# Towards Dependable Emergent Ensembles of Components:

# The DEECo Component Model

http://d3s.mff.cuni.cz

*Jaroslav Keznikl, Tomáš Bureš,*

*František Plášil, Michal Kit*

bures@d3s.mff.cuni.cz

CHARLES UNIVERSITY IN PRAGUE

faculty of mathematics and physics

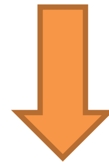
# The Scope

- ASCENS EU project
  - FP7 IP FET
  - 14 partners
  - Goal: Self-aware, self-adaptive systems from components
  - Case-studies:
    - Self-aware and autonomous robots
    - Resource management in cloud computing
    - Intelligent navigation of electric vehicles (VW)

Department of Distributed and Dependable Systems D3S

# Challenges

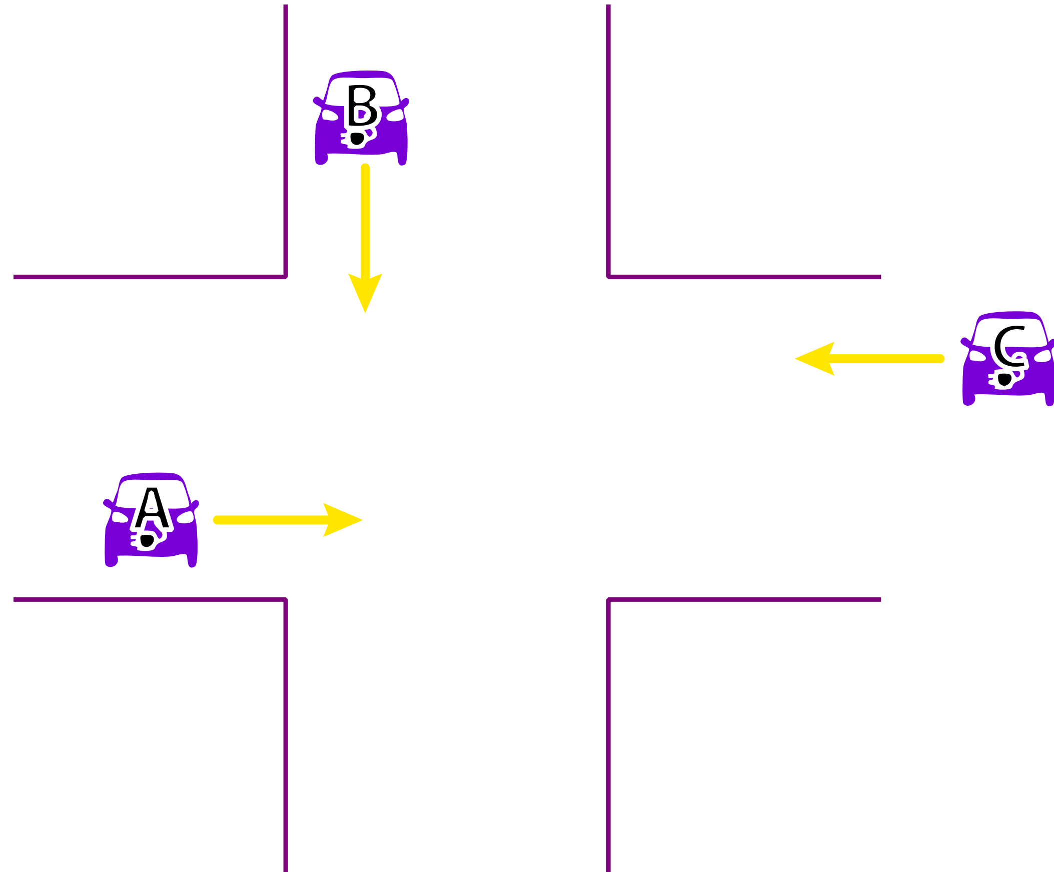
- Dynamic & changing environment, open system
- Massively distributed, ad-hoc networks
  - components appear/disappear
  - connections formed to reflect current „physical“ situation
  - communication is unreliable
  - no notion of the global state of the system



