopsticks are re-inserted by the table tuple centre
  - late releases (by agents through `seat` tuple centres) are to be ignored—linkability used to make `seat` tuple centres consistent
- With a very simple extension using timed reactions, Distributed Timed Dining Philosophers are done
  - see [Omicini et al., 2005b]

Timed Dining Philosophers

- An example for situatedness in the spatio-temporal fabric
- `table` tuple centre stores the maximum amount of time for any agent (philosopher) to use the resource (to eat using chops)
  - in terms of a tuple `max_eating_time(@Time)`
  - if this time expires the locks are automatically released—chopsticks are re-inserted by the table tuple centre
  - late releases (by agents through `seat` tuple centres) are to be ignored—linkability used to make `seat` tuple centres consistent
- With a very simple extension using timed reactions, Distributed Timed Dining Philosophers are done
  - see [Omicini et al., 2005b]

Timed Dining Philosophers

- An example for situatedness in the spatio-temporal fabric
- `table` tuple centre stores the maximum amount of time for any agent (philosopher) to use the resource (to eat using chops)
  - in terms of a tuple `max_eating_time(@Time)`
  - if this time expires the locks are automatically released—chopsticks are re-inserted by the table tuple centre
  - late releases (by agents through `seat` tuple centres) are to be ignored—linkability used to make `seat` tuple centres consistent
- With a very simple extension using timed reactions, Distributed Timed Dining Philosophers are done
  - see [Omicini et al., 2005b]

Timed Dining Philosophers

- An example for situatedness in the spatio-temporal fabric
- `table` tuple centre stores the maximum amount of time for any agent (philosopher) to use the resource (to eat using chops)
  - in terms of a tuple `max_eating_time(@Time)`
  - if this time expires the locks are automatically released—chopsticks are re-inserted by the table tuple centre
  - late releases (by agents through `seat` tuple centres) are to be ignored—linkability used to make `seat` tuple centres consistent
- With a very simple extension using timed reactions, Distributed Timed Dining Philosophers are done
  - see [Omicini et al., 2005b]

Timed Dining Philosophers

- An example for situatedness in the spatio-temporal fabric
- `table` tuple centre stores the maximum amount of time for any agent (philosopher) to use the resource (to eat using chops)
  - in terms of a tuple `max_eating_time(@Time)`
  - if this time expires the locks are automatically released—chopsticks are re-inserted by the table tuple centre
  - late releases (by agents through `seat` tuple centres) are to be ignored—linkability used to make `seat` tuple centres consistent
- With a very simple extension using timed reactions, Distributed Timed Dining Philosophers are done
  - see [Omicini et al., 2005b]

Timed Dining Philosophers

- An example for situatedness in the spatio-temporal fabric
- `table` tuple centre stores the maximum amount of time for any agent (philosopher) to use the resource (to eat using chops)
  - in terms of a tuple `max_eating_time(@Time)`
  - if this time expires the locks are automatically released—chopsticks are re-inserted by the table tuple centre
  - late releases (by agents through `seat` tuple centres) are to be ignored—linkability used to make `seat` tuple centres consistent
- With a very simple extension using timed reactions, Distributed Timed Dining Philosophers are done
  - see [Omicini et al., 2005b]

Timed Dining Philosophers: Philosopher

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

With respect to Dining Philosopher's protocol...

...this is left unchanged

Timed Dining Philosophers: Philosopher

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

With respect to Dining Philosopher's protocol...

...this is left unchanged

Timed Dining Philosophers: `table` ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (     % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).


reaction( in(chops(C1,C2)), (operation, invocation), (      % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (      % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                 % (4)
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                         % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                         % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)) )).
reaction( in(chops(C1,C2)), (operation, completion), (      % (6)
    current_time(T), rd(max eating time(Max)), T1 is T + Max,
    out(used(C1,C2,T)),
    out_s(time(T1),(in(used(C1,C2,T)), out(chop(C1)), out(chop(C2)))) )).
```

Timed Dining Philosophers: `table` ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (      % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).

reaction( in(chops(C1,C2)), (operation, invocation), (       % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                  % (4)
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                          % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                          % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (6)
    current_time(T), rd(max eating time(Max)), T1 is T + Max,
    out(used(C1,C2,T)),
    out_s(time(T1),(in(used(C1,C2,T)), out(chop(C1)), out(chop(C2)))) )).
```

Timed Dining Philosophers: `table` ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (      % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).


reaction( in(chops(C1,C2)), (operation, invocation), (       % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                  % (4)
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                          % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                          % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (6)
    current_time(T), rd(max eating time(Max)), T1 is T + Max,
    out(used(C1,C2,T)),
    out_s(time(T1),(in(used(C1,C2,T)), out(chop(C1)), out(chop(C2)))) )).
```

Timed Dining Philosophers: table ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (     % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).


reaction( in(chops(C1,C2)), (operation, invocation), (      % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (      % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                 % (4)
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                         % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                         % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)) )).
reaction( in(chops(C1,C2)), (operation, completion), (      % (6)
    current_time(T), rd(max eating time(Max)), T1 is T + Max,
    out(used(C1,C2,T)),
    out_s(time(T1),(in(used(C1,C2,T)), out(chop(C1)), out(chop(C2)))) )).
```

Timed Dining Philosophers: table ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (      % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).


reaction( in(chops(C1,C2)), (operation, invocation), (       % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                  % (4)
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                          % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                          % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (6)
    current_time(T), rd(max eating time(Max)), T1 is T + Max,
    out(used(C1,C2,T)),
    out_s(time(T1),(in(used(C1,C2,T)), out(chop(C1)), out(chop(C2)))) )).
```

Timed Dining Philosophers: `table` ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (      % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).


reaction( in(chops(C1,C2)), (operation, invocation), (       % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                  % (4)
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                          % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                          % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (6)
    current_time(T), rd(max eating time(Max)), T1 is T + Max,
    out(used(C1,C2,T)),
    out_s(time(T1),(in(used(C1,C2,T)), out(chop(C1)), out(chop(C2)))) )).
```

Timed Dining Philosophers: `table` ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (     % (1)
    in(chops(C1,C2)) )).
reaction( out(chops(C1,C2)), (operation, completion), (     % (1')
    out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (      % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (      % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                 % (4)
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                         % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                         % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)) )).
reaction( in(chops(C1,C2)), (operation, completion), (      % (6)
    current_time(T), rd(max eating time(Max)), T1 is T + Max,
    out(used(C1,C2,T)),
    out_s(time(T1),(in(used(C1,C2,T)), out(chop(C1)), out(chop(C2)))) ))
```

Timed Dining Philosophers: `table` ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (     % (1)
    in(chops(C1,C2)) )).
reaction( out(chops(C1,C2)), (operation, completion), (     % (1')
    in(used(C1,C2,_)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (      % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (      % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                 % (4)
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                         % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                         % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)) )).
reaction( in(chops(C1,C2)), (operation, completion), (      % (6)
    current_time(T), rd(max eating time(Max)), T1 is T + Max,
    out(used(C1,C2,T)),
    out_s(time(T1),(in(used(C1,C2,T)), out(chop(C1)), out(chop(C2)))) ))
```

Timed Dining Philosophers in ReSpecT: Results

Results

protocol no deadlock

protocol fairness

protocol trivial philosopher's interaction protocol

tuple centre shared resources handled properly

tuple centre no starvation

Timed Dining Philosophers in ReSpecT: Results

Results

protocol no deadlock

protocol fairness

protocol trivial philosopher's interaction protocol

tuple centre shared resources handled properly

tuple centre no starvation

Timed Dining Philosophers in ReSpecT: Results

Results

protocol	no deadlock
protocol	fairness
protocol	trivial philosopher's interaction protocol
tuple centre	shared resources handled properly
tuple centre	no starvation

Timed Dining Philosophers in ReSpecT: Results

Results

- protocol no deadlock
- protocol fairness
- protocol trivial philosopher's interaction protocol
- tuple centre shared resources handled properly
- tuple centre no starvation

Timed Dining Philosophers in ReSpecT: Results

Results

protocol	no deadlock
protocol	fairness
protocol	trivial philosopher's interaction protocol
tuple centre	shared resources handled properly
tuple centre	no starvation

Timed Dining Philosophers in ReSpecT: Results

Results

protocol	no deadlock
protocol	fairness
protocol	trivial philosopher's interaction protocol
tuple centre	shared resources handled properly
tuple centre	no starvation

Outline

ReSpecT Basic Syntax for Reactions

Logic Tuples

- ReSpecT tuple centres adopt logic tuples for both ordinary tuples and specification tuples
- ordinary tuples are simple first-order logic (FOL) facts, written with a Prolog syntax
  - while ordinary logic tuples are typically ground facts, there is nothing to constrain them to be such
- specification tuples are logic tuples of the form `reaction`(E, G, R)
  - if event Ev occurs in the tuple centre,
  - which matches event descriptor E such that $\theta = mgu(E, Ev)$, and
  - guard G is true,
  - then reaction $R\theta$ to Ev is triggered for execution in the tuple centre

ReSpecT Basic Syntax for Reactions

Logic Tuples

- ReSpecT tuple centres adopt logic tuples for both ordinary tuples and specification tuples
- ordinary tuples are simple first-order logic (FOL) facts, written with a Prolog syntax
  - while ordinary logic tuples are typically ground facts, there is nothing to constrain them to be such
- specification tuples are logic tuples of the form `reaction`(E, G, R)
  - if event Ev occurs in the tuple centre,
  - which matches event descriptor E such that $\theta = mgu(E, Ev)$, and
  - guard G is true,
  - then reaction $R\theta$ to Ev is triggered for execution in the tuple centre

ReSpecT Basic Syntax for Reactions

Logic Tuples

- ReSpecT tuple centres adopt logic tuples for both ordinary tuples and specification tuples
- ordinary tuples are simple first-order logic (FOL) facts, written with a Prolog syntax
  - while ordinary logic tuples are typically ground facts, there is nothing to constrain them to be such
- specification tuples are logic tuples of the form `reaction`(E, G, R)
  - if event Ev occurs in the tuple centre,
  - which matches event descriptor E such that $\theta = mgu(E, Ev)$, and
  - guard G is true,
  - then reaction $R\theta$ to Ev is triggered for execution in the tuple centre

ReSpecT Basic Syntax for Reactions

Logic Tuples

- ReSpecT tuple centres adopt logic tuples for both ordinary tuples and specification tuples
- ordinary tuples are simple first-order logic (FOL) facts, written with a Prolog syntax
  - while ordinary logic tuples are typically ground facts, there is nothing to constrain them to be such
- specification tuples are logic tuples of the form `reaction`(E, G, R)
  - if event Ev occurs in the tuple centre,
  - which matches event descriptor E such that $\theta = mgu(E, Ev)$, and
  - guard G is true,
  - then reaction $R\theta$ to Ev is triggered for execution in the tuple centre

ReSpecT Basic Syntax for Reactions

Logic Tuples

- ReSpecT tuple centres adopt logic tuples for both ordinary tuples and specification tuples
- ordinary tuples are simple first-order logic (FOL) facts, written with a Prolog syntax
  - while ordinary logic tuples are typically ground facts, there is nothing to constrain them to be such
- specification tuples are logic tuples of the form `reaction(`E`,`G`,`R`)`
  - if event Ev occurs in the tuple centre,
  - which matches event descriptor E such that $\theta = mgu(E,Ev)$, and
  - guard G is true,
  - then reaction $R\theta$ to Ev is triggered for execution in the tuple centre

ReSpecT Basic Syntax for Reactions

Logic Tuples

- ReSpecT tuple centres adopt logic tuples for both ordinary tuples and specification tuples
- ordinary tuples are simple first-order logic (FOL) facts, written with a Prolog syntax
  - while ordinary logic tuples are typically ground facts, there is nothing to constrain them to be such
- specification tuples are logic tuples of the form `reaction(`E`,`G`,`R`)`
  - if event Ev occurs in the tuple centre,
  - which matches event descriptor E such that $\theta = mgu(E,Ev)$, and
  - guard G is true,
  - then reaction $R\theta$ to Ev is triggered for execution in the tuple centre

ReSpecT Basic Syntax for Reactions

Logic Tuples

- ReSpecT tuple centres adopt logic tuples for both ordinary tuples and specification tuples
- ordinary tuples are simple first-order logic (FOL) facts, written with a Prolog syntax
  - while ordinary logic tuples are typically ground facts, there is nothing to constrain them to be such
- specification tuples are logic tuples of the form `reaction(`E`,`G`,`R`)`
  - if event Ev occurs in the tuple centre,
  - which matches event descriptor E such that $\theta = mgu(E,Ev)$, and
  - guard G is true,
  - then reaction $R\theta$ to Ev is triggered for execution in the tuple centre

ReSpecT Basic Syntax for Reactions

Logic Tuples

- ReSpecT tuple centres adopt logic tuples for both ordinary tuples and specification tuples
- ordinary tuples are simple first-order logic (FOL) facts, written with a Prolog syntax
  - while ordinary logic tuples are typically ground facts, there is nothing to constrain them to be such
- specification tuples are logic tuples of the form `reaction`(E, G, R)
  - if event Ev occurs in the tuple centre,
  - which matches event descriptor E such that $\theta = mgu(E, Ev)$, and
  - guard G is true,
  - then reaction $R\theta$ to Ev is triggered for execution in the tuple centre

