Autonomic component ensembles for dynamic evolving architectures of context aware smart systems

http://d3s.mff.cuni.cz

Tomáš Bureš

bures@d3s.mff.cuni.cz

CHARLES UNIVERSITY IN PRAGUE

faculty of mathematics and physics

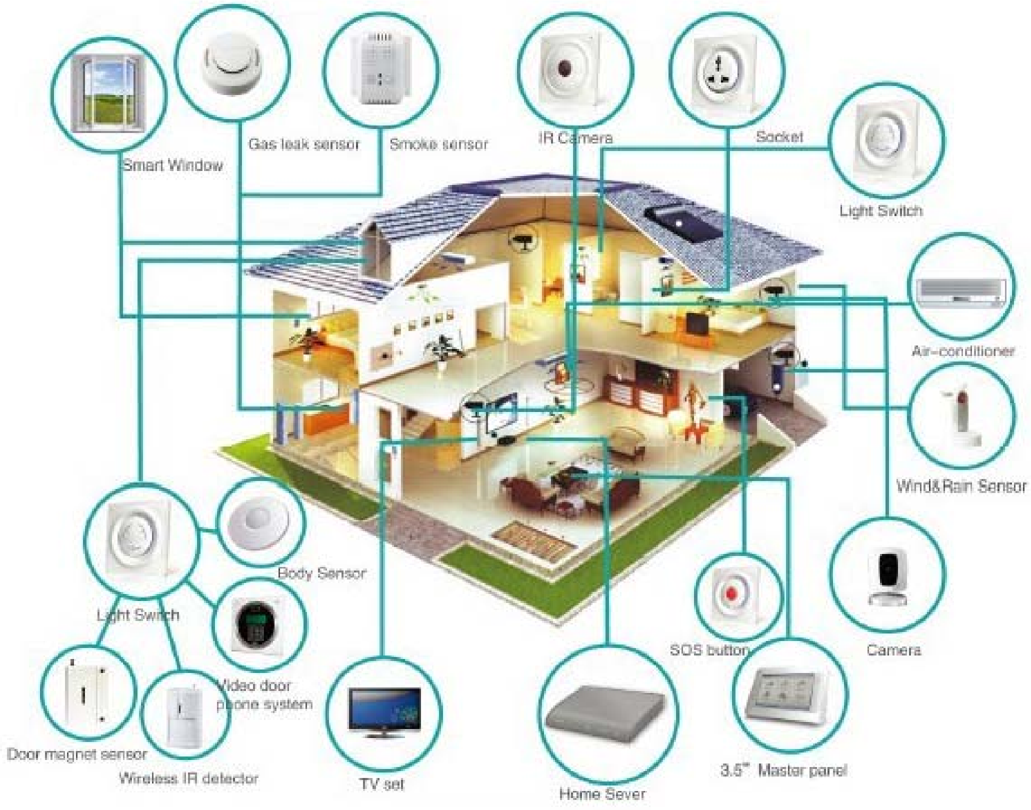
Department of Distributed and Dependable Systems

Context-aware, Autonomous, Smart ?

https://www.petagadget.com/gadget/satechi-revogi-bluetooth-4-0-rgbw-smart-led-bulb/

https://www.techinasia.com/this-self-watering-plant-pot-just-hit-its-crowdfunding-goal

https://www.pcmag.com/news/350867/smart-fridge-showdown-lg-smart-instaview-vs-samsung-family



http://smarthomeenergy.co.uk/what-smart-home

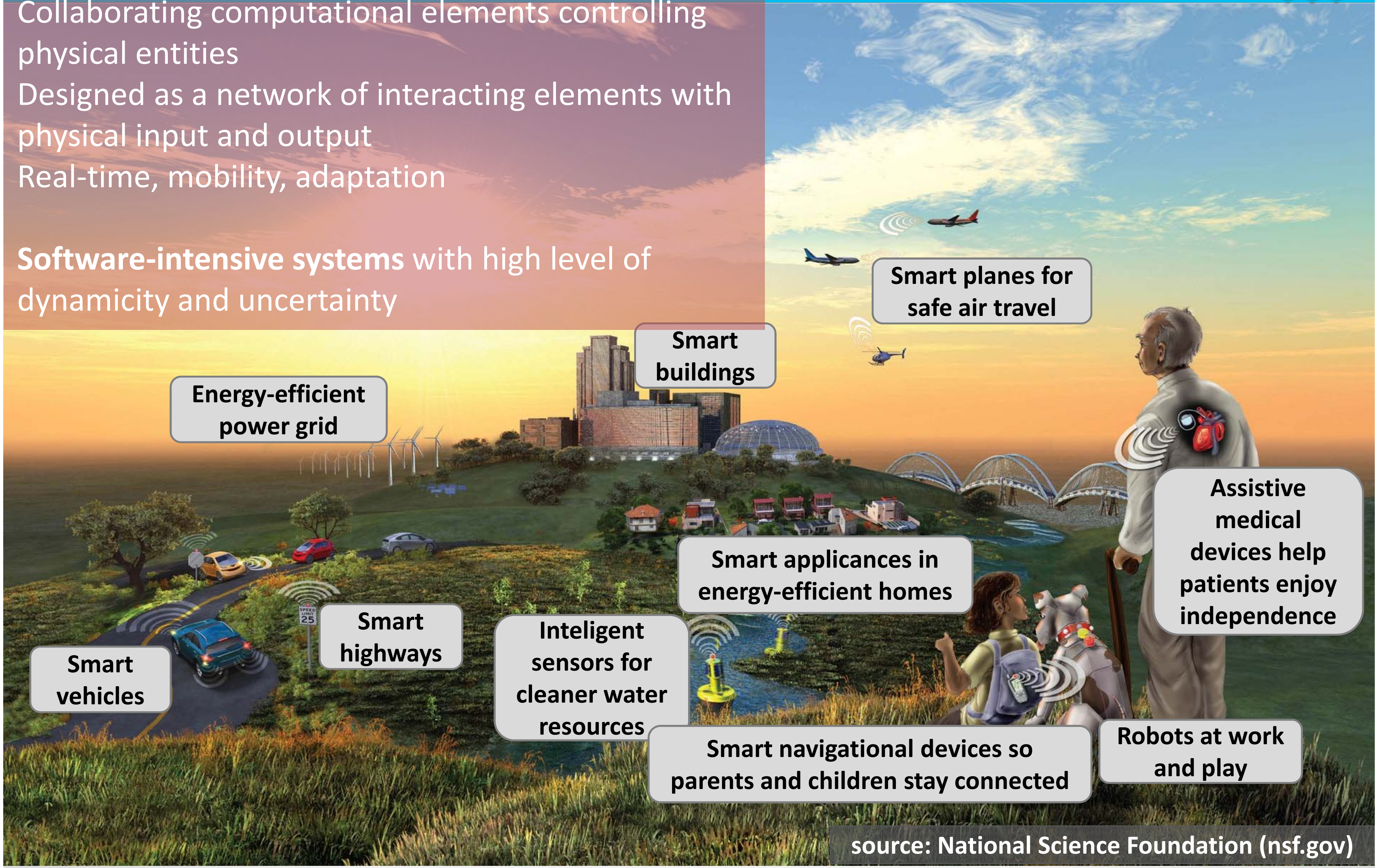
Context-aware, Autonomous, Smart ?