Classical approaches to component architectures **do not scale**

Department of Distributed and Dependable Systems D3S

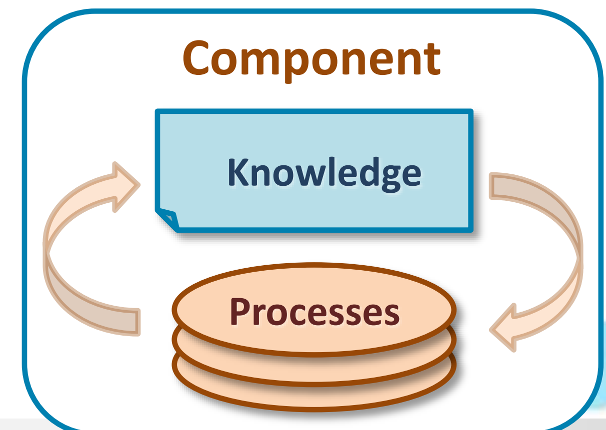
# Example




Department of Distributed and Dependable Systems
D3S

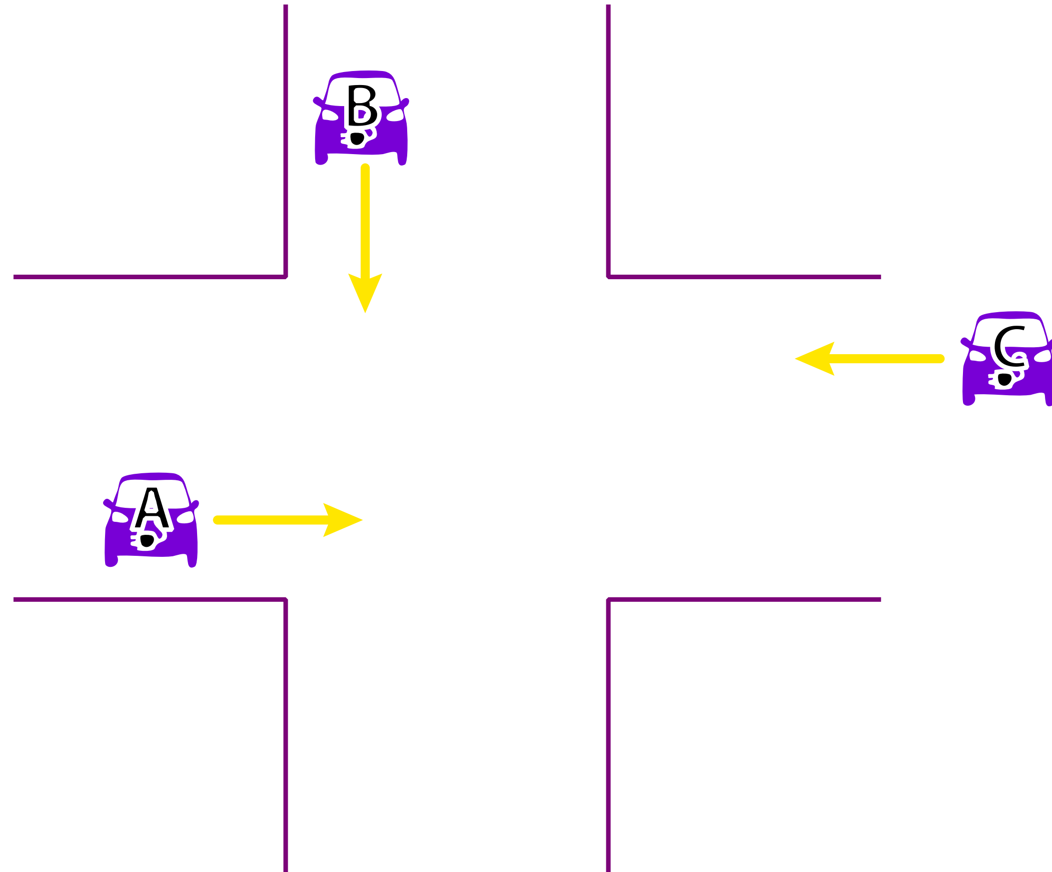
# Solution – DEECo component model

- Component
  - independent autonomous unit responsible for computation and sensing/actuating
  - local knowledge (tree data structure)
  - processes (periodic or event-based) performing computation over the local knowledge
  - **no proactive** (message or procedure call-based) **communication** with other components