ReSpecT Basic Syntax for Reactions

Logic Tuples

- ReSpecT tuple centres adopt logic tuples for both ordinary tuples and specification tuples
- ordinary tuples are simple first-order logic (FOL) facts, written with a Prolog syntax
  - while ordinary logic tuples are typically ground facts, there is nothing to constrain them to be such
- specification tuples are logic tuples of the form `reaction`(E, G, R)
  - if event Ev occurs in the tuple centre,
  - which matches event descriptor E such that $\theta = mgu(E, Ev)$, and
  - guard G is true,
  - then reaction $R\theta$ to Ev is triggered for execution in the tuple centre

ReSpecT Core Syntax

⟨*TCSpecification*⟩	::=	{⟨*SpecificationTuple*⟩ .}
⟨*SpecificationTuple*⟩	::=	`reaction(` ⟨*SimpleTCEvent*⟩ `,` [⟨*Guard*⟩ `,`] ⟨*Reaction*⟩ `)`
⟨*SimpleTCEvent*⟩	::=	⟨*SimpleTCPredicate*⟩ `(` ⟨*Tuple*⟩ `)` | `time(` ⟨*Time*⟩ `)` | ⟨*EnvPredicate*⟩
⟨*Guard*⟩	::=	⟨*GuardPredicate*⟩ | `(` ⟨*GuardPredicate*⟩ {`,` ⟨*GuardPredicate*⟩} `)`
⟨*Reaction*⟩	::=	⟨*ReactionGoal*⟩ | `(` ⟨*ReactionGoal*⟩ {`,` ⟨*ReactionGoal*⟩} `)`
⟨*ReactionGoal*⟩	::=	⟨*TCPredicate*⟩ | ⟨*ObservationPredicate*⟩ | ⟨*Computation*⟩ | `(` ⟨*ReactionGoal*⟩ `;` ⟨*ReactionGoal*⟩ `)`
⟨*TCPredicate*⟩	::=	⟨*SimpleTCPredicate*⟩ | ⟨*TCLinkPredicate*⟩ | ⟨*TCEnvPredicate*⟩
⟨*EnvPredicate*⟩	::=	`get(` ⟨*Key*⟩ `,` ⟨*Value*⟩ `)` | `set(` ⟨*Key*⟩ `,` ⟨*Value*⟩ `)`
⟨*SimpleTCPredicate*⟩	::=	⟨*TCStatePredicate*⟩ `(` ⟨*Tuple*⟩ `)` | ⟨*TCForgePredicate*⟩ `(` ⟨*SpecificationTuple*⟩ `)`
⟨*TCLinkPredicate*⟩	::=	⟨*TCIdentifier*⟩ `?` ⟨*SimpleTCPredicate*⟩
⟨*TCEnvPredicate*⟩	::=	⟨*EnvResIdentifier*⟩ `?` ⟨*EnvPredicate*⟩
⟨*TCStatePredicate*⟩	::=	`in` | `inp` | `rd` | `rdp` | `out` | `no`
⟨*TCForgePredicate*⟩	::=	⟨*TCStatePredicate*⟩`_s`
⟨*ObservationPredicate*⟩	::=	⟨*EventView*⟩`_`⟨*EventInformation*⟩ `(` ⟨*Tuple*⟩ `)` | `env(` ⟨*Key*⟩ `,` ⟨*Value*⟩ `)`
⟨*EventView*⟩	::=	`current` | `event` | `start`
⟨*EventInformation*⟩	::=	`predicate` | `tuple` | `source` | `target` | `time`
⟨*GuardPredicate*⟩	::=	`request` | `response` | `success` | `failure` | `endo` | `exo` | `intra` | `inter` | `from_agent` | `to_agent` | `from_tc` | `to_tc` | `from_env` | `to_env` | `before(` ⟨*Time*⟩ `)` | `after(` ⟨*Time*⟩ `)`
⟨*Computation*⟩	is	a Prolog-like goal performing arithmetic / logic computations
⟨*Time*⟩	is	a non-negative integer
⟨*Tuple*⟩, ⟨*Key*⟩, ⟨*Value*⟩	are	Prolog terms

ReSpecT Behaviour Specification

$$
\begin{array}{rcl}
\langle \mathit{TCSpecification} \rangle & ::= & \{ \langle \mathit{SpecificationTuple} \rangle \ . \} \\
\langle \mathit{SpecificationTuple} \rangle & ::= & \texttt{reaction(} \\
 & & \quad \langle \mathit{SimpleTCEvent} \rangle \ , \\
 & & \quad [\langle \mathit{Guard} \rangle \ ,] \\
 & & \quad \langle \mathit{Reaction} \rangle \\
 & & \texttt{)}
\end{array}
$$

- a behaviour specification $\langle \mathit{TCSpecification} \rangle$ is a logic theory of FOL tuples `reaction/3`
- a specification tuple contains an event descriptor $\langle \mathit{SimpleTCEvent} \rangle$, a guard $\langle \mathit{Guard} \rangle$ (optional), and a sequence $\langle \mathit{Reaction} \rangle$ of reaction goals
  - a `reaction/2` specification tuple implicitly defines an empty guard

ReSpecT Behaviour Specification

$$
\begin{array}{rcl}
\langle \mathit{TCSpecification} \rangle & ::= & \{ \langle \mathit{SpecificationTuple} \rangle \ . \} \\
\langle \mathit{SpecificationTuple} \rangle & ::= & \texttt{reaction(} \\
 & & \quad \langle \mathit{SimpleTCEvent} \rangle \ , \\
 & & \quad [\langle \mathit{Guard} \rangle \ ,] \\
 & & \quad \langle \mathit{Reaction} \rangle \\
 & & \texttt{)}
\end{array}
$$

- a behaviour specification $\langle \mathit{TCSpecification} \rangle$ is a logic theory of FOL tuples `reaction/3`
- a specification tuple contains an event descriptor $\langle \mathit{SimpleTCEvent} \rangle$, a guard $\langle \mathit{Guard} \rangle$ (optional), and a sequence $\langle \mathit{Reaction} \rangle$ of reaction goals
  - a `reaction/2` specification tuple implicitly defines an empty guard

ReSpecT Behaviour Specification

$$
\begin{array}{rcl}
\langle \mathit{TCSpecification} \rangle & ::= & \{ \langle \mathit{SpecificationTuple} \rangle \ . \} \\
\langle \mathit{SpecificationTuple} \rangle & ::= & \texttt{reaction(} \\
 & & \quad \langle \mathit{SimpleTCEvent} \rangle \ , \\
 & & \quad [\langle \mathit{Guard} \rangle \ ,] \\
 & & \quad \langle \mathit{Reaction} \rangle \\
 & & \texttt{)}
\end{array}
$$

- a behaviour specification $\langle \mathit{TCSpecification} \rangle$ is a logic theory of FOL tuples `reaction/3`
- a specification tuple contains an event descriptor $\langle \mathit{SimpleTCEvent} \rangle$, a guard $\langle \mathit{Guard} \rangle$ (optional), and a sequence $\langle \mathit{Reaction} \rangle$ of reaction goals
  - a `reaction/2` specification tuple implicitly defines an empty guard

ReSpecT Event Descriptor

$$\langle SimpleTCEvent \rangle \quad ::= \quad \langle SimpleTCPredicate \rangle \, (\, \langle Tuple \rangle \,) \; | \; \texttt{time}(\, \langle Time \rangle \,) \; | \; \langle EnvPredicate \rangle$$

- an event descriptor $\langle SimpleTCEvent \rangle$ is either the invocation of a primitive $\langle SimpleTCPredicate \rangle \, (\, \langle Tuple \rangle \,)$, a time event $\texttt{time}(\, \langle Time \rangle \,)$, or an environment event in terms of an $\langle EnvPredicate \rangle$
- an event descriptor $\langle SimpleTCEvent \rangle$ is used to match with with *admissible A&A events*

ReSpecT Event Descriptor

$$\langle SimpleTCEvent \rangle \quad ::= \quad \langle SimpleTCPredicate \rangle \, (\, \langle Tuple \rangle \,) \mid \texttt{time}(\, \langle Time \rangle \,) \mid \langle EnvPredicate \rangle$$

- an event descriptor $\langle SimpleTCEvent \rangle$ is either the invocation of a primitive $\langle SimpleTCPredicate \rangle \, (\, \langle Tuple \rangle \,)$, a time event $\texttt{time}(\, \langle Time \rangle \,)$, or an environment event in terms of an $\langle EnvPredicate \rangle$
- an event descriptor $\langle SimpleTCEvent \rangle$ is used to match with with *admissible A&A events*

ReSpecT Admissible Event

$$
\begin{array}{rcl}
\langle \mathit{GeneralTCEvent} \rangle & ::= & \langle \mathit{StartCause} \rangle \, , \langle \mathit{Cause} \rangle \, , \langle \mathit{TCCycleResult} \rangle \\
\langle \mathit{StartCause} \rangle \, , \langle \mathit{Cause} \rangle & ::= & \langle \mathit{SimpleTCEvent} \rangle \, , \langle \mathit{Source} \rangle \, , \langle \mathit{Target} \rangle \, , \langle \mathit{Time} \rangle \\
\langle \mathit{Source} \rangle \, , \langle \mathit{Target} \rangle & ::= & \langle \mathit{AgentIdentifier} \rangle \mid \langle \mathit{TCIdentifier} \rangle \mid \langle \mathit{EnvResIdentifier} \rangle \\
\langle \mathit{AgentIdentifier} \rangle & ::= & \langle \mathit{AgentName} \rangle \, @ \, \langle \mathit{NetworkLocation} \rangle \\
\langle \mathit{TCIdentifier} \rangle & ::= & \langle \mathit{TCName} \rangle \, @ \, \langle \mathit{NetworkLocation} \rangle \\
\langle \mathit{EnvResIdentifier} \rangle & ::= & \langle \mathit{EnvResName} \rangle \, @ \, \langle \mathit{NetworkLocation} \rangle \\
\langle \mathit{AgentName} \rangle \, , \langle \mathit{TCName} \rangle \, , \langle \mathit{EnvResName} \rangle & \text{are} & \text{Prolog ground terms} \\
\langle \mathit{NetworkLocation} \rangle & \text{is} & \text{a Prolog string representing either an IP name or a DNS entry} \\
\langle \mathit{Time} \rangle & \text{is} & \text{a non-negative integer} \\
\langle \mathit{TCCycleResult} \rangle & ::= & \bot \mid \{ \langle \mathit{Tuple} \rangle \} \\
\langle \mathit{Tuple} \rangle & \text{is} & \text{a Prolog term}
\end{array}
$$

- an admissible A&A event descriptor includes its prime cause, its immediate cause, and the result of the tuple centre response
  - prime cause and immediate cause may coincide—such as when an agent invocation reaches its target tuple centre
  - or, they might be different—such as when a link primitive is invoked by a tuple centre reacting to an agent primitive invocation upon another tuple centre
- a reaction specification tuple `reaction`(E, G, R) and an admissible A&A event ϵ match if E unifies with $\epsilon . \langle \mathit{Cause} \rangle . \langle \mathit{SimpleTCEvent} \rangle$
- the result is undefined in the invocation stage, whereas it is defined in the completion stage

ReSpecT Admissible Event

$$
\begin{array}{rcl}
\langle \mathit{GeneralTCEvent} \rangle & ::= & \langle \mathit{StartCause} \rangle \, , \langle \mathit{Cause} \rangle \, , \langle \mathit{TCCycleResult} \rangle \\
\langle \mathit{StartCause} \rangle \, , \langle \mathit{Cause} \rangle & ::= & \langle \mathit{SimpleTCEvent} \rangle \, , \langle \mathit{Source} \rangle \, , \langle \mathit{Target} \rangle \, , \langle \mathit{Time} \rangle \\
\langle \mathit{Source} \rangle \, , \langle \mathit{Target} \rangle & ::= & \langle \mathit{AgentIdentifier} \rangle \mid \langle \mathit{TCIdentifier} \rangle \mid \langle \mathit{EnvResIdentifier} \rangle \\
\langle \mathit{AgentIdentifier} \rangle & ::= & \langle \mathit{AgentName} \rangle \, @ \, \langle \mathit{NetworkLocation} \rangle \\
\langle \mathit{TCIdentifier} \rangle & ::= & \langle \mathit{TCName} \rangle \, @ \, \langle \mathit{NetworkLocation} \rangle \\
\langle \mathit{EnvResIdentifier} \rangle & ::= & \langle \mathit{EnvResName} \rangle \, @ \, \langle \mathit{NetworkLocation} \rangle \\
\langle \mathit{AgentName} \rangle \, , \langle \mathit{TCName} \rangle \, , \langle \mathit{EnvResName} \rangle & \text{are} & \text{Prolog ground terms} \\
\langle \mathit{NetworkLocation} \rangle & \text{is} & \text{a Prolog string representing either an IP name or a DNS entry} \\
\langle \mathit{Time} \rangle & \text{is} & \text{a non-negative integer} \\
\langle \mathit{TCCycleResult} \rangle & ::= & \bot \mid \{ \langle \mathit{Tuple} \rangle \} \\
\langle \mathit{Tuple} \rangle & \text{is} & \text{a Prolog term}
\end{array}
$$