Collaborating computational elements controlling physical entities

Designed as a network of interacting elements with physical input and output

Real-time, mobility, adaptation

Software-intensive systems with high level of dynamicity and uncertainty



Context-aware, Autonomous, Smart ?

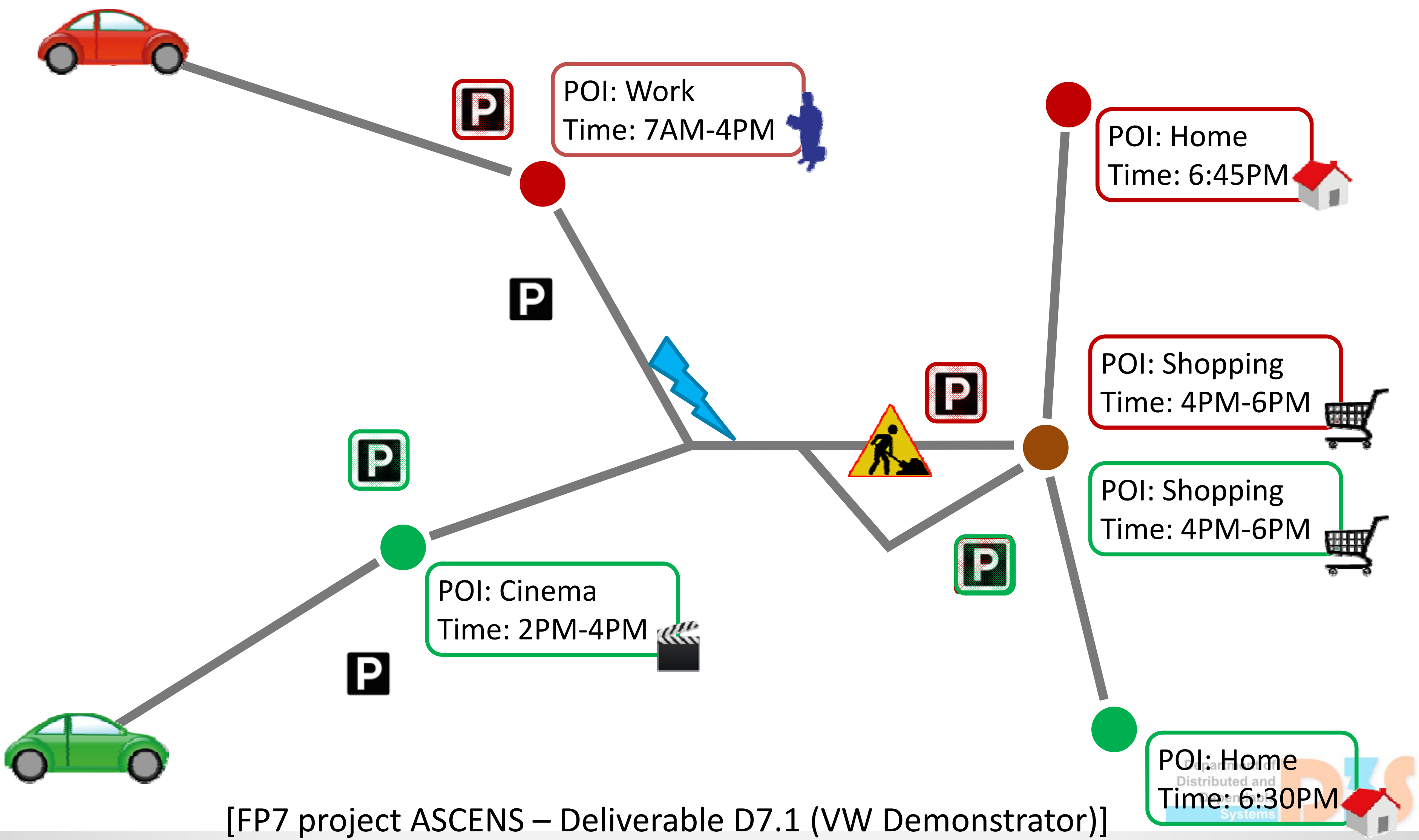
- "Traditional" systems
  - Well known in embedded systems community
  - Focus on HW
  - Limited architecture, limited dynamicity
  - Sometimes not even perceived as distributed

- "Smart" systems
  - Exploit what we pretty already can do by letting things cooperate
  - …and proactively act in their environment

Example: Robot Swarms

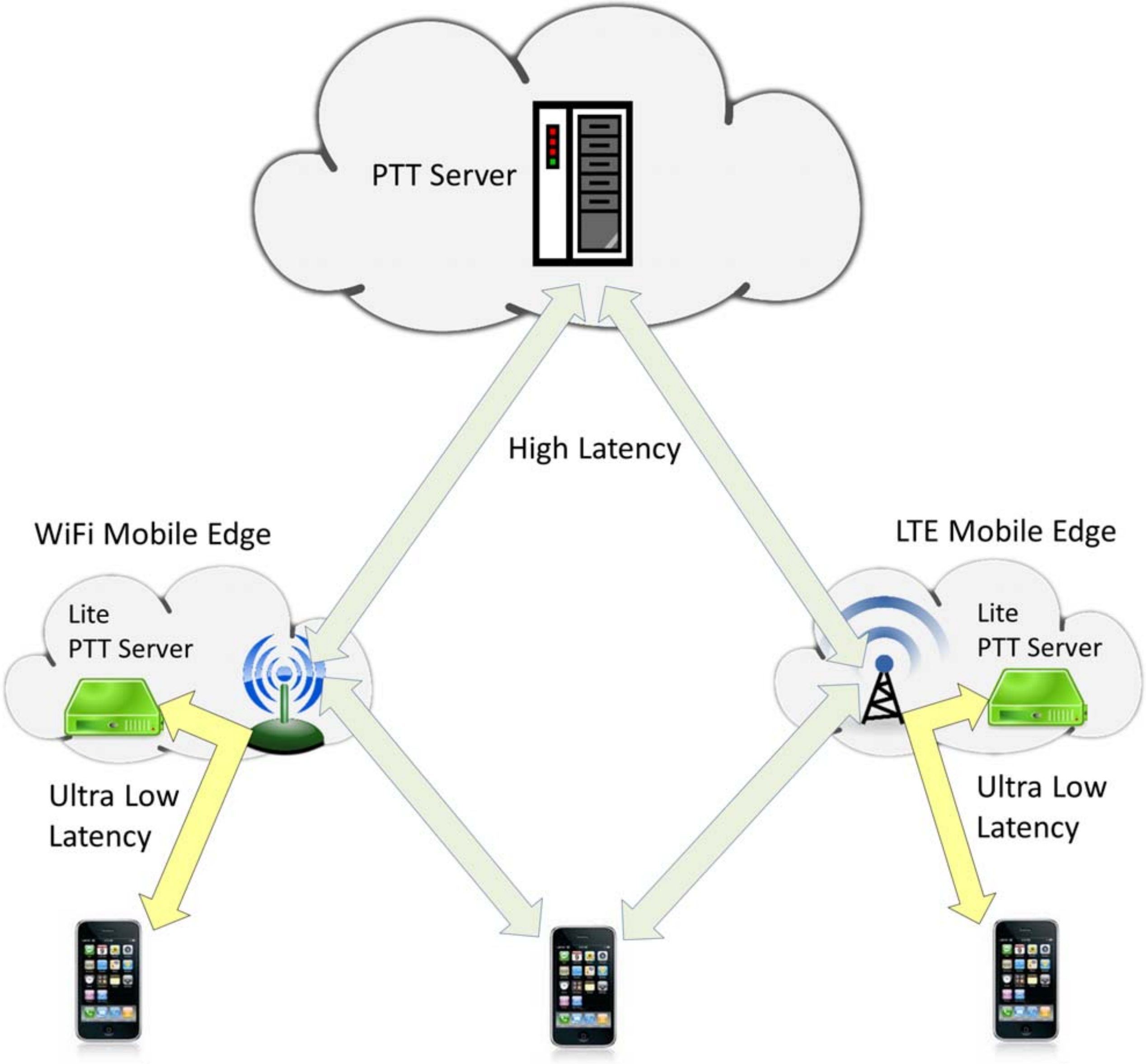
[FP7 project ASCENS – Deliverable D7.1]