# Example

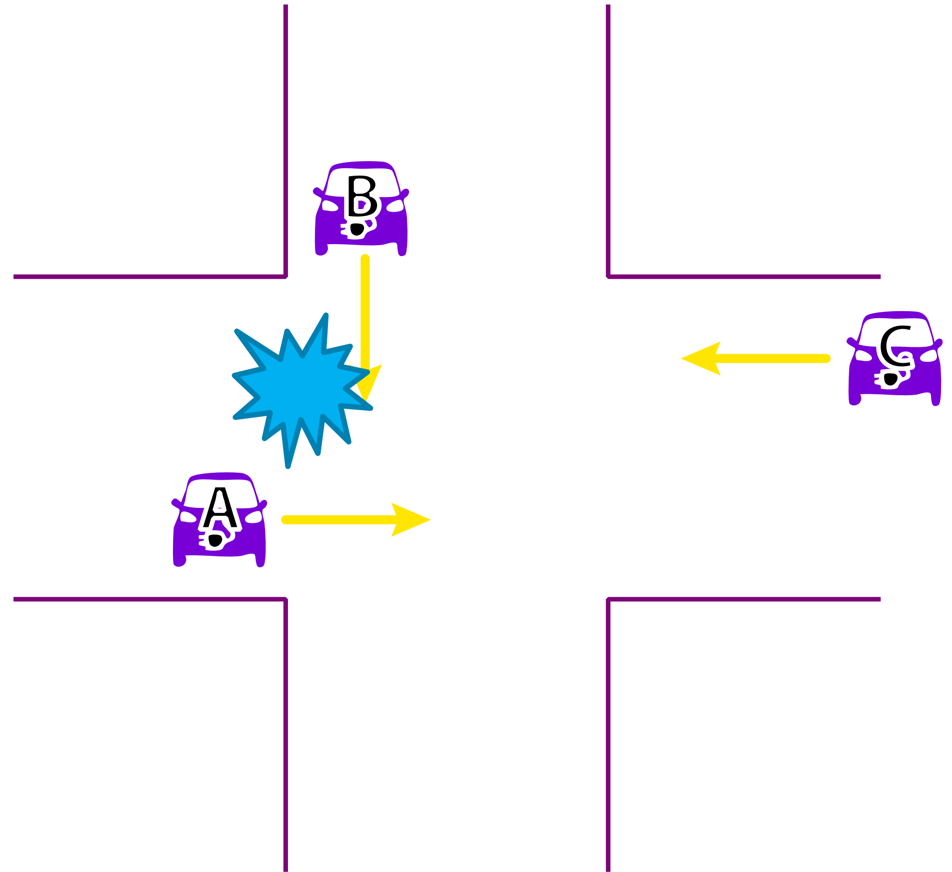
```
component CarA = {

Knowledge
    id: CarId = "A";
    info: CarInfo = {
      position: Position;
      path: list Position;
    };
    otherCars: map CarId -> CarInfo;

Processes
    /* Driving according to common
       traffic rules */
    step: Process = {
        function = fun(inout i: CarInfo,
               in o: map CarId->CarInfo) { ... } ;
        input=[info, otherCars];
        output=[info];
        scheduling = PERIODIC(100ms);
    };

};
```