- an admissible A&A event descriptor includes its prime cause, its immediate cause, and the result of the tuple centre response
  - prime cause and immediate cause may coincide—such as when an agent invocation reaches its target tuple centre
  - or, they might be different—such as when a link primitive is invoked by a tuple centre reacting to an agent primitive invocation upon another tuple centre
- a reaction specification tuple $\texttt{reaction}(E, G, R)$ and an admissible A&A event ϵ match if E unifies with $\epsilon . \langle \mathit{Cause} \rangle . \langle \mathit{SimpleTCEvent} \rangle$
- the result is undefined in the invocation stage, whereas it is defined in the completion stage

ReSpecT Admissible Event

$$
\begin{array}{rcl}
\langle \mathit{GeneralTCEvent} \rangle & ::= & \langle \mathit{StartCause} \rangle \, , \langle \mathit{Cause} \rangle \, , \langle \mathit{TCCycleResult} \rangle \\
\langle \mathit{StartCause} \rangle \, , \langle \mathit{Cause} \rangle & ::= & \langle \mathit{SimpleTCEvent} \rangle \, , \langle \mathit{Source} \rangle \, , \langle \mathit{Target} \rangle \, , \langle \mathit{Time} \rangle \\
\langle \mathit{Source} \rangle \, , \langle \mathit{Target} \rangle & ::= & \langle \mathit{AgentIdentifier} \rangle \mid \langle \mathit{TCIdentifier} \rangle \mid \langle \mathit{EnvResIdentifier} \rangle \\
\langle \mathit{AgentIdentifier} \rangle & ::= & \langle \mathit{AgentName} \rangle \, @ \, \langle \mathit{NetworkLocation} \rangle \\
\langle \mathit{TCIdentifier} \rangle & ::= & \langle \mathit{TCName} \rangle \, @ \, \langle \mathit{NetworkLocation} \rangle \\
\langle \mathit{EnvResIdentifier} \rangle & ::= & \langle \mathit{EnvResName} \rangle \, @ \, \langle \mathit{NetworkLocation} \rangle \\
\langle \mathit{AgentName} \rangle \, , \langle \mathit{TCName} \rangle \, , \langle \mathit{EnvResName} \rangle & \text{are} & \text{Prolog ground terms} \\
\langle \mathit{NetworkLocation} \rangle & \text{is} & \text{a Prolog string representing either an IP name or a DNS entry} \\
\langle \mathit{Time} \rangle & \text{is} & \text{a non-negative integer} \\
\langle \mathit{TCCycleResult} \rangle & ::= & \bot \mid \{ \langle \mathit{Tuple} \rangle \} \\
\langle \mathit{Tuple} \rangle & \text{is} & \text{a Prolog term}
\end{array}
$$

- an admissible A&A event descriptor includes its prime cause, its immediate cause, and the result of the tuple centre response
  - prime cause and immediate cause may coincide—such as when an agent invocation reaches its target tuple centre
  - or, they might be different—such as when a link primitive is invoked by a tuple centre reacting to an agent primitive invocation upon another tuple centre
- a reaction specification tuple $\texttt{reaction}(E, G, R)$ and an admissible A&A event ϵ match if E unifies with $\epsilon . \langle \mathit{Cause} \rangle . \langle \mathit{SimpleTCEvent} \rangle$
- the result is undefined in the invocation stage, whereas it is defined in the completion stage

ReSpecT Admissible Event

$$
\begin{array}{rcl}
\langle \mathit{GeneralTCEvent} \rangle & ::= & \langle \mathit{StartCause} \rangle \, , \langle \mathit{Cause} \rangle \, , \langle \mathit{TCCycleResult} \rangle \\
\langle \mathit{StartCause} \rangle \, , \langle \mathit{Cause} \rangle & ::= & \langle \mathit{SimpleTCEvent} \rangle \, , \langle \mathit{Source} \rangle \, , \langle \mathit{Target} \rangle \, , \langle \mathit{Time} \rangle \\
\langle \mathit{Source} \rangle \, , \langle \mathit{Target} \rangle & ::= & \langle \mathit{AgentIdentifier} \rangle \mid \langle \mathit{TCIdentifier} \rangle \mid \langle \mathit{EnvResIdentifier} \rangle \\
\langle \mathit{AgentIdentifier} \rangle & ::= & \langle \mathit{AgentName} \rangle \, @ \, \langle \mathit{NetworkLocation} \rangle \\
\langle \mathit{TCIdentifier} \rangle & ::= & \langle \mathit{TCName} \rangle \, @ \, \langle \mathit{NetworkLocation} \rangle \\
\langle \mathit{EnvResIdentifier} \rangle & ::= & \langle \mathit{EnvResName} \rangle \, @ \, \langle \mathit{NetworkLocation} \rangle \\
\langle \mathit{AgentName} \rangle \, , \langle \mathit{TCName} \rangle \, , \langle \mathit{EnvResName} \rangle & \text{are} & \text{Prolog ground terms} \\
\langle \mathit{NetworkLocation} \rangle & \text{is} & \text{a Prolog string representing either an IP name or a DNS entry} \\
\langle \mathit{Time} \rangle & \text{is} & \text{a non-negative integer} \\
\langle \mathit{TCCycleResult} \rangle & ::= & \bot \mid \{ \langle \mathit{Tuple} \rangle \} \\
\langle \mathit{Tuple} \rangle & \text{is} & \text{a Prolog term}
\end{array}
$$

- an admissible A&A event descriptor includes its prime cause, its immediate cause, and the result of the tuple centre response
  - prime cause and immediate cause may coincide—such as when an agent invocation reaches its target tuple centre
  - or, they might be different—such as when a link primitive is invoked by a tuple centre reacting to an agent primitive invocation upon another tuple centre
- a reaction specification tuple `reaction(`E`,`G`,`R`)` and an admissible A&A event ϵ match if E unifies with $\epsilon . \langle \mathit{Cause} \rangle . \langle \mathit{SimpleTCEvent} \rangle$
- the result is undefined in the invocation stage, whereas it is defined in the completion stage

ReSpecT Admissible Event

$$
\begin{array}{rcl}
\langle \mathit{GeneralTCEvent} \rangle & ::= & \langle \mathit{StartCause} \rangle \, , \langle \mathit{Cause} \rangle \, , \langle \mathit{TCCycleResult} \rangle \\
\langle \mathit{StartCause} \rangle \, , \langle \mathit{Cause} \rangle & ::= & \langle \mathit{SimpleTCEvent} \rangle \, , \langle \mathit{Source} \rangle \, , \langle \mathit{Target} \rangle \, , \langle \mathit{Time} \rangle \\
\langle \mathit{Source} \rangle \, , \langle \mathit{Target} \rangle & ::= & \langle \mathit{AgentIdentifier} \rangle \mid \langle \mathit{TCIdentifier} \rangle \mid \langle \mathit{EnvResIdentifier} \rangle \\
\langle \mathit{AgentIdentifier} \rangle & ::= & \langle \mathit{AgentName} \rangle \, @ \, \langle \mathit{NetworkLocation} \rangle \\
\langle \mathit{TCIdentifier} \rangle & ::= & \langle \mathit{TCName} \rangle \, @ \, \langle \mathit{NetworkLocation} \rangle \\
\langle \mathit{EnvResIdentifier} \rangle & ::= & \langle \mathit{EnvResName} \rangle \, @ \, \langle \mathit{NetworkLocation} \rangle \\
\langle \mathit{AgentName} \rangle \, , \langle \mathit{TCName} \rangle \, , \langle \mathit{EnvResName} \rangle & \text{are} & \text{Prolog ground terms} \\
\langle \mathit{NetworkLocation} \rangle & \text{is} & \text{a Prolog string representing either an IP name or a DNS entry} \\
\langle \mathit{Time} \rangle & \text{is} & \text{a non-negative integer} \\
\langle \mathit{TCCycleResult} \rangle & ::= & \bot \mid \{ \langle \mathit{Tuple} \rangle \} \\
\langle \mathit{Tuple} \rangle & \text{is} & \text{a Prolog term}
\end{array}
$$

- an admissible A&A event descriptor includes its prime cause, its immediate cause, and the result of the tuple centre response
  - prime cause and immediate cause may coincide—such as when an agent invocation reaches its target tuple centre
  - or, they might be different—such as when a link primitive is invoked by a tuple centre reacting to an agent primitive invocation upon another tuple centre
- a reaction specification tuple `reaction(`E`,`G`,`R`)` and an admissible A&A event ϵ match if E unifies with $\epsilon . \langle \mathit{Cause} \rangle . \langle \mathit{SimpleTCEvent} \rangle$
- the result is undefined in the invocation stage, whereas it is defined in the completion stage

ReSpecT Guards

$$
\begin{array}{rcl}
\langle \mathit{Guard} \rangle & ::= & \langle \mathit{GuardPredicate} \rangle \mid \\
 & & (\, \langle \mathit{GuardPredicate} \rangle \, \{ \, , \langle \mathit{GuardPredicate} \rangle \} \,) \\
\langle \mathit{GuardPredicate} \rangle & ::= & \texttt{request} \mid \texttt{response} \mid \texttt{success} \mid \texttt{failure} \mid \\
 & & \texttt{endo} \mid \texttt{exo} \mid \texttt{intra} \mid \texttt{inter} \mid \\
 & & \texttt{from\_agent} \mid \texttt{to\_agent} \mid \texttt{from\_tc} \mid \texttt{to\_tc} \mid \\
 & & \texttt{from\_env} \mid \texttt{to\_env} \mid \\
 & & \texttt{before}(\, \langle \mathit{Time} \rangle \,) \mid \texttt{after}(\, \langle \mathit{Time} \rangle \,) \\
\langle \mathit{Time} \rangle & \text{is} & \text{a non-negative integer}
\end{array}
$$

- A triggered reaction is actually executed only if its guard is true
- All guard predicates are ground ones, so their have always a success / failure semantics
- Guard predicates concern properties of the event, so they can be used to further select some classes of events after the initial matching between the admissible event and the event descriptor

ReSpecT Guards

$$\begin{array}{rcl} \langle \mathit{Guard} \rangle & ::= & \langle \mathit{GuardPredicate} \rangle \mid \\ & & (\, \langle \mathit{GuardPredicate} \rangle \, \{ \, , \langle \mathit{GuardPredicate} \rangle \} \,) \\ \langle \mathit{GuardPredicate} \rangle & ::= & \texttt{request} \mid \texttt{response} \mid \texttt{success} \mid \texttt{failure} \mid \\ & & \texttt{endo} \mid \texttt{exo} \mid \texttt{intra} \mid \texttt{inter} \mid \\ & & \texttt{from\_agent} \mid \texttt{to\_agent} \mid \texttt{from\_tc} \mid \texttt{to\_tc} \mid \\ & & \texttt{from\_env} \mid \texttt{to\_env} \mid \\ & & \texttt{before}(\, \langle \mathit{Time} \rangle \,) \mid \texttt{after}(\, \langle \mathit{Time} \rangle \,) \\ \langle \mathit{Time} \rangle & \text{is} & \text{a non-negative integer} \end{array}$$

- A triggered reaction is actually executed only if its guard is true
- All guard predicates are ground ones, so their have always a success / failure semantics
- Guard predicates concern properties of the event, so they can be used to further select some classes of events after the initial matching between the admissible event and the event descriptor

ReSpecT Guards

$$\langle Guard \rangle ::= \langle GuardPredicate \rangle \mid (\langle GuardPredicate \rangle \{, \langle GuardPredicate \rangle\})$$

$$\langle GuardPredicate \rangle ::= \texttt{request} \mid \texttt{response} \mid \texttt{success} \mid \texttt{failure} \mid \texttt{endo} \mid \texttt{exo} \mid \texttt{intra} \mid \texttt{inter} \mid \texttt{from\_agent} \mid \texttt{to\_agent} \mid \texttt{from\_tc} \mid \texttt{to\_tc} \mid \texttt{from\_env} \mid \texttt{to\_env} \mid \texttt{before}(\langle Time \rangle) \mid \texttt{after}(\langle Time \rangle)$$

$$\langle Time \rangle \text{ is } \text{a non-negative integer}$$

- A triggered reaction is actually executed only if its guard is true
- All guard predicates are ground ones, so their have always a success / failure semantics
- Guard predicates concern properties of the event, so they can be used to further select some classes of events after the initial matching between the admissible event and the event descriptor