Example: E-mobility

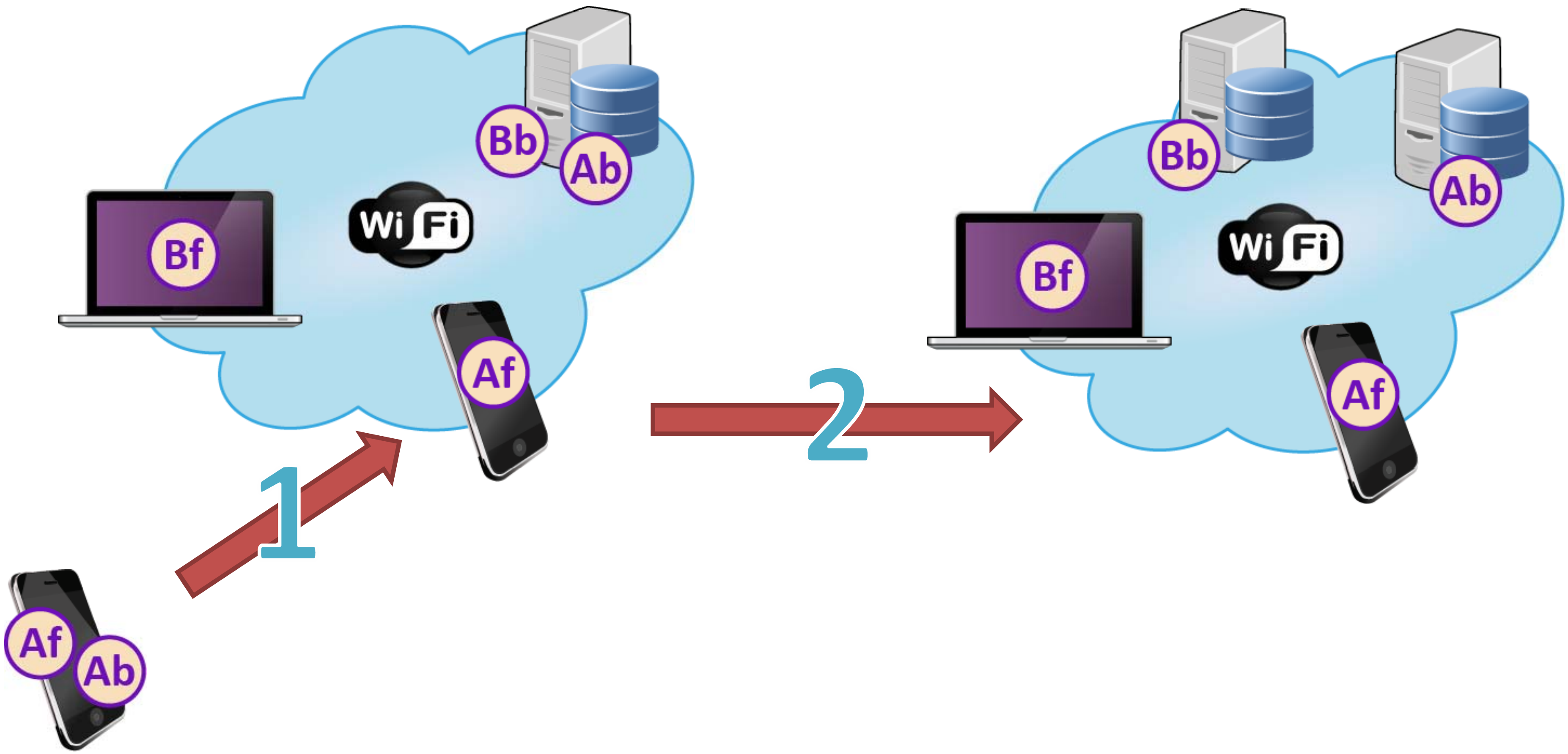


[FP7 project ASCENS – Deliverable D7.1 (VW Demonstrator)]

Example: Mobile Edge Clouds



Example: Ad-hoc Clouds



[FP7 project ASCENS – WP7.2 Demonstrator]

Software architecture challenge

How to model smart, autonomous and context-aware systems to tame their complexity and give some level of predictability

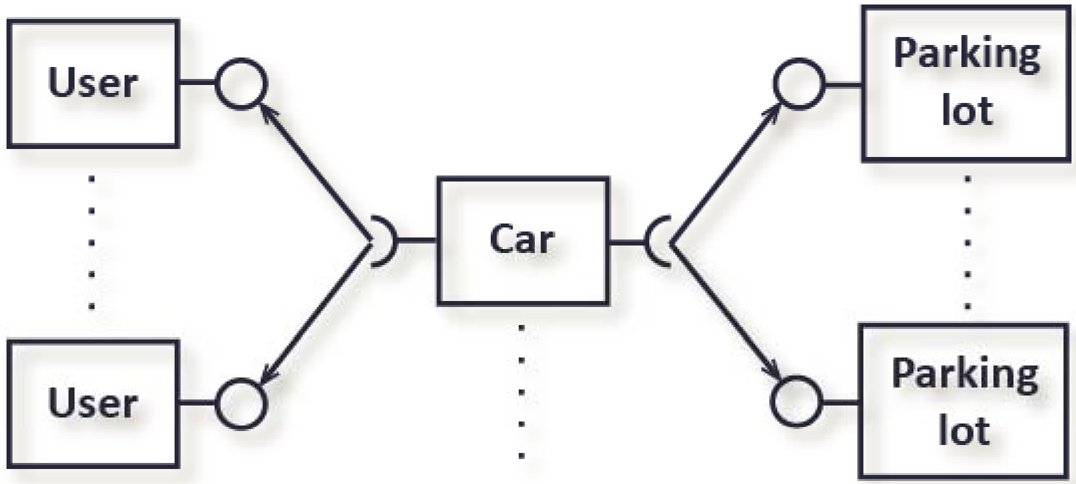
In particular:

- Architectural models
  - Dynamicity
    - Constantly evolving based on situations in the environment
  - Autonomy
    - Tradeoff between centralized and decentralized behavior
    - Ability to make decisions at real-time
  - Adaptivity
    - Ability to function “well” in different (sometimes unforeseen) contexts

Challenges and existing approaches

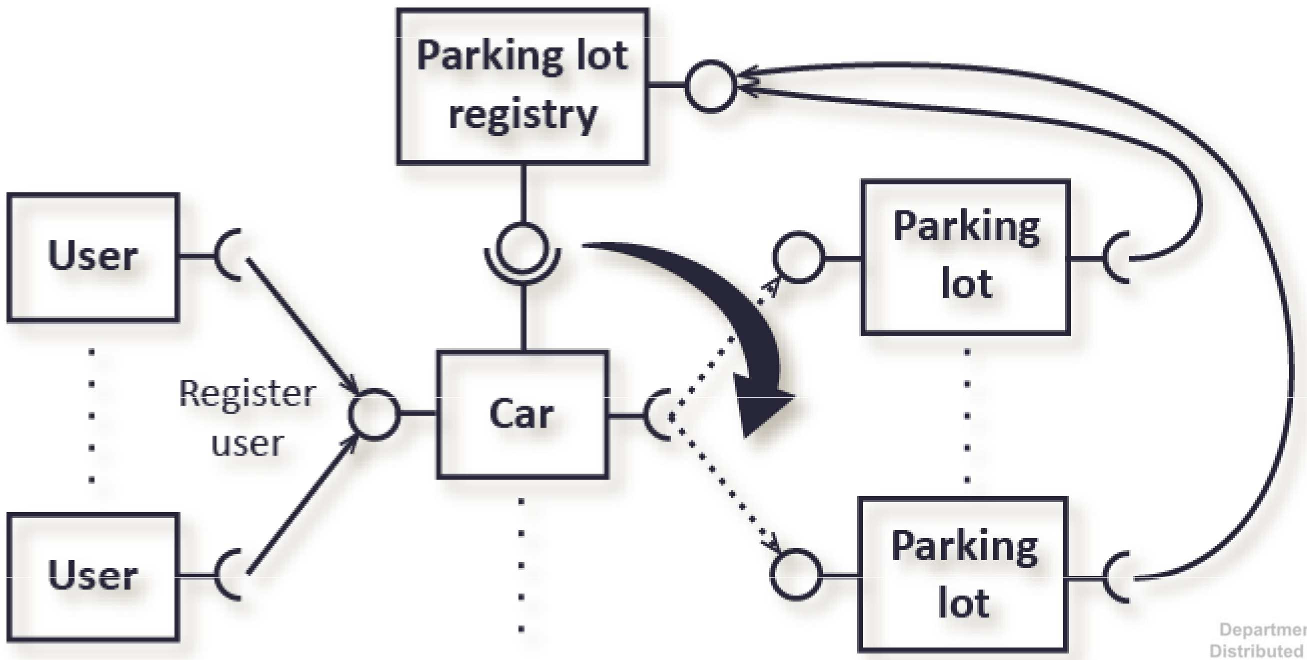
"Classical" Component-Based Approach

- Centralized ownership & deployment
- Cannot capture dynamic changes in architecture
- Strong reliance on other components



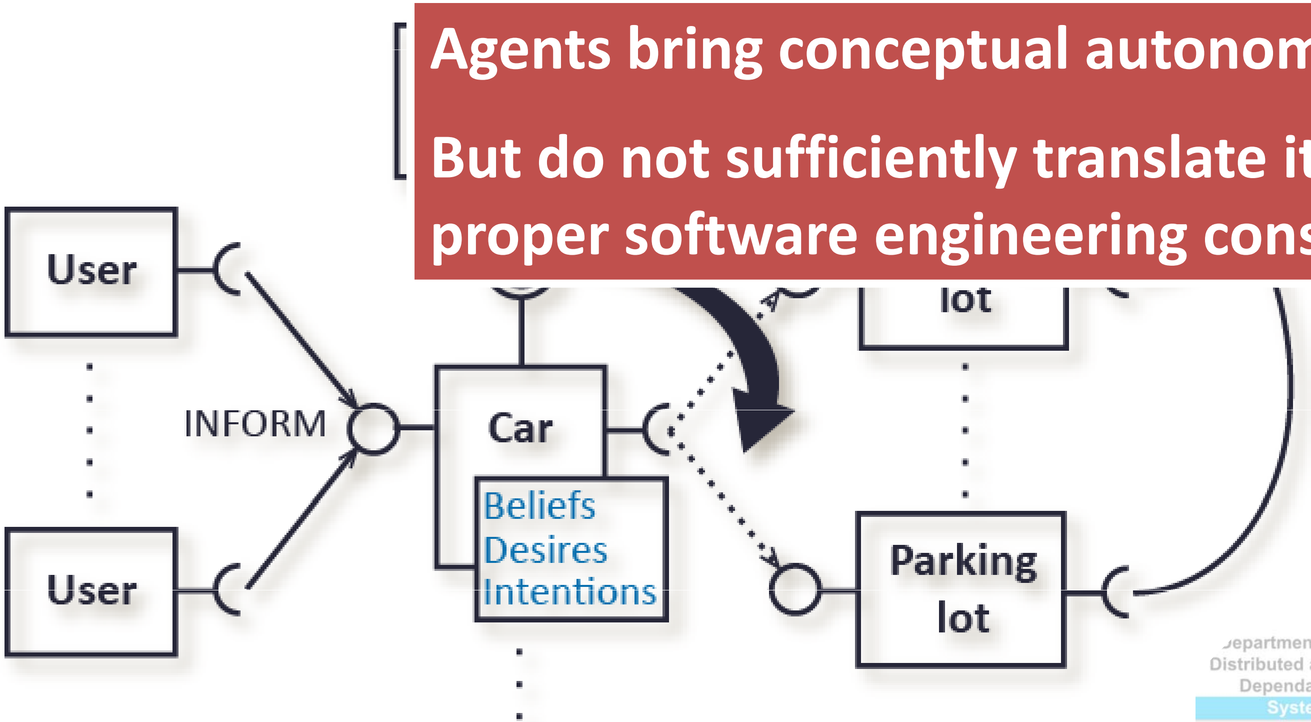
Service-Oriented Approach

- 3-rd party ownership & deployment
- Dynamic composition (but not visible in the architecture)
- Strong reliance on other services