Department of Distributed and Dependable Systems D3S

# Example

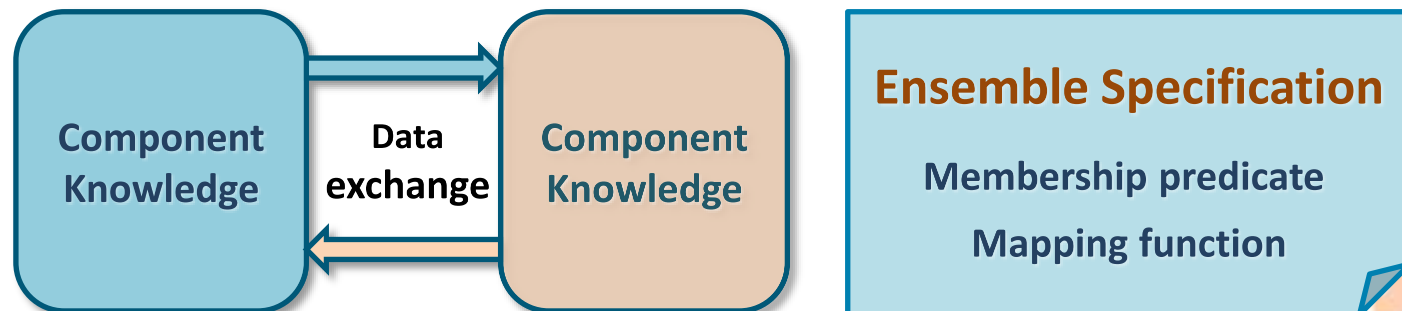
```
component CarA = {

Knowledge
    id: CarId = "A";
    info: CarInfo = {
      position: Position;
      path: list Position;
    };
    otherCars: map CarId -> CarInfo;

Processes
    /* Driving according to common
       traffic rules */
    step: Process = {
        function = fun(inout i: CarInfo,
              in o: map CarId->CarInfo) { ... } ;
        input=[info, otherCars];
        output=[info];
        scheduling = PERIODIC(100ms);
    };

};
```




Department of Distributed and Dependable Systems

# Solution – DEECo component model

- Ensemble
  - group of components cooperating for a common goal
  - formed dynamically based on predicate over components' knowledge
    - membership
  - synchronizes parts of components' knowledge
    - periodic / triggered