Semantics of Guard Predicates in ReSpecT

Guard atom	True if
$Guard(\epsilon, (g, G))$	$Guard(\epsilon, g) \wedge Guard(\epsilon, G)$
$Guard(\epsilon, \texttt{endo})$	$\epsilon.Cause.Source = c$
$Guard(\epsilon, \texttt{exo})$	$\epsilon.Cause.Source \neq c$
$Guard(\epsilon, \texttt{intra})$	$\epsilon.Cause.Target = c$
$Guard(\epsilon, \texttt{inter})$	$\epsilon.Cause.Target \neq c$
$Guard(\epsilon, \texttt{from\_agent})$	$\epsilon.Cause.Source$ *is an agent*
$Guard(\epsilon, \texttt{to\_agent})$	$\epsilon.Cause.Target$ *is an agent*
$Guard(\epsilon, \texttt{from\_tc})$	$\epsilon.Cause.Source$ *is a tuple centre*
$Guard(\epsilon, \texttt{to\_tc})$	$\epsilon.Cause.Target$ *is a tuple centre*
$Guard(\epsilon, \texttt{from\_env})$	$\epsilon.Cause.Source$ *is the environment*
$Guard(\epsilon, \texttt{to\_env})$	$\epsilon.Cause.Target$ *is the environment*
$Guard(\epsilon, \texttt{before}(t))$	$\epsilon.Cause.Time < t$
$Guard(\epsilon, \texttt{after}(t))$	$\epsilon.Cause.Time > t$
$Guard(\epsilon, \texttt{request})$	$\epsilon.TCCycleResult$ *is undefined*
$Guard(\epsilon, \texttt{response})$	$\epsilon.TCCycleResult$ *is defined*
$Guard(\epsilon, \texttt{success})$	$\epsilon.TCCycleResult \neq \perp$
$Guard(\epsilon, \texttt{failure})$	$\epsilon.TCCycleResult = \perp$

⟨*GuardPredicate*⟩ aliases

```
request invocation, inv, req, pre
response completion, compl, resp, post
before(Time),after(Time') between(Time,Time')
from_agent,to_tc operation
from_tc,to_tc,endo,inter link_out
from_tc,to_tc,exo,intra link_in
from_tc,to_tc,endo,intra internal
```

⟨*GuardPredicate*⟩ aliases

```
request invocation, inv, req, pre
response completion, compl, resp, post
before(Time),after(Time') between(Time,Time')
from_agent,to_tc operation
from_tc,to_tc,endo,inter link_out
from_tc,to_tc,exo,intra link_in
from_tc,to_tc,endo,intra internal
```

⟨*GuardPredicate*⟩ aliases

```
   request invocation, inv, req, pre
  response completion, compl, resp, post
before(Time),after(Time') between(Time,Time')
from_agent,to_tc operation
from_tc,to_tc,endo,inter link_out
from_tc,to_tc,exo,intra link_in
from_tc,to_tc,endo,intra internal
```

⟨*GuardPredicate*⟩ aliases

```
   request invocation, inv, req, pre
  response completion, compl, resp, post
before(Time),after(Time') between(Time,Time')
from_agent,to_tc operation
from_tc,to_tc,endo,inter link_out
from_tc,to_tc,exo,intra link_in
from_tc,to_tc,endo,intra internal
```

⟨*GuardPredicate*⟩ aliases

```
   request invocation, inv, req, pre
  response completion, compl, resp, post
before(Time),after(Time') between(Time,Time')
from_agent,to_tc operation
from_tc,to_tc,endo,inter link_out
from_tc,to_tc,exo,intra link_in
from_tc,to_tc,endo,intra internal
```

⟨*GuardPredicate*⟩ aliases

```
   request invocation, inv, req, pre
  response completion, compl, resp, post
before(Time),after(Time') between(Time,Time')
from_agent,to_tc operation
from_tc,to_tc,endo,inter link_out
from_tc,to_tc,exo,intra link_in
from_tc,to_tc,endo,intra internal
```

⟨*GuardPredicate*⟩ aliases

```
   request invocation, inv, req, pre
  response completion, compl, resp, post
before(Time),after(Time') between(Time,Time')
from_agent,to_tc operation
from_tc,to_tc,endo,inter link_out
from_tc,to_tc,exo,intra link_in
from_tc,to_tc,endo,intra internal
```

ReSpecT Reactions

$$
\begin{array}{rcl}
\langle Reaction \rangle & ::= & \langle ReactionGoal \rangle \mid \\
 & & (\langle ReactionGoal \rangle \{ , \langle ReactionGoal \rangle \}) \\
\langle ReactionGoal \rangle & ::= & \langle TCPredicate \rangle \mid \\
 & & \langle ObservationPredicate \rangle \mid \\
 & & \langle Computation \rangle \mid \\
 & & (\langle ReactionGoal \rangle ; \langle ReactionGoal \rangle) \\
\langle TCPredicate \rangle & ::= & \langle SimpleTCPredicate \rangle \mid \langle TCLinkPredicate \rangle \mid \\
 & & \langle TCEnvPredicate \rangle \\
\langle TCLinkPredicate \rangle & ::= & \langle TCIdentifier \rangle ? \langle SimpleTCPredicate \rangle
\end{array}
$$

- A reaction goal is either a primitive invocation (possibly, a link), a predicate recovering properties of the event, or some logic-based computation
- Sequences of reaction goals are executed transactionally with an overall success / failure semantics

ReSpecT Reactions

$$
\begin{aligned}
\langle Reaction \rangle &::= \langle ReactionGoal \rangle \mid \\
&\quad (\, \langle ReactionGoal \rangle \, \{ \, , \langle ReactionGoal \rangle \} \,) \\
\langle ReactionGoal \rangle &::= \langle TCPredicate \rangle \mid \\
&\quad \langle ObservationPredicate \rangle \mid \\
&\quad \langle Computation \rangle \mid \\
&\quad (\, \langle ReactionGoal \rangle \, ; \langle ReactionGoal \rangle \,) \\
\langle TCPredicate \rangle &::= \langle SimpleTCPredicate \rangle \mid \langle TCLinkPredicate \rangle \mid \\
&\quad \langle TCEnvPredicate \rangle \\
\langle TCLinkPredicate \rangle &::= \langle TCIdentifier \rangle \, ? \, \langle SimpleTCPredicate \rangle
\end{aligned}
$$

- A reaction goal is either a primitive invocation (possibly, a link), a predicate recovering properties of the event, or some logic-based computation
- Sequences of reaction goals are executed transactionally with an overall success / failure semantics

ReSpecT Tuple Centre Predicates

$$\begin{array}{rcl}
\langle \mathit{SimpleTCPredicate} \rangle & ::= & \langle \mathit{TCStatePredicate} \rangle \, (\, \langle \mathit{Tuple} \rangle \,) \mid \\
 & & \langle \mathit{TCForgePredicate} \rangle \, (\, \langle \mathit{SpecificationTuple} \rangle \,) \\
\langle \mathit{TCStatePredicate} \rangle & ::= & \texttt{in} \mid \texttt{inp} \mid \texttt{rd} \mid \texttt{rdp} \mid \texttt{out} \mid \texttt{no} \\
\langle \mathit{TCForgePredicate} \rangle & ::= & \langle \mathit{TCStatePredicate} \rangle \texttt{_s}
\end{array}$$

- Tuple centre predicates are uniformly used for agent invocations, internal operations, and link invocations
- The same predicates are substantially used for changing the specification state, with essentially the same semantics
  - *pred*`_s` invocations affect the specification state, and can be used within reactions, also as links
- `no` works as a test for absence

ReSpecT Tuple Centre Predicates

$$\begin{aligned}
\langle \textit{SimpleTCPredicate} \rangle &::= \langle \textit{TCStatePredicate} \rangle \texttt{(} \langle \textit{Tuple} \rangle \texttt{)} \mid \\
&\quad\ \langle \textit{TCForgePredicate} \rangle \texttt{(} \langle \textit{SpecificationTuple} \rangle \texttt{)} \\
\langle \textit{TCStatePredicate} \rangle &::= \texttt{in} \mid \texttt{inp} \mid \texttt{rd} \mid \texttt{rdp} \mid \texttt{out} \mid \texttt{no} \\
\langle \textit{TCForgePredicate} \rangle &::= \langle \textit{TCStatePredicate} \rangle \texttt{_s}
\end{aligned}$$

- Tuple centre predicates are uniformly used for agent invocations, internal operations, and link invocations
- The same predicates are substantially used for changing the specification state, with essentially the same semantics
  - *pred*`_s` invocations affect the specification state, and can be used within reactions, also as links
- `no` works as a test for absence

ReSpecT Tuple Centre Predicates

$$\begin{aligned}
\langle \mathit{SimpleTCPredicate} \rangle &::= \langle \mathit{TCStatePredicate} \rangle\ (\ \langle \mathit{Tuple} \rangle\ )\ \mid \\
&\quad\ \langle \mathit{TCForgePredicate} \rangle\ (\ \langle \mathit{SpecificationTuple} \rangle\ ) \\
\langle \mathit{TCStatePredicate} \rangle &::= \texttt{in} \mid \texttt{inp} \mid \texttt{rd} \mid \texttt{rdp} \mid \texttt{out} \mid \texttt{no} \\
\langle \mathit{TCForgePredicate} \rangle &::= \langle \mathit{TCStatePredicate} \rangle \texttt{_s}
\end{aligned}$$

- Tuple centre predicates are uniformly used for agent invocations, internal operations, and link invocations
- The same predicates are substantially used for changing the specification state, with essentially the same semantics
  - *pred*`_s` invocations affect the specification state, and can be used within reactions, also as links
- `no` works as a test for absence

ReSpecT Tuple Centre Predicates

$$\begin{aligned}
\langle \mathit{SimpleTCPredicate} \rangle &::= \langle \mathit{TCStatePredicate} \rangle \, (\, \langle \mathit{Tuple} \rangle \,) \mid \\
&\quad \langle \mathit{TCForgePredicate} \rangle \, (\, \langle \mathit{SpecificationTuple} \rangle \,) \\
\langle \mathit{TCStatePredicate} \rangle &::= \texttt{in} \mid \texttt{inp} \mid \texttt{rd} \mid \texttt{rdp} \mid \texttt{out} \mid \texttt{no} \\
\langle \mathit{TCForgePredicate} \rangle &::= \langle \mathit{TCStatePredicate} \rangle \texttt{_s}
\end{aligned}$$

- Tuple centre predicates are uniformly used for agent invocations, internal operations, and link invocations
- The same predicates are substantially used for changing the specification state, with essentially the same semantics
  - pred`_s` invocations affect the specification state, and can be used within reactions, also as links
- `no` works as a test for absence

ReSpecT Observation Predicates

$$\begin{aligned}
\langle \textit{ObservationPredicate} \rangle &::= \langle \textit{EventView} \rangle _ \langle \textit{EventInformation} \rangle \; (\; \langle \textit{Tuple} \rangle \;) \\
\langle \textit{EventView} \rangle &::= \texttt{current} \mid \texttt{event} \mid \texttt{start} \\
\langle \textit{EventInformation} \rangle &::= \texttt{predicate} \mid \texttt{tuple} \mid \\
&\quad\; \texttt{source} \mid \texttt{target} \mid \texttt{time}
\end{aligned}$$

- `event` & `start` clearly refer to immediate and prime cause, respectively—`current` refers to what is currently happening, whenever this means something useful
- ⟨*EventInformation*⟩ aliases

```
predicate pred, call; deprecated: operation, op
    tuple arg
    source from
    target to
```

ReSpecT Observation Predicates

$$\begin{aligned}
\langle \textit{ObservationPredicate} \rangle &::= \langle \textit{EventView} \rangle _ \langle \textit{EventInformation} \rangle \; (\; \langle \textit{Tuple} \rangle \;) \\
\langle \textit{EventView} \rangle &::= \texttt{current} \mid \texttt{event} \mid \texttt{start} \\
\langle \textit{EventInformation} \rangle &::= \texttt{predicate} \mid \texttt{tuple} \mid \\
&\qquad \texttt{source} \mid \texttt{target} \mid \texttt{time}
\end{aligned}$$

- `event` & `start` clearly refer to immediate and prime cause, respectively—`current` refers to what is currently happening, whenever this means something useful
- ⟨*EventInformation*⟩ aliases

```
predicate pred, call; deprecated: operation, op
    tuple arg
   source from
   target to
```

ReSpecT Observation Predicates

$$\begin{aligned}
\langle \textit{ObservationPredicate} \rangle &::= \langle \textit{EventView} \rangle _ \langle \textit{EventInformation} \rangle \; (\; \langle \textit{Tuple} \rangle \;) \\
\langle \textit{EventView} \rangle &::= \texttt{current} \mid \texttt{event} \mid \texttt{start} \\
\langle \textit{EventInformation} \rangle &::= \texttt{predicate} \mid \texttt{tuple} \mid \\
&\qquad \texttt{source} \mid \texttt{target} \mid \texttt{time}
\end{aligned}$$

- `event` & `start` clearly refer to immediate and prime cause, respectively—`current` refers to what is currently happening, whenever this means something useful
- ⟨*EventInformation*⟩ aliases

  `predicate` `pred`, `call`; *deprecated*: `operation`, `op`

  `tuple` `arg`

  `source` `from`

  `target` `to`

ReSpecT Observation Predicates

$$\begin{aligned}
\langle \textit{ObservationPredicate} \rangle &::= \langle \textit{EventView} \rangle _ \langle \textit{EventInformation} \rangle \; (\; \langle \textit{Tuple} \rangle \;) \\
\langle \textit{EventView} \rangle &::= \texttt{current} \mid \texttt{event} \mid \texttt{start} \\
\langle \textit{EventInformation} \rangle &::= \texttt{predicate} \mid \texttt{tuple} \mid \\
&\qquad \texttt{source} \mid \texttt{target} \mid \texttt{time}
\end{aligned}$$

- `event` & `start` clearly refer to immediate and prime cause, respectively—`current` refers to what is currently happening, whenever this means something useful
- ⟨*EventInformation*⟩ aliases