Agent-Based Approach

- 3-rd party ownership & deployment
- Dynamic composition (but no architecture)
- Autonomous (beliefs – desires – intentions)

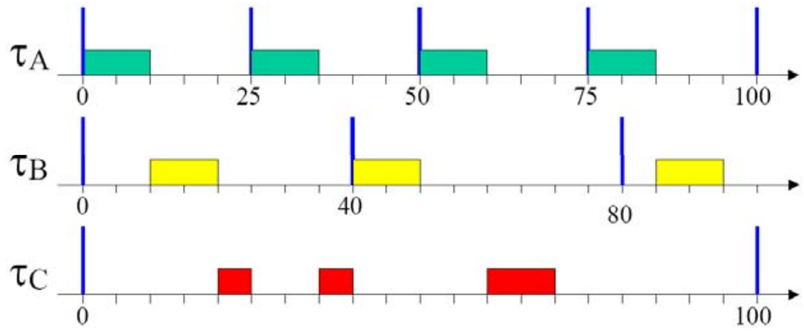


Agents bring conceptual autonomy

But do not sufficiently translate it to proper software engineering constructs

Distributed Control Systems

- Software as a set of real-time tasks
  - Sensing, actuating
  - Real-time scheduling
    - Period, deadline, WCET



- Distributed communication
  - Reliable, real-time guarantees
  - CAN, TTP, FlexRay, …

Attribute-based Communication

- Appears in coordination languages like SCEL, AbC Calculus
  - De Nicola R., Loreti M., Pugliese R., Tiezzi F.: A formal approach to autonomic systems programming: The SCEL language, TAAS vol. 9, issue 2, 2014
  - Alrahman Y. A., De Nicola R., Loreti M.: On the Power of Attribute-based Communication, FORTE 2016

$$\mathbf{qry}(\text{``}\mathit{targetLocation}\text{''}, ?x, ?y)@(\mathit{task} = \text{``}\mathit{task}_1\text{''})$$
$$\mathbf{put}(\text{``}\mathit{targetLocation}\text{''}, x, y)@\mathsf{self}.\, P_1'.$$

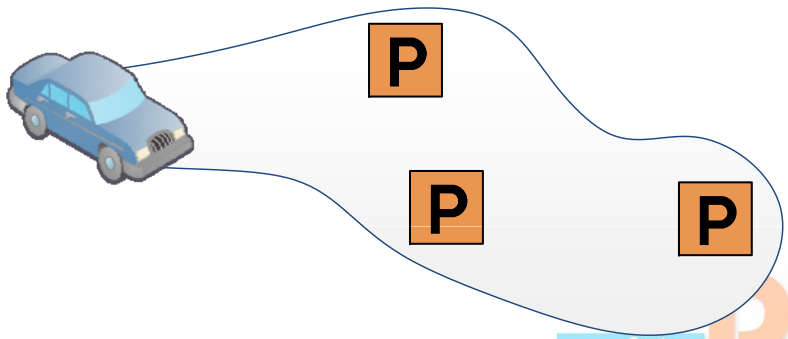
Component Ensembles

Component Ensembles

- Dynamic, goal-oriented groups of components
- Content-based addressing

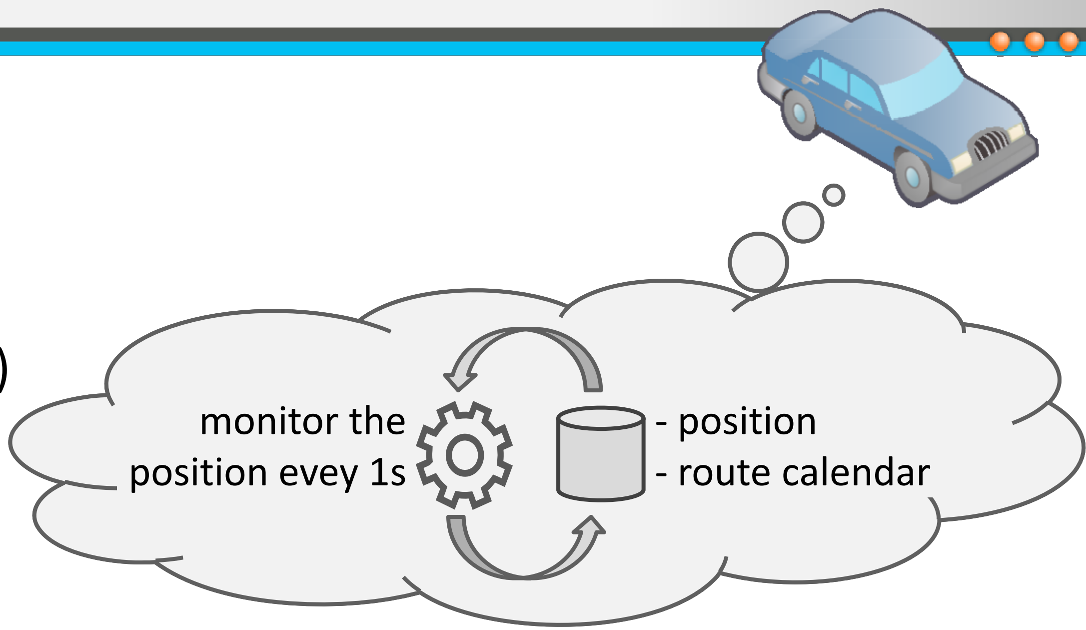
- Components
  - Autonomic
  - (Self-) adaptive

- Ensembles
  - Emergent, distributed
  - Mediate component cooperation to achieve global system goals