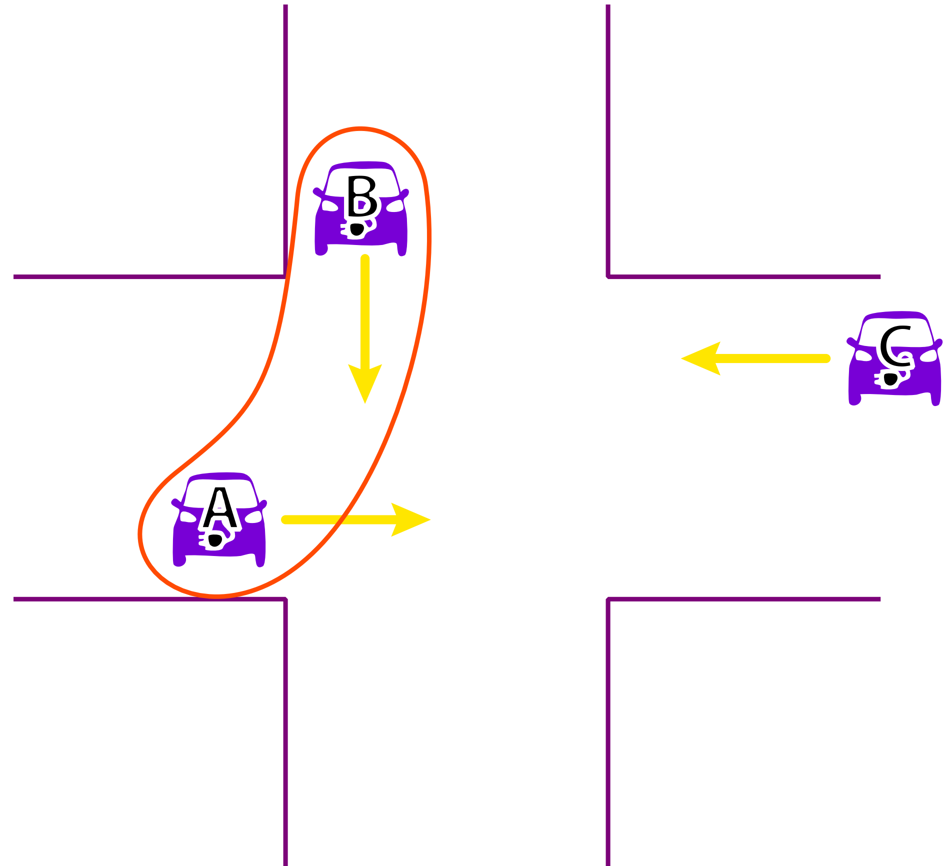


# Example

```
ensemble CarsInCrossingEnsemble {

    membership: fun(in r: ICar, in c: ICar,
                    out ret: Boolean) = {
      ret = bothCarsCloseToSameCrossing(
            r.info.position,
            m.info.position,
            TRESHOLD);
    };

    mapping: fun(inout m: ICar,
                 inout c: ICar) = {

        m.otherCars=m.otherCars.merge(
          c.otherCars).except(m.id);

        c.otherCars[m.id] = m.info;
    };

};
```

Membership

Synchronization




Component Processes

```
/* Driving according to common
   traffic rules */
step: Process = {
    function = fun(inout i: CarInfo,
        in o: map CarId->CarInfo) { ... } ;
    input=[info, otherCars];
    output=[info];
    scheduling = PERIODIC(100ms);
};
```

# Example

**ensemble** *CarsInCrossingEnsemble* {

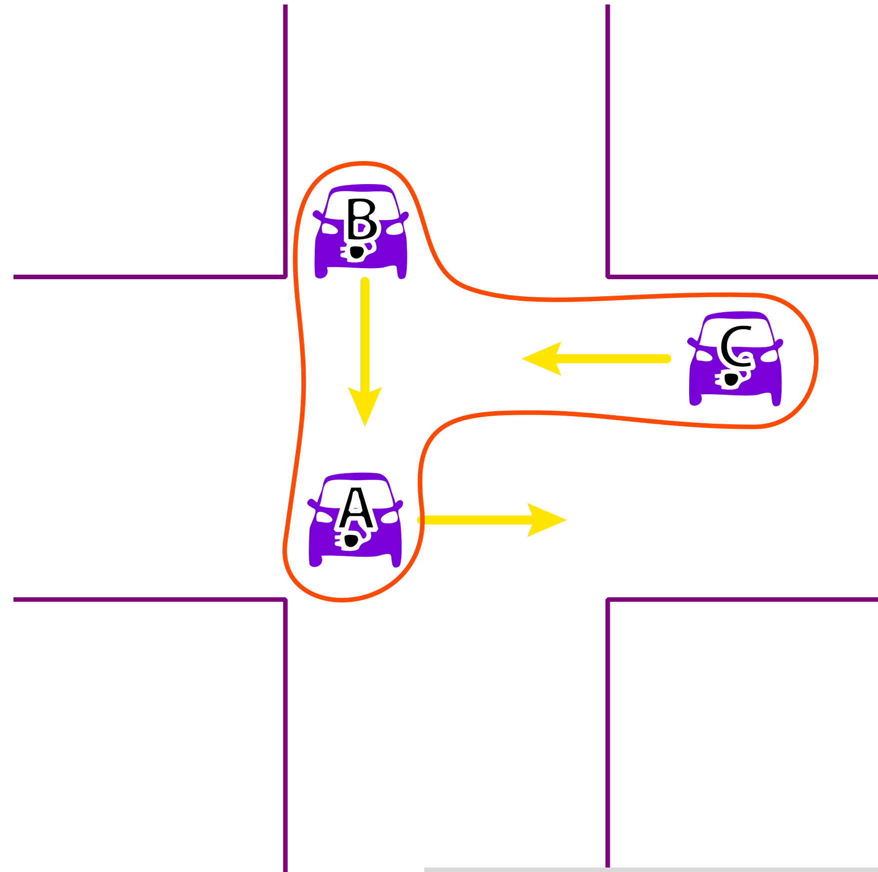
Membership

```
membership: fun(in r: ICar, in c: ICar,
                out ret: Boolean) = {
  ret = bothCarsCloseToSameCrossing(
          r.info.position,
          m.info.position,
          TRESHOLD);
};
```