  `predicate` `pred`, `call`; *deprecated*: `operation`, `op`

  `tuple` `arg`

  `source` `from`

  `target` `to`

ReSpecT Observation Predicates

$$\begin{aligned}
\langle \textit{ObservationPredicate} \rangle &::= \langle \textit{EventView} \rangle _ \langle \textit{EventInformation} \rangle \; (\; \langle \textit{Tuple} \rangle \;) \\
\langle \textit{EventView} \rangle &::= \texttt{current} \mid \texttt{event} \mid \texttt{start} \\
\langle \textit{EventInformation} \rangle &::= \texttt{predicate} \mid \texttt{tuple} \mid \\
&\quad\; \texttt{source} \mid \texttt{target} \mid \texttt{time}
\end{aligned}$$

- `event` & `start` clearly refer to immediate and prime cause, respectively—`current` refers to what is currently happening, whenever this means something useful
- ⟨*EventInformation*⟩ aliases

  `predicate` `pred`, `call`; *deprecated*: `operation`, `op`

  `tuple` `arg`

  `source` `from`

  `target` `to`

ReSpecT Observation Predicates

$$\begin{aligned}
\langle \textit{ObservationPredicate} \rangle &::= \langle \textit{EventView} \rangle _ \langle \textit{EventInformation} \rangle \; (\; \langle \textit{Tuple} \rangle \;) \\
\langle \textit{EventView} \rangle &::= \texttt{current} \mid \texttt{event} \mid \texttt{start} \\
\langle \textit{EventInformation} \rangle &::= \texttt{predicate} \mid \texttt{tuple} \mid \\
&\qquad \texttt{source} \mid \texttt{target} \mid \texttt{time}
\end{aligned}$$

- `event` & `start` clearly refer to immediate and prime cause, respectively—`current` refers to what is currently happening, whenever this means something useful
- $\langle \textit{EventInformation} \rangle$ aliases

```
predicate pred, call; deprecated: operation, op
    tuple arg
   source from
   target to
```

Semantics of Observation Predicates

$$\langle (r, R), Tu, \Sigma, Re, Out \rangle_\epsilon \longrightarrow_e \langle R\theta, Tu, \Sigma, Re, Out \rangle_\epsilon$$

r	where
$\texttt{env}(\texttt{K}, \texttt{V})$	$\theta = mgu((\epsilon.Key, \epsilon.Value), (\texttt{K}, \texttt{V}))$
$\texttt{event\_predicate}(\texttt{Obs})$	$\theta = mgu(\epsilon.Cause.SimpleTCEvent.SimpleTCPredicate, \texttt{Obs})$
$\texttt{event\_tuple}(\texttt{Obs})$	$\theta = mgu(\epsilon.Cause.SimpleTCEvent.Tuple, \texttt{Obs})$
$\texttt{event\_source}(\texttt{Obs})$	$\theta = mgu(\epsilon.Cause.Source, \texttt{Obs})$
$\texttt{event\_target}(\texttt{Obs})$	$\theta = mgu(\epsilon.Cause.Target, \texttt{Obs})$
$\texttt{event\_time}(\texttt{Obs})$	$\theta = mgu(\epsilon.Cause.Time, \texttt{Obs})$
$\texttt{start\_predicate}(\texttt{Obs})$	$\theta = mgu(\epsilon.StartCause.SimpleTCEvent.SimpleTCPredicate, \texttt{Obs})$
$\texttt{start\_tuple}(\texttt{Obs})$	$\theta = mgu(\epsilon.StartCause.SimpleTCEvent.Tuple, \texttt{Obs})$
$\texttt{start\_source}(\texttt{Obs})$	$\theta = mgu(\epsilon.StartCause.Source, \texttt{Obs})$
$\texttt{start\_target}(\texttt{Obs})$	$\theta = mgu(\epsilon.StartCause.Target, \texttt{Obs})$
$\texttt{start\_time}(\texttt{Obs})$	$\theta = mgu(\epsilon.StartCause.Time, \texttt{Obs})$
$\texttt{current\_predicate}(\texttt{Obs})$	$\theta = mgu(\texttt{current\_predicate}, \texttt{Obs})$
$\texttt{current\_tuple}(\texttt{Obs})$	$\theta = mgu(\texttt{Obs}, \texttt{Obs}) = \{\}$
$\texttt{current\_source}(\texttt{Obs})$	$\theta = mgu(c, \texttt{Obs})$
$\texttt{current\_target}(\texttt{Obs})$	$\theta = mgu(c, \texttt{Obs})$
$\texttt{current\_time}(\texttt{Obs})$	$\theta = mgu(nc, \texttt{Obs})$

Re-interpreting ReSpecT

- ReSpecT tuple centres as coordination artifacts
  - tuple centres as social artifacts
  - tuple centres as individual artifacts?
  - tuple centres as environment artifacts?
- ReSpecT tuple centres
  - encapsulate knowledge in terms of logic tuples
  - encapsulates behaviour in terms of ReSpecT specifications
- ReSpecT tuple centres are
  - inspectable
    - not controllable
  - malleable
  - (linkable)
  - situated
    - time
    - environment

Inspectability of ReSpecT Tuple Centres

- ReSpecT tuple centres: twofold space for tuples
  - tuple space: ordinary (logic) tuples
    - for knowledge, information, messages, communication
    - working as the (logic) *theory of communication* for MAS
  - specification space: specification (logic, ReSpecT) tuples
    - for behaviour, function, coordination
    - working as the (logic) *theory of coordination* for MAS
- Both spaces are inspectable
  - by MAS engineers, via ReSpecT inspectors
  - by agents, via `rd` & `no` primitives
    - `rd` & `no` for the tuple space; `rd_s` & `no_s` for the specification space
    - either directly or indirectly, through either a coordination primitive, or another artifact / tuple centre

Malleability of ReSpecT Tuple Centres

- The behaviour of a ReSpecT tuple centre is defined by the ReSpecT tuples in the specification space
  - it can be adapted / changed by changing its ReSpecT specification
- ReSpecT tuple centres are malleable
  - by MAS engineers, via ReSpecT tools
  - by agents, via `in` & `out` primitives
    - `in` & `out` for the tuple space; `in_s` & `out_s` for the specification space
    - either directly or indirectly, through either a coordination primitive, or another artifact / tuple centre

Linkability of ReSpecT Tuple Centres

- Every tuple centre coordination primitive is also a ReSpecT primitive for reaction goals, and a primitive for linking, too
  - all primitives are asynchronous
    - so they do not affect the transactional semantics of reactions
  - all primitives have a request / response semantics
    - including `out` / `out_s`
    - so reactions can be defined to handle both primitive invocations & completions
  - all primitives could be executed within a ReSpecT reaction
    - as either a reaction goal executed within the same tuple centre
    - or as a link primitive invoked upon another tuple centre
- ReSpecT tuple centres are linkable
  - by using tuple centre identifiers within ReSpecT reactions
    $< \mathit{TCIdentifier} > @ < \mathit{NetworkLocation} >? < \mathit{SimpleTCPredicate} >$
  - any ReSpecT reaction can invoke any coordination primitive upon any tuple centre in the network

Situatedness of ReSpecT Tuple Centres

Time [Omicini et al., 2007]

- Every tuple centre is immersed in time
  - reacting to time events
  - observing time properties of events
  - implementing timed coordination policies

Environment [Casadei and Omicini, 2009]

- Every tuple centre is immersed in the environment
  - reacting to environment events
  - observing environmental properties
  - affecting environmental properties

Outline

Situated ReSpecT

ReSpecT artifacts for environment engineering

- Artifacts are immersed into the MAS environment, and should be reactive to events of *any* sort
- Also, artifacts should mediate any agent activity toward the environment, allowing for a fruitful interaction

$\Rightarrow$ ReSpecT tuple centres should be able to *capture general environment events*, and to generally *mediate agent-environment interaction*

Situated ReSpecT: extensions

- The ReSpecT language has been revised and extended so as to *capture environment events*, and *express general MAS-environment interactions* [Casadei and Omicini, 2009]

$\Rightarrow$ ReSpecT captures, reacts to, and observes general environment events

$\Rightarrow$ ReSpecT can explicitly interact with the environment

Extending ReSpecT towards Situatedness I

Environment events

- ReSpecT tuple centres are extended to capture two classes of environmental events
  - the interaction with sensors perceiving environmental properties, through *environment predicate* `get`($\langle Key \rangle$,$\langle Value \rangle$)
  - the interaction with actuators affecting environmental properties, through *environment predicate* `set`($\langle Key \rangle$,$\langle Value \rangle$)
- Source and target of a tuple centre event can be any external resource
  - a suitable identification scheme – both at the syntax and at the infrastructure level – is introduced for environmental resources
- Properties of an environmental event can be observed through the *observation predicate* `env`($\langle Key \rangle$,$\langle Value \rangle$)

Extending ReSpecT towards Situatedness II

Environment communication

- The ReSpecT language is extended to express explicit communication with environmental resources
- The body of a ReSpecT reaction can contain a *tuple centre predicate* of the form
  - $\langle EnvResIdentifier \rangle$ `?` `get(`$\langle Key \rangle$`,`$\langle Value \rangle$`)`
    enabling a tuple centre to get properties of environmental resources
  - $\langle EnvResIdentifier \rangle$ `?` `set(`$\langle Key \rangle$`,`$\langle Value \rangle$`)`
    enabling a tuple centre to set properties of environmental resources

Extending ReSpecT towards Situatedness III

Transducers

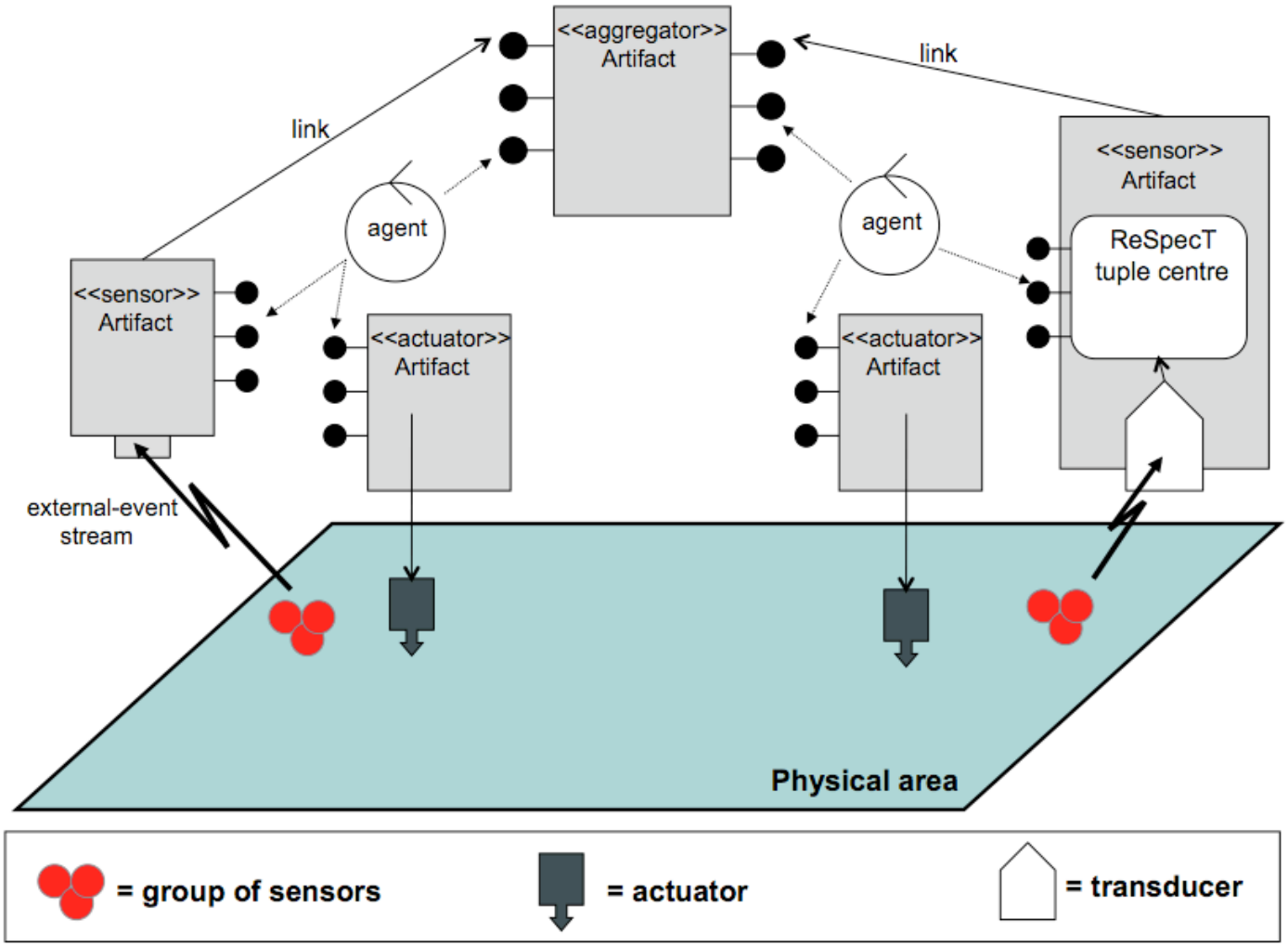
- Specific environment events have to be translated into well-formed ReSpecT tuple centre events
- This should be done at the infrastructure level, through a general-purpose schema that could be specialised according to the nature of any specific resource
- A ReSpecT *transducer* is a component able to bring environment-generated events to a ReSpecT tuple centre (and back), suitably translated according to the general ReSpecT event model
- Each transducer is specialised according to the specific portion of the environment it is in charge of handling—typically, the specific resource it is aimed at handling, like a temperature sensor, or a heater.