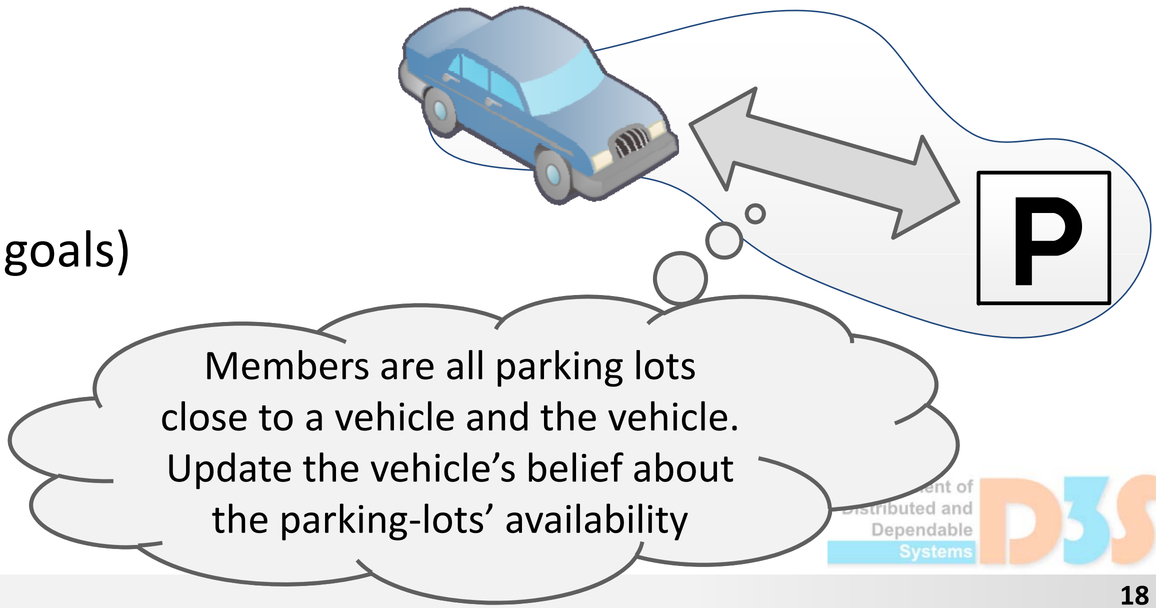


Ensembles-based Component Systems

- Components
  - Knowledge
    - Local data + belief
  - Processes (agent-level goals)
    - Cyclic execution
    - Sensing/actuation

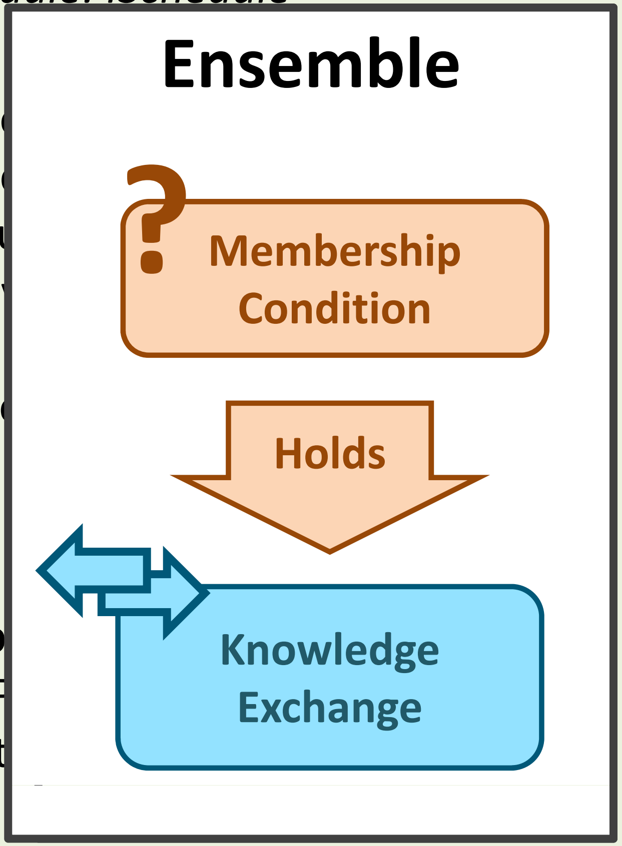


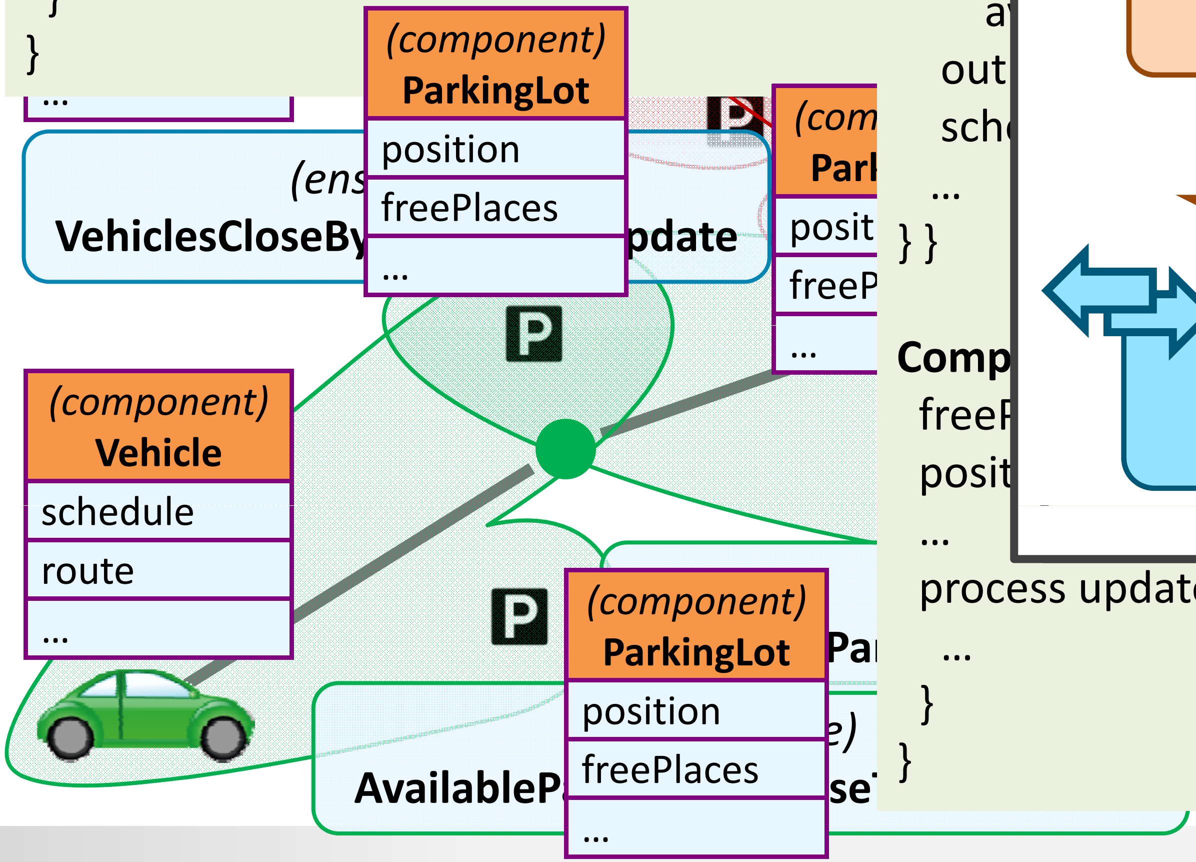
- Ensembles
  - Membership
    - Declarative
  - Coordination (group-level goals)
    - Cyclic execution
    - Dynamic formation



```
Ensemble VehiclesCloseByWithTrafficUpdat
 v1: IVehicle
 v2: IVehicle

 membership :
  proximity(v1.position, v2.position) <= DIS

 coordination {
  v1.trafficInfo <- v2.trafficInfo
 }
}
```

```
Component Vehicle = {
 position: IPosition
 availableParkingLots: IParkingLot[]
 route: IRoute
 schedule: ISchedule
 …
 proc
  fun
  inpu
   a
  out
  sch
 …
} }

Comp
 freeP
 posit
 …
 process updateFreePlaces {
  …
 }
}
```