Synchronization

```
mapping: fun(inout m: ICar,
             inout c: ICar) = {

    m.otherCars=m.otherCars.merge(
      c.otherCars).except(m.id);

    c.otherCars[m.id] = m.info;
};
```

};




Component Processes

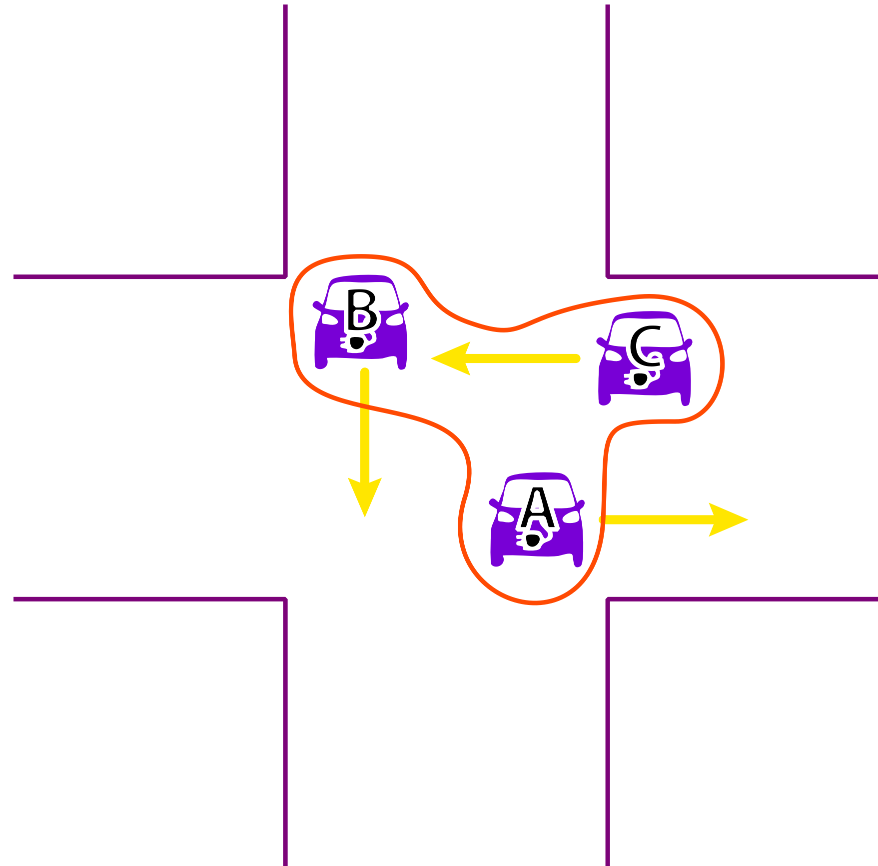
```
/* Driving according to common
   traffic rules */
step: Process = {
    function = fun(inout i: CarInfo,
        in o: map CarId->CarInfo) { ... } ;
    input=[info, otherCars];
    output=[info];
    scheduling = PERIODIC(100ms);
};
```

# Example

```
ensemble CarsInCrossingEnsemble {

  membership: fun(in r: ICar, in c: ICar,
                  out ret: Boolean) = {
    ret = bothCarsCloseToSameCrossing(
            r.info.position,
            m.info.position,
            TRESHOLD);
  };

  mapping: fun(inout m: ICar,
               inout c: ICar) = {

      m.otherCars=m.otherCars.merge(
        c.otherCars).except(m.id);

      c.otherCars[m.id] = m.info;
  };

};
```

Membership

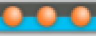
Synchronization




Component Processes

```
/* Driving according to common
   traffic rules */
step: Process = {
    function = fun(inout i: CarInfo,
        in o: map CarId->CarInfo) { ... } ;
    input=[info, otherCars];
    output=[info];
    scheduling = PERIODIC(100ms);
};
```