Outline

Controlling Environmental Properties of Physical Areas

- A set of real *sensors* are used to measure some environmental property (for instance, temperature) within an area where they are located
- Such information is then exploited to govern suitably placed *actuators* (say, heaters) that can affect the value of the observed property in the environment
- Sensors are supposed to be cheap and non-smart, but provided with some kind of communication interface – either wireless or wired – that makes it possible to send streams of sampled values of the environmental property under observation
- Accordingly, sensors are active devices, that is, devices *pro-actively* sending sensed values at a certain rate with no need of being asked for such data—this is what typically occurs in *pervasive computing* scenarios
- Altogether, actuators and sensors are part of a MAS aimed at controlling environmental properties (in the case study, temperature), which are affected by actuators based on the values measured by sensors and the designed control policies as well
- Coordination policies can be suitably automated and encapsulated within environment artifacts controlling sensors and actuators

Case Study: ReSpecT-based Architecture



Case Study: Artifact Structure

Artifacts are internally defined in terms of A&A ReSpecT tuple centres:

- `<<sensor>> artifacts` wrapping real temperature sensors which perceive temperature of different areas of the room
- `<<actuator>> artifacts` wrapping actuators, which act as heating devices so as to control temperature
- `<<aggregator>> artifact` provides an aggregated view of the temperature values perceived by sensors spread in the room since it is linked to `<<sensor>> artifacts`:
  - `<<sensor>> artifacts` update tuples on `<<aggregator>> artifact` through *linkability*

Case Study: Sensor Artifacts

```
%(1)
reaction( get(temperature, Temp), from_env, (
      event_time(Time), event_source(sensor(Id)),
      out(sensed_temperature(Id,Temp,Time)),
      tc_aggr@node_aggr ? out(sensed_temperature(Id,Temp)) )
).
%(2)
reaction( out(sensed_temperature(_,Temp,_)), from_tc, (
      in(current_temperature(_)),
      out(current_temperature(Temp)) )
).
```

Behaviour

- Reaction (1) is triggered by external events generated by a temperature sensor
- Reaction (2) updates current temperature

Case Study: Aggregator Artifacts

```
%(4)
reaction( out(sensed_temperature(Id,Temp)), from_tc, (
      in(total_temperature(OldTotalTemp),
      in(sensed_temperature(Id,OldTemp)),
      TotalTemp is OldTotalTemp - OldTemp + Temp,
      out(total_temperature(TotalTemp),
      rd(number_of_sensors(SensorNo),
      AvgTemp is TotalTemp / SensorNo,
      in(average_temp(_)), out(average_temp(AvgTemp)) )
).
```

Behaviour

- Reaction (4) keeps track of the current state of the average temperature

Case Study: Agents

Observable behaviour

Agents are goal-oriented and proactive entities that control temperature of the room

1. **get local information from sensor**
   `tc_sens@node_i ? rd(current_temperature(Temp_i))`
2. **get global information from aggregator**
   `tc_aggr@node_aggr ? rd(average_temp(AvgTemp))`
3. **deliberate action** by determining `TempVar` based on `Temp_i` and `AvgTemp`
4. **act upon actuators** (if `TempVar`$\neq 0$)
   `tc-heat_i@node_i ? out(change_temperature(TempVar))`

Case Study: Actuator Artifacts

```
%(3)
reaction( out(change_temperature(TempVar)), from_agent,
    actuator_i ? set(temp_inc,TempVar)
).
```

Behaviour

When the controller agent deliberate an increment in the temperature

- a `tc-heat_i@node_i ? out(change_temperature(TempVar))` reaches the actuator artifact
- by reaction (3), a suitable signal is sent to the actuator, through the suitably-installed transducer

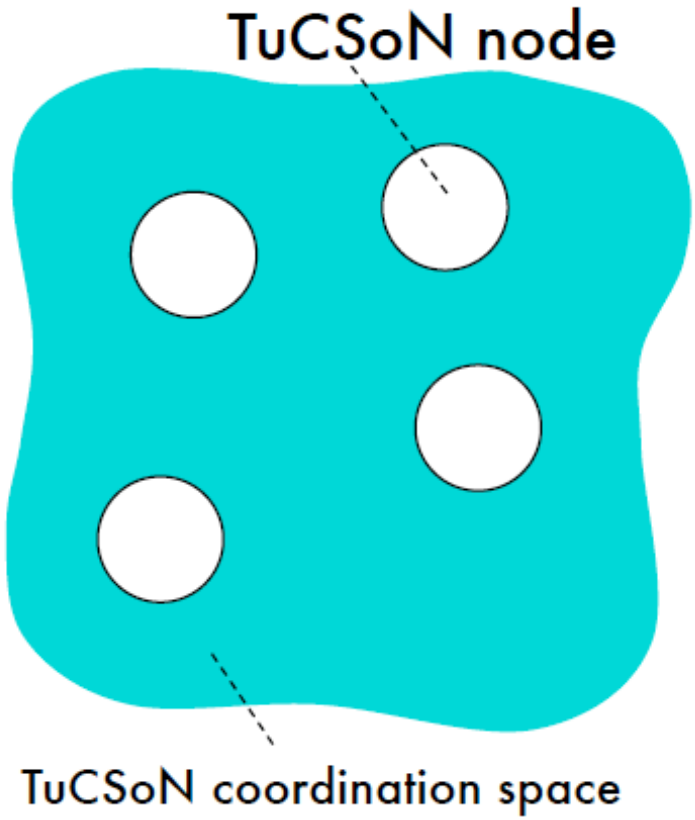
Outline

TuCSoN Coordination Model

- Tuple Centre Spread over the Network [Omicini and Zambonelli, 1999].
- Started from the idea of proposing a notion of an associative shared dataspace whose behaviour can be tailored according to the specific application needs
  - From tuple spaces to tuples centres [Omicini and Denti, 2001]: actually, ReSpecT was born here
  - Tuple centres are programmable tuple spaces
    - $\Rightarrow$ Programmabile coordination media
    - $\Rightarrow$ The coordination model it is the same as Linda
- Tuple centres are distributed over the network, collected in nodes
  - $\Rightarrow$ Distributed coordination media

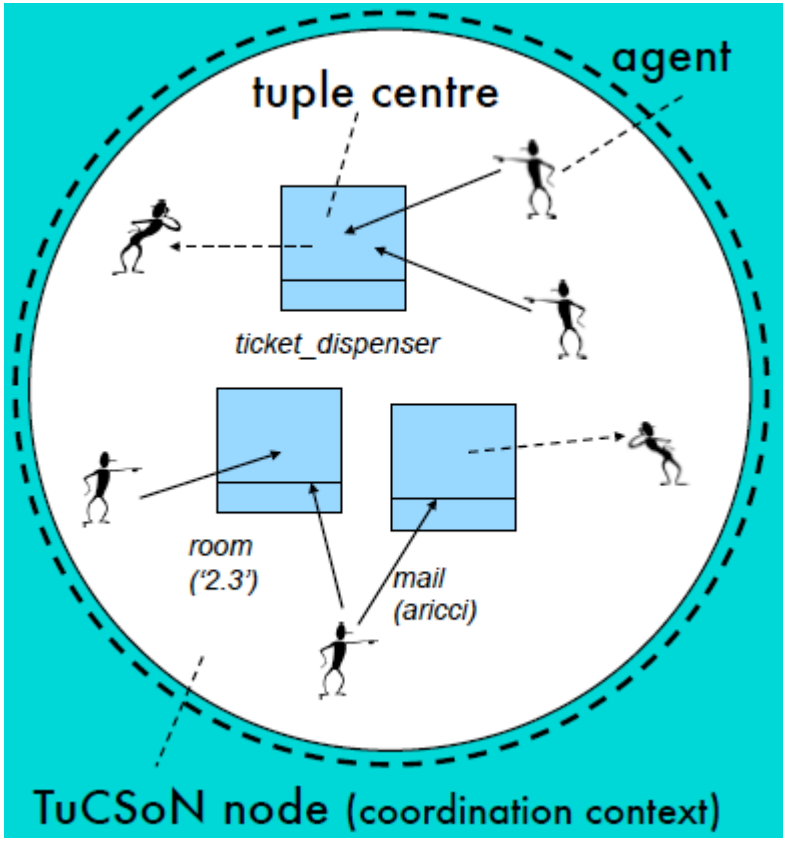
TuCSoN Coordination Space

- Set of distributed nodes
  - Each TuCSoN node is an Internet node identified by the IP (logic) address
- TuCSoN topology
  - Here, Internet topology
  - HiMAT [Cremonini et al., 1999]: hierarchical, dynamic, configurable topology



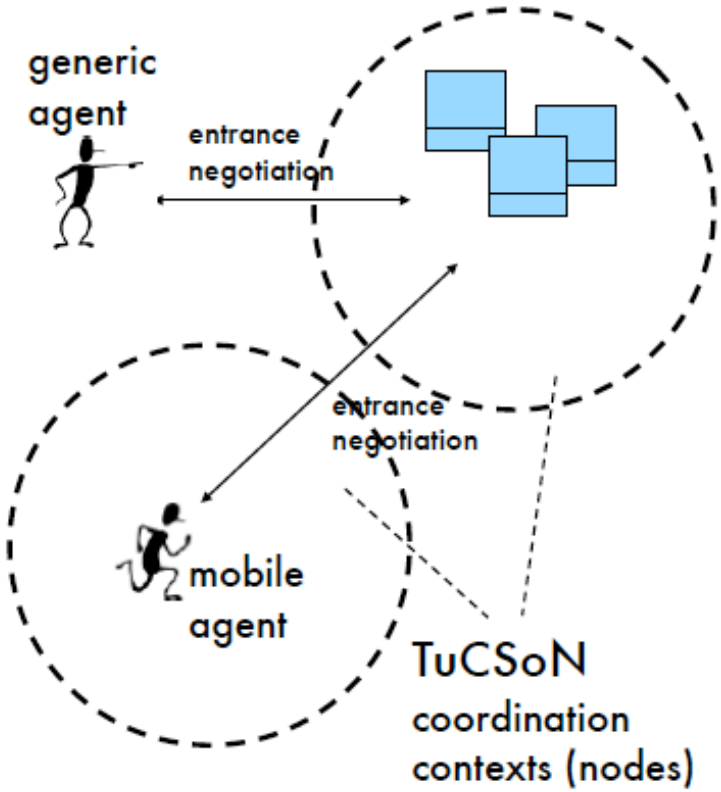
TuCSoN Node / Context I

- Each TuCSoN *node* defines a coordination context, providing an open / dynamic set of tuple centres as coordination media
  - Identified by means of a logic name (a ground FOL term).
    Ex: `mail(aricci)`, `room('2.3')`, `ticket_dispenser`, ...
  - Full tuple centre identifier: `<name>@<node>`.
    Ex: `mail(aricci)@myhome.org`, `room('2.3')@ingce.unibo.it`, `ticket_dispenser@137.204.191.188`, ...



TuCSoN Node / Context II

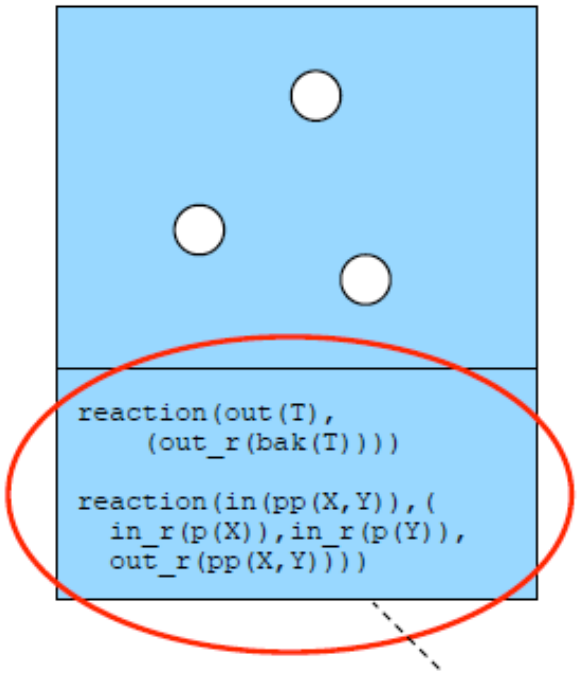
- In order to access and use the tuple centres of a node, an agent should enter the coordination context, either logically or physically (mobile agents)
- Agent Coordination Context (ACC) [Omicini, 2001].