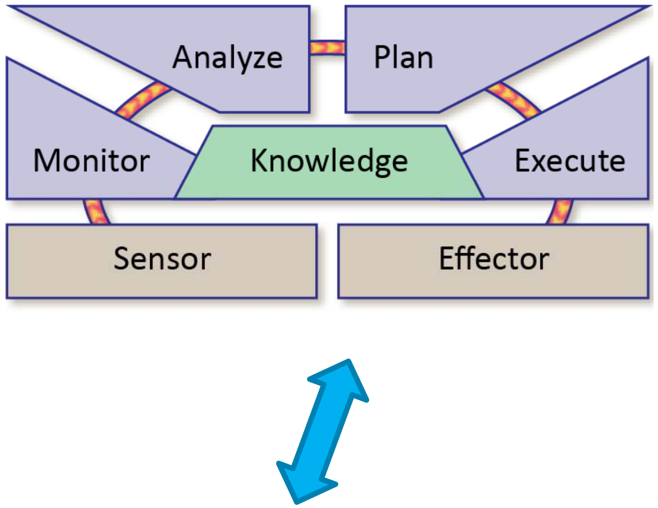


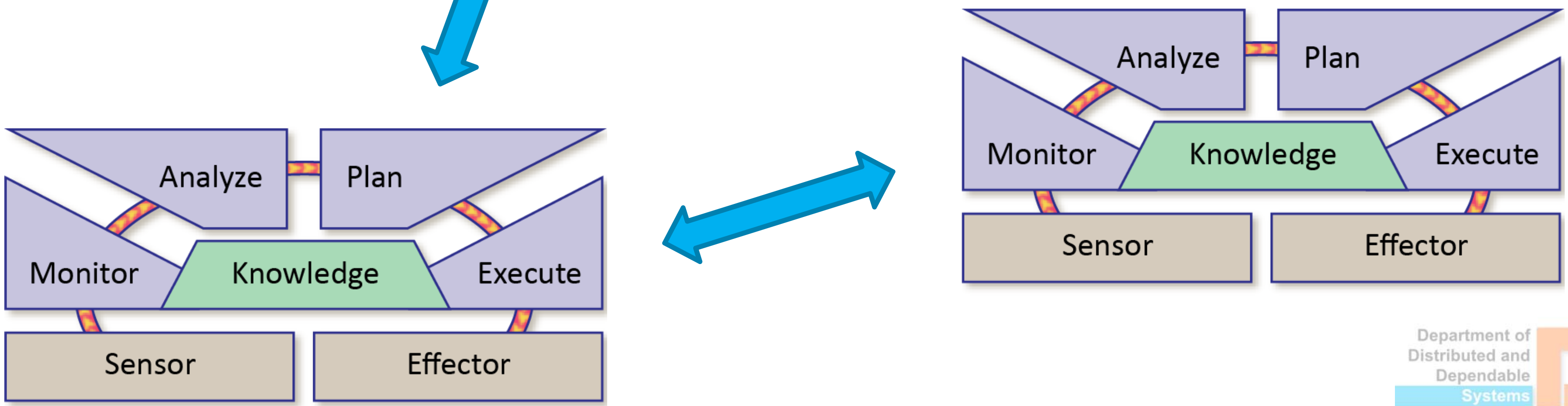


Component Ensembles

Can be seen as a system of conditionally interacting MAPE-K loops

MAPE-K Loop
- **Central concept of autonomic computing**
- **Introduced by IBM**





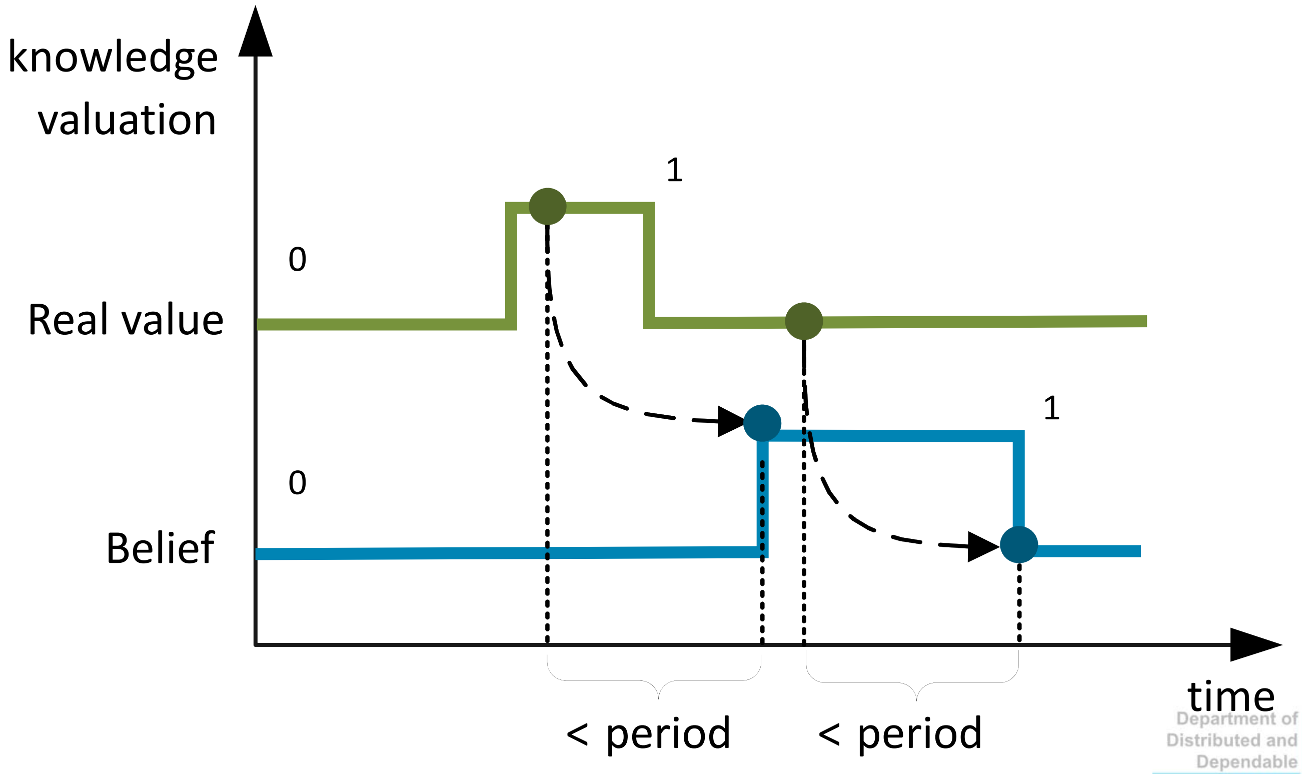
Communication

- Need to distinguish between physical communication and conceptual communication