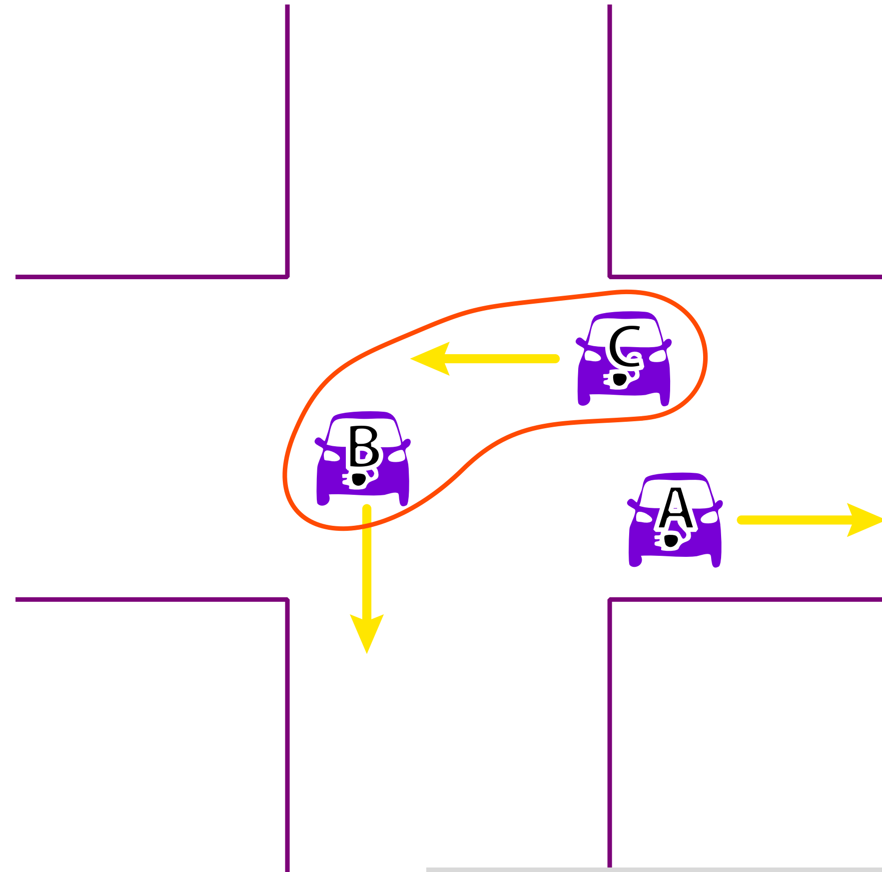
# Example



```
ensemble CarsInCrossingEnsemble {

  membership: fun(in r: ICar, in c: ICar,
                  out ret: Boolean) = {
    ret = bothCarsCloseToSameCrossing(
            r.info.position,
            m.info.position,
            TRESHOLD);
  };

  mapping: fun(inout m: ICar,
               inout c: ICar) = {

    m.otherCars=m.otherCars.merge(
      c.otherCars).except(m.id);

    c.otherCars[m.id] = m.info;
  };

};
```

Membership

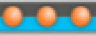
Synchronization




Component Processes

```
/* Driving according to common
   traffic rules */
step: Process = {
    function = fun(inout i: CarInfo,
        in o: map CarId->CarInfo) { ... } ;
    input=[info, otherCars];
    output=[info];
    scheduling = PERIODIC(100ms);
};
```