Tuple Centres Features I

- Programmable
  - Tuple centre behaviour can be programmed to enact the desired coordination policies
  - ReSpecT is an example of programming language for specification of the behaviour
    - It programs as set of logic tuples (reactions, first order logic terms such as Prolog terms) specifying medium behaviour reacting to interaction events

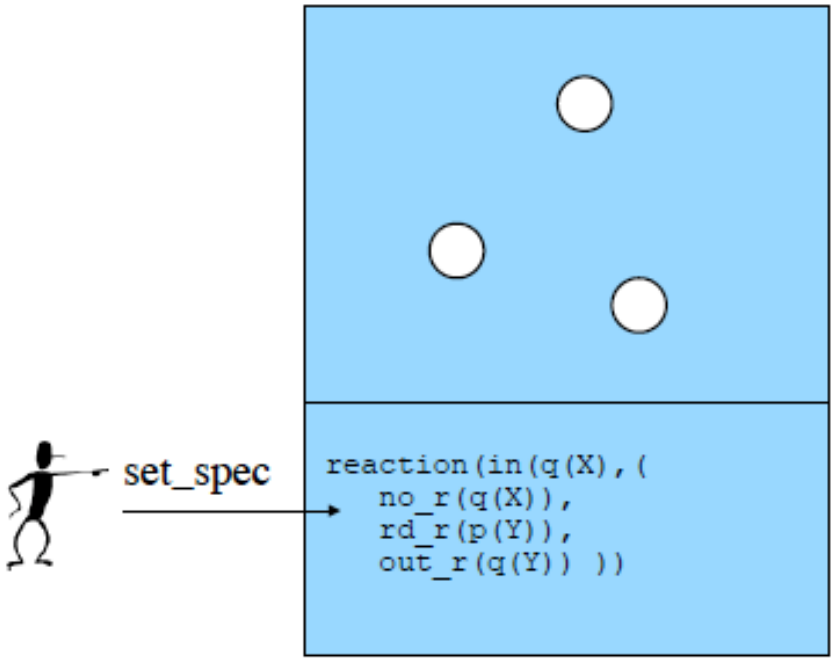
⇒ Tuple centres as a general purpose coordination media customisabled by means of a specification language like ReSpecT



specification tuples (ReSpecT language)

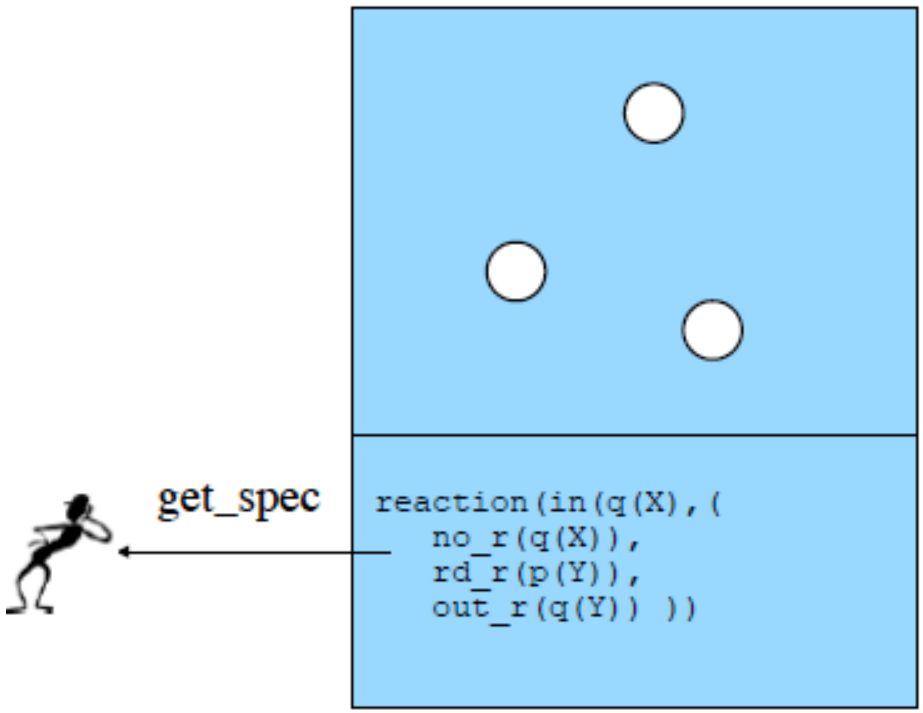
Tuple Centres Features II

- Adaptable at runtime
  - Tuple centre behaviour can be changed / adapted dynamically, at runtime, by re-programming the coordination media
- Locality / encapsulation
  - Tuple centres embed coordination laws
    - $\Rightarrow$ A tuple centre can be a full-fledged coordination abstraction
  - Reaction model ensure encapsulation of low-level coordination policies

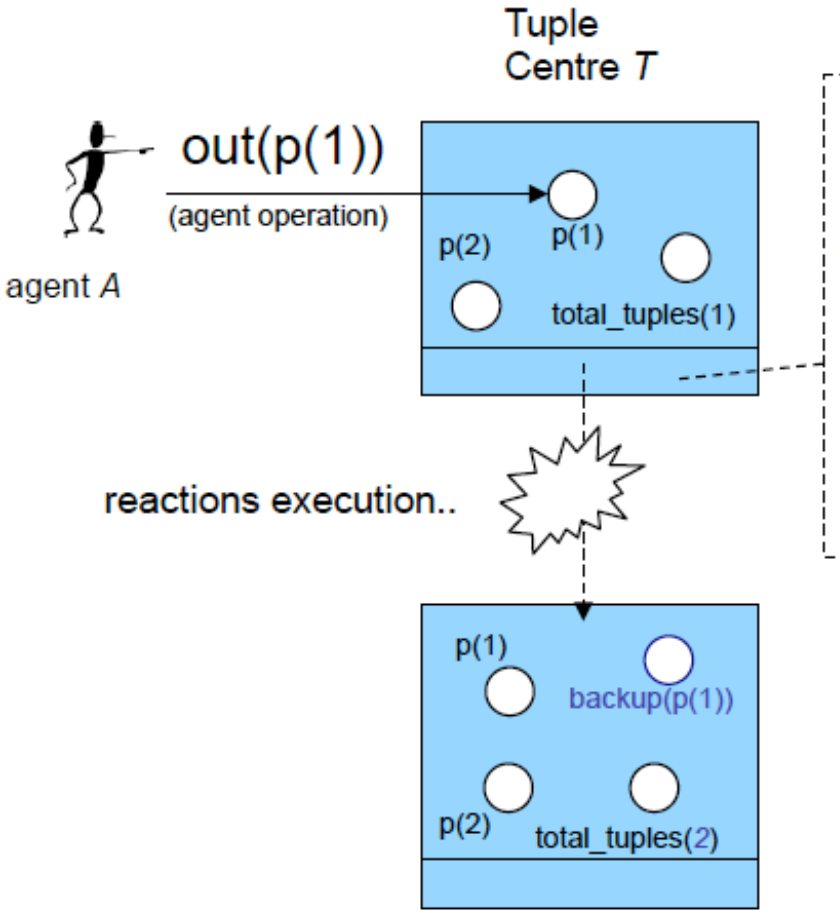


Tuple Centres Features III

- Inspectable at runtime
  - Tuple centre behaviour can be inspected dinamically, at runtime
- Uniformity of languages
  - Same structure / primitives for communication and coordination



Simple Examples I



Current behaviour of the tuple centre (pseudocode):

When a tuple T is inserted, produce a tuple `backup(T)`

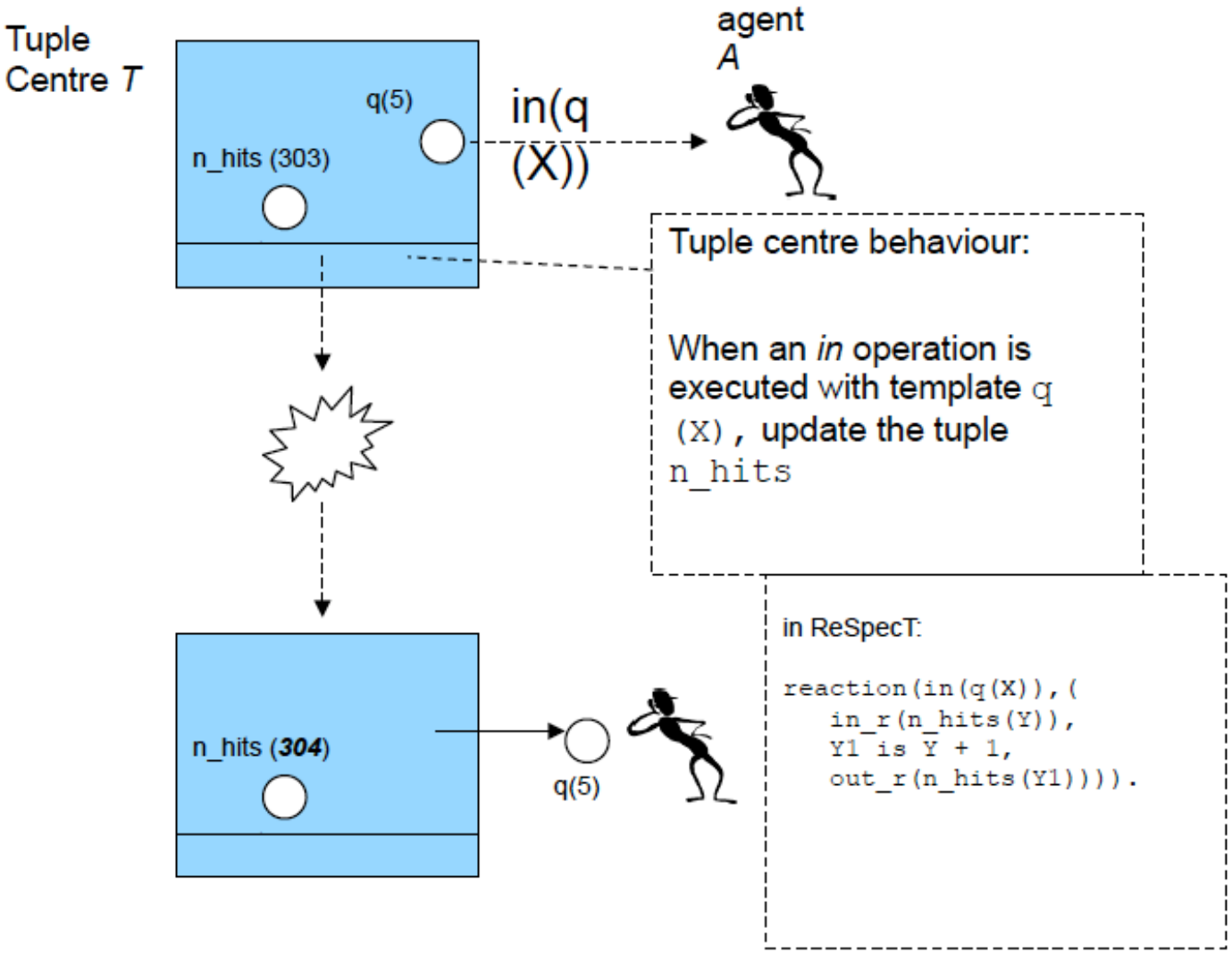
When a tuple p(X) is inserted, update the tuple `total_tuple(N)` (retrieve and store the tuple with N incremented)…

in ReSpecT:

```
reaction(out(T),(
   out_r(backup(T)))).

reaction(out(p(X)),(
   in_r(total_tuples(N)),
   N1 is N + 1,
   out_r(total_tuples(N1)))).
```

Simple Examples II



TuCSoN Technology I

- TuCSoN API
  - Virtually any hosting language, currently Java and Prolog
    - $\Rightarrow$ Support for Java and Prolog agents
  - Heterogeneous hardware support
- TuCSoN Service
  - Booting the TuCSoN Service daemon
    - The host becomes a TuCSoN node
    - With current version (1.4.5):
      `java -cp` *dir*`/tucson.jar alice.tucson.service.Node`

TuCSoN Technology II

- TuCSoN Tools
  - Inspector
    - Fundamental tool to monitor tuple centre communication and coordination state, and to debug tuple centre behaviour
    - With current version (1.4.5):
      `java -cp` *dir*`/tucson.jar alice.tucson.tools.Inspector`
  - TuCSoN Shell
    - Shell interface for human agents
    - With current version (1.4.5):
      `java -cp` *dir*`/tucson.jar alice.tucson.tools.CLIAgent`
- TuCSoN technology is freely available in
  `http://tucson.alice.unibo.it/`

TuCSoN on the Fly

- Booting a TuCSoN node
- Using a tuble centre (as a human agent) by exploiting TuCSoN shell node
- Inspecting and debugging tuple centres by exploiting TuCSoN inspector

Development in TuCSoN

- Building simple systems
  - Experiments with the "Hello world" simple Java agent.
  - Creating simple coordination among Java, human and Prolog agents.