# Example



**ensemble** *CarsInCrossingEnsemble* {

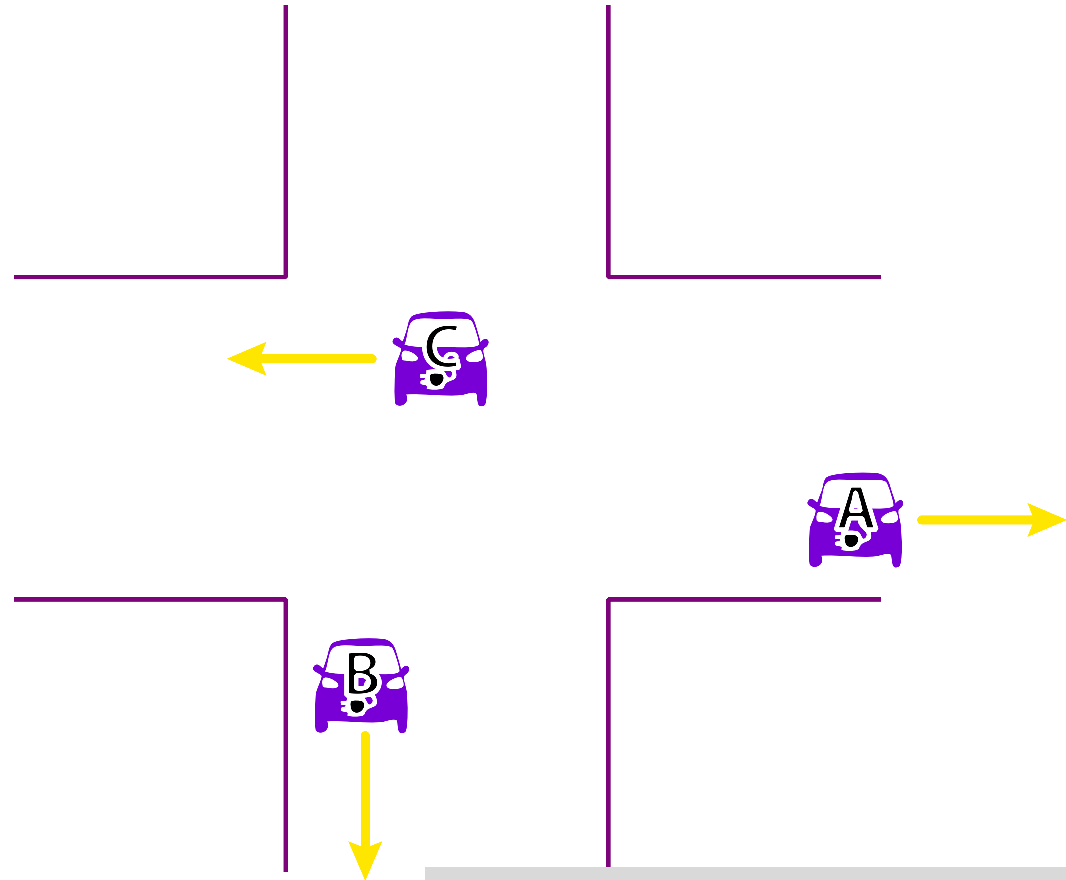
Membership

```
membership: fun(in r: ICar, in c: ICar,
                out ret: Boolean) = {
  ret = bothCarsCloseToSameCrossing(
          r.info.position,
          m.info.position,
          TRESHOLD);
};
```

Synchronization

```
mapping: fun(inout m: ICar,
             inout c: ICar) = {

    m.otherCars=m.otherCars.merge(
      c.otherCars).except(m.id);

    c.otherCars[m.id] = m.info;
};
```

};




Component Processes

```
/* Driving according to common
   traffic rules */
step: Process = {
    function = fun(inout i: CarInfo,
        in o: map CarId->CarInfo) { ... } ;
    input=[info, otherCars];
    output=[info];
    scheduling = PERIODIC(100ms);
};
```

# Conclusion & Future Directions

- DEECo highlights
  - separation of **autonomous computation** x **communication**
  - more resilient to communication failures
  - inherent support for partial belief of the system
  - facilitates development of highly distributed open systems

- Ongoing work
  - Formalization of the behavior
    - SCEL (formal model used in ASCENS)
    - Stochastic model-checking
  - Distributed Java runtime
    - Using JavaSpaces (LINDA-like middleware)
    - https://github.com/d3scomp/JDEECo
  - UML-based system-level design
    - https://github.com/d3scomp/UMLDEECo

Department of Distributed and Dependable Systems D3S