TuCSoN in Java I

```
import alice.tucson.api.*;
import alice.logictuple.*;

public class Test {
  public static void main(String[] args) throws Exception {
    TucsonContext cn = Tucson.enterDefaultContext();
    TupleCentreId tid = new TupleCentreId("test_tc");
    cn.out(tid, new LogicTuple("p",new Value("hello world")));
    LogicTuple t=cn.in(tid, new LogicTuple("p",new Var("X"));
    System.out.println(t);
  }
}
```

TuCSoN in Java II

```
import logictuple.*;
import tucson.api.*;

public class Test2 {
    public static void main (String args[]) throws Exception{

        AgentId aid=new AgentId("agent-0");
        TucsonContext cn = Tucson.enterContext(new DefaultContextDescription(aid));

        // put the tuple value(1,38.5) on the temperature tuple centre
        TupleCentreId tid=new TupleCentreId("temperature");
        LogicTuple outTuple=LogicTuple.parse("value(1,38.5)");
        cn.out(tid,outTuple);

        // retrieve the tuple using value(1,X) as a template
        LogicTuple tupleTemplate=new LogicTuple("value",
                                                new Value(1),
                                                new Var("X"));
        LogicTuple inTuple=cn.in(tid,tupleTemplate);

        cn.exit();
        System.out.println(inTuple);
    }
}
```

TuCSoN in Java III

```
import alice.tucson.api.*;
import alice.logictuple.*;

class MyAgent extends Agent {
  protected void body(){
    try {
      TupleCentreId tid = new TupleCentreId("test_tc");
      out(tid, new LogicTuple("p",new Value("hello world")));
      LogicTuple t=in(tid, new LogicTuple("p",new Var("X"));
      System.out.println(t);
    } catch (Exception ex){}
  }
}

public class Test {
  public static void main(String[] args) throws Exception {
    AgentId aid=new AgentId("alice");
    new MyAgent(aid).spawn();
  }
}
```

TuCSoN in Prolog (tuProlog)

```
:- load_library('alice.tuprologx.lib.TucsonLibrary').
:- solve(go).

go:-
  test_tc ? out(p('hello world')),
  test_tc ? in(p(X)),
  write(X), nl.
```

Remind

- Ruling inter-agent and agent-environment interactions is an environment concern [Weyns et al., 2007]

$\Rightarrow$ TuCSoN is a part of the agent environment

- Until now, we have seen TuCSoN as an infrastructure supporting inter-agent interactions ...
- ... but we can also see TuCSoN as an infrastructure supporting agent-environment interactions
  - Internal environment (work environment)
  - External environment (see Situated ReSpecT [Casadei and Omicini, 2009])

Coordination in Modern Software Systems

- We call context *the physical / virtual and social situation in which an agent is situated* [Moran and Dourish, 2001]
  - $\Rightarrow$ In open world components need some form of context awareness in order to interact with both other agents and environment
- When an agent enters in a new context, the environment should provide it with a sort of control room that provides agents with context awareness [Omicini, 2002]
  - Is the only way in which the agent can perceive the environment as well as ...
  - ... the only way in which the agent can interact

$\Rightarrow$ It is possible to scale with openness of modern software systems

$\Rightarrow$ While the environment manages social coordination, the control room manage coordination of the particular agent

Agent Coordination Context (ACC) [Omicini, 2001] I

- Should be works as a model for the agent environment, by describing the environment where an agent can interact

  Subjective viewpoint — an ACC should provide agents with a suitable representation of the environment where they live, interact, and communicate

  Objective viewpoint — an ACC should provide a framework to express the interaction within a MAS as a whole, i.e. the space of MAS interaction, that is, the admissible interactions occurring among the agents of a MAS, and between the agents of a MAS and the MAS environment

Agent Coordination Context (ACC) [Omicini, 2001] II

- Should enables and rules the interactions between the agent and the environment by defining the space of the admissible agent interactions.

  Subjective viewpoint — the coordination context enables in principle agents to perceive the space where they act and interact, reason about the effect of their actions and communications, and possibly affect the environment to accomplish their own goals.

  Objective viewpoint — the coordination context would allow engineers to encapsulate rules for governing applications built as agent systems, mediate the interactions amongst agents and the environment, and possibly affect them so as to change global application behaviour incrementally and dynamically.

Agent Coordination Context (ACC) [Omicini, 2001] III

- Should be conceived not only as a tool for human designers, but also as a run-time abstraction provided as a service to agents by a suitable infrastructure.
  - $\Rightarrow$ agent model or behaviour is not constrained *a priori*

$\Rightarrow$ Two basic stages characterize the ACC dynamics [Ricci et al., 2006]:

ACC negotiation — An ACC is meant to be negotiated by the agents with the MAS infrastructure, in order to start a working session inside an organisation, specifying which roles to activate

ACC use — The agent then can use the ACC to interact with the organisation environment, by exploiting the actions / perceptions enabled by the ACC

Agent Coordination Context (ACC) [Omicini, 2001] IV

- Should be dynamically configurable and inspectable by both agents and humans.
  - Configurability would allow a MAS to evolve at run time, by suitably adapting its behaviour to changes.
  - Inspectability would allow both humans and intelligent agents to reason about the current laws of coordination as represented and embodied within coordination contexts, and to possibly change them by properly reconfigure coordination contexts according to new application needs.

ACC in MAS Infrastructure [Ricci et al., 2006] I

- ACC framework is orthogonal both to the specific computational model(s) adopted to define agent behaviour, and to the interaction model(s) adopted to specify how agents communicate, and more generally, interact within the environment.
  - $\Rightarrow$ It is possible to extend any MAS infrastructure with the ACC framework in order to support the organisation and security features.
- As minimum requirements, the infrastructure must explicitly define two different models:
  - It must provide a model of interaction, expressing agent / perceptions (including eventually communication).
  - It should specify a basic organisational model, at least including the notion of agent identity.

ACC in MAS Infrastructure [Ricci et al., 2006] II

- A general model of ACC can be defined, by describing three distinct aspects characterizing the ACC concept [Ricci et al., 2006]:

  ACC Interface — It defines the set of admissible operations provided by the infrastructure for interacting with the (social and resource) environment

  ACC Contract — It is a description of the relationships between the agent and the (organisation) environment where the agent is playing, in particular of the policy enacted by the ACC ruling agent actions and interaction protocols

  ACC Configuration — The ACC configuration is a description of the run-time state of the ACC, concerning the evolution of ongoing interaction protocols

ACC as a Unifying Abstraction for Organisation and Security I

- ACC can be exploited as a unifying abstraction to face a number of otherwise heterogeneous issues in the modelling and engineering of MASs where MASs are seen as structural / social settings [Ricci et al., 2006]. In particular:
  - Modelling Organisation
  - Modelling Access Control
  - Modelling the Quality of Interaction
  - Modelling Relationships between Agents and Institutions

ACC as a Unifying Abstraction for Organisation and Security II

Modelling Organisation — When engineering complex software systems by adopting agent-oriented abstractions, organisation emerges a fundamental dimension [Omicini et al., 2005a]. The ACC abstraction makes it possible to explicitly model the presence of an agent in an organisational context where specific structures and rules are defined.

Modelling Access Control — Agent autonomy and system openness are among the main features that make the engineering of security particularly challenging in the context of MASs [Omicini and Ricci, 2004]. The governing behaviour enacted by the ACC on the agent actions makes this abstraction suitable to model forms of dynamic access control to environment resources.

ACC as a Unifying Abstraction for Organisation and Security III

Modelling the Quality of Interaction — As an interface, the ACC is the conceptual framework place where non-functional properties related to the quality of the interaction / communication can be suitably modelled [Ricci and Omicini, 2002].

Modelling Relationships between Agents and Institutions — The ACC represents a contract between the agent and the institution (organisation) that released it.

ACC as a Unifying Abstraction for Organisation, Coordination, and Security IV

- Conceiving and representing different issues exploiting a unified abstraction have several benefits [Omicini and Ricci, 2003]. In particular:
  - Conceptual economy is obviously the first benefit
  - A common framework is the most obvious way to consistently support adaption and evolution of such issues
  - There are system aspects that can be modelled and engineered in their complex articulation only by considering such issues at the same time

Experiments in TuCSoN Infrastructure

- In [Cremonini et al., 1999] TuCSoN was extended in order to deal with security and topology issues
- The access control model adopted, however was unrelated from organisation specification and management

  $\Rightarrow$ In [Omicini et al., 2005a] the previous approach was integrated with RBAC-like architecture, by explicitly considering access control as linked to organisation structures and rules

1. The Limits of Linda
2. ReSpecT: Programming Tuple Spaces
   - Hybrid Coordination Models
   - Tuple Centres
   - Dining Philosophers with ReSpecT
   - ReSpecT: Language & Semantics
   - Situated ReSpecT
   - Situated ReSpecT: A Case Study
3. TuCSoN: A Space-based Infrastructure
   - The TuCSoN Model
4. Towards a Notion of Agent Coordination Context

Bibliography I

Arbab, F. (2004).
Reo: A channel-based coordination model for component composition.
Mathematical Structures in Computer Science, 14:329–366.

Casadei, M. and Omicini, A. (2009).
Situated tuple centres in ReSpecT.
In Shin, S. Y., Ossowski, S., Menezes, R., and Viroli, M., editors, *24th Annual ACM Symposium on Applied Computing (SAC 2009)*, Honolulu, Hawai'i, USA. ACM.

Cremonini, M., Omicini, A., and Zambonelli, F. (1999).
Multi-agent systems on the Internet: Extending the scope of coordination towards security and topology.
In Garijo, F. J. and Boman, M., editors, *Multi-Agent Systems Engineering*, volume 1647 of *LNAI*, pages 77–88. Springer-Verlag.
9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'99), Valencia, Spain, 30 June– 2 July 1999. Proceedings.

Dastani, M., Arbab, F., and de Boer, F. S. (2005).
Coordination and composition in multi-agent systems.
In Dignum, F., Dignum, V., Koenig, S., Kraus, S., Singh, M. P., and Wooldridge, M. J., editors, *4rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005)*, pages 439–446, Utrecht, The Netherlands. ACM.

Bibliography II

Dijkstra, E. W. (2002).
Co-operating sequential processes.
In Hansen, P. B., editor, *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*, chapter 2, pages 65–138. Springer.
Reprinted. 1st edition: 1965.

Moran, T. and Dourish, P. (2001).
Introduction to this special issue on context-aware computing.
Human-Computer Interaction, 20(2–4):87–95.

Omicini, A. (2001).
On the notion of agent coordination context: Preliminary notes.
*AI*IA Notizie*, XIV(4):44–46.

Omicini, A. (2002).
Towards a notion of agent coordination context.
In Marinescu, D. C. and Lee, C., editors, *Process Coordination and Ubiquitous Computing*, chapter 12, pages 187–200. CRC Press, Boca Raton, FL, USA.

Omicini, A. and Denti, E. (2001).
From tuple spaces to tuple centres.
Science of Computer Programming, 41(3):277–294.

Bibliography III


Omicini, A. and Ricci, A. (2003).
Reasoning about organisation: Shaping the infrastructure.
*AI*IA Notizie*, XVI(2):7–16.


Omicini, A. and Ricci, A. (2004).
MAS organisation within a coordination infrastructure: Experiments in TuCSoN.
In Omicini, A., Petta, P., and Pitt, J., editors, *Engineering Societies in the Agents World IV*, volume 3071 of *LNAI*, pages 200–217. Springer-Verlag.
4th International Workshop (ESAW 2003), London, UK, 29–31 October 2003. Revised Selected and Invited Papers.


Omicini, A., Ricci, A., and Viroli, M. (2005a).
RBAC for organisation and security in an agent coordination infrastructure.
Electronic Notes in Theoretical Computer Science, 128(5):65–85.
2nd International Workshop on Security Issues in Coordination Models, Languages and Systems (SecCo'04), 30 August 2004. Proceedings.

Bibliography IV

Omicini, A., Ricci, A., and Viroli, M. (2005b).
Time-aware coordination in ReSpecT.
In Jacquet, J.-M. and Picco, G. P., editors, *Coordination Models and Languages*, volume 3454 of *LNCS*, pages 268–282. Springer-Verlag.
7th International Conference (COORDINATION 2005), Namur, Belgium, 20–23 April 2005. Proceedings.

Omicini, A., Ricci, A., and Viroli, M. (2007).
Timed environment for Web agents.
Web Intelligence and Agent Systems, 5(2):161–175.

Omicini, A. and Zambonelli, F. (1999).
Coordination for Internet application development.
Autonomous Agents and Multi-Agent Systems, 2(3):251–269.
Special Issue: Coordination Mechanisms for Web Agents.

Ricci, A. and Omicini, A. (2002).
Agent coordination contexts: Experiments in TuCSoN.
In De Paoli, F., Manzoni, S., and Poggi, A., editors, *AI*IA/TABOO Joint Workshop "Dagli oggetti agli agenti: dall'informazione alla conoscenza" (WOA 2002)*, Milano, Italy. Pitagora Editrice Bologna.

Bibliography V


Ricci, A., Viroli, M., and Omicini, A. (2006).
Agent coordination contexts in a MAS coordination infrastructure.
Applied Artificial Intelligence, 20(2–4):179–202.
Special Issue: Best of "From Agent Theory to Agent Implementation (AT2AI) – 4".


Weyns, D., Omicini, A., and Odell, J. (2007).
Environment as a first-class abstraction in multi-agent systems.
Autonomous Agents and Multi-Agent Systems, 14(1):5–30.
Special Issue on Environments for Multi-agent Systems.

Tuple-based Coordination: From Linda to ReSpecT & TuCSoN

Multiagent Systems LS
Sistemi Multiagente LS

Andrea Omicini
after Matteo Casadei, Elena Nardini, Alessandro Ricci
andrea.omicini@unibo.it

Ingegneria Due
Alma Mater Studiorum—Università di Bologna a Cesena

Academic Year 2009/2010