- Physical communication:
  - Infrastructure-less – gossip with communication boundary
  - Infrastructure-based – decentralized with keys
- Conceptual communication
  - Components see only that which is specified by an ensemble

Communication Latency

- Knowledge evolves
  - Asynchrony, delays due to distribution



Ensembles (DEECo Component Model)

Programming with Ensembles

- DEECo Component model
- Implements the concept of autonomic components and ensembles

- Written in Java
- Available at GitHub
  https://github.com/d3scomp/JDEECo

Hello World Component

```java
@Component
public class HelloWorld {

/**
 * Id of the vehicle component.
 */
public String id;
```

Knowledge

```java
/**
 * Output of count process
 */
public int counter;

public HelloWorld(String id) {
  this.id = id;
  this.counter = 1;
}
```

```java
/**
 * Periodically prints "Hello world!"
 */
@Process
@PeriodicScheduling(period=1000)
public static void sayHello(@In("id") String id) {
  System.out.format("Hello world!\n");
}

/**
 * Periodically increments the counter.
 */
@Process
@PeriodicScheduling(period=500)
public static void updateCounter(
  @InOut("counter") ParamHolder<Integer> counter
) {
  counter.value ++;
}
```

Processes

```java
}
```

Department of Distributed and Dependable Systems

Ensemble

```
@Ensemble
@PeriodicScheduling(period = 1000)
public class FollowerLeaderEnsemble {

public static final double ENSEMBLE_RADIUS = 2000.0; // in meters
```

Membership condition

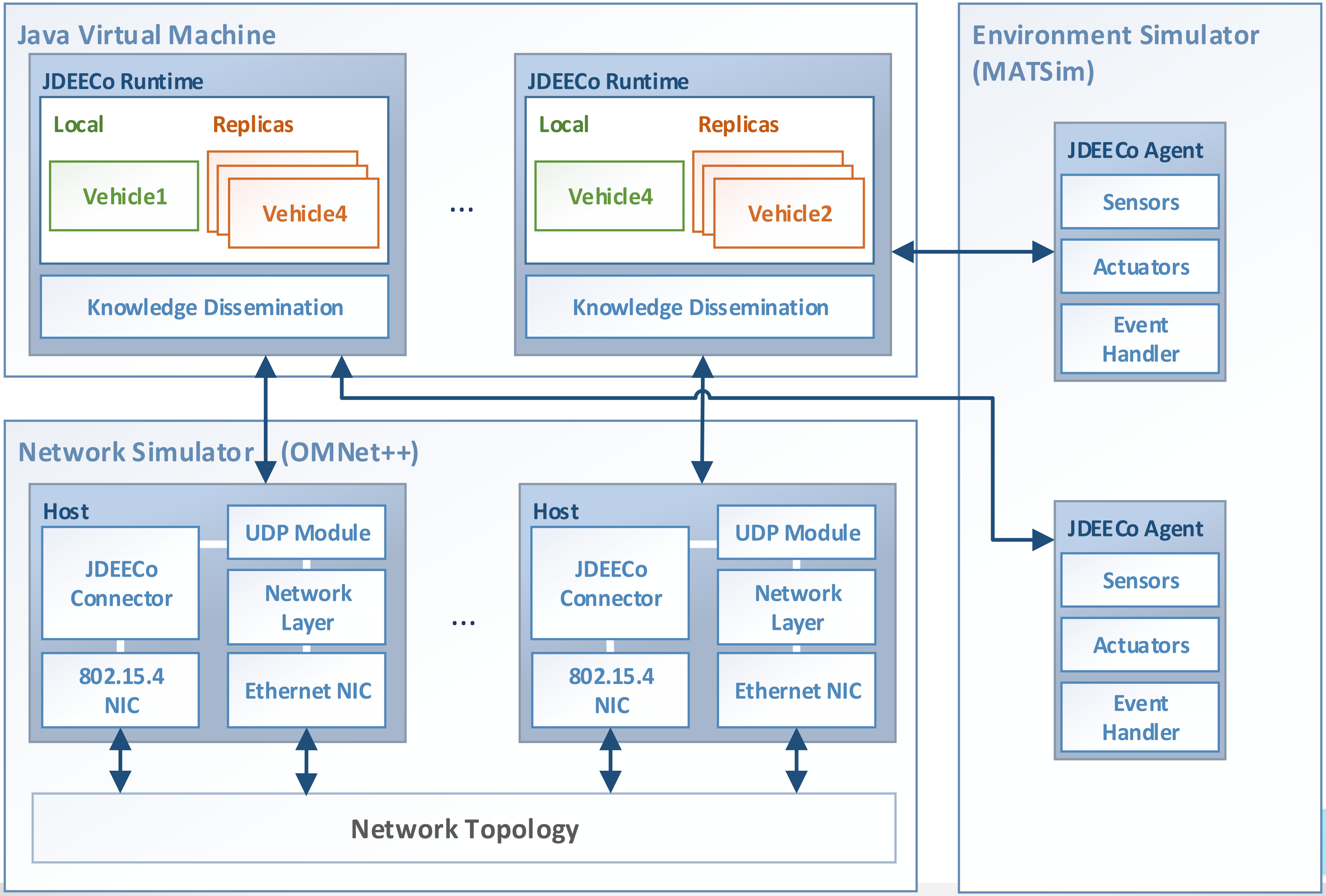
```
@Membership
public static boolean membership(
  @In("member.id") String mId,
  @In("coord.id") String cId,
  @In("member.position") Coord mPos,
  @In("coord.position") Coord cPos) {

  return getEuclidDistance(cPos, mPos) <= ENSEMBLE_RADIUS && cId.compareTo(mId) == -1;
}
```

```
@KnowledgeExchange
public static void exchange(
  @In("coord.destinationLink") Id cDestinationLink,
  @Out("member.leaderDestinationLink") ParamHolder<Id> mLeaderDestinationLink) {

    mLeaderDestinationLink.value = cDestinationLink;
}
```

Knowledge exchange

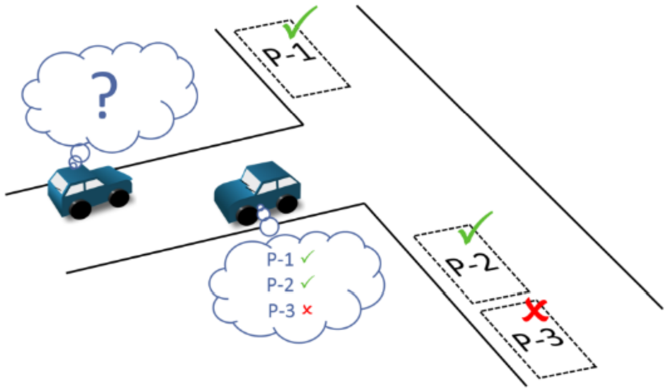
```
...
}
```

JDEECo – Framework for Simulations



Testbeds

- MATSIM-based
  - Simulates mobile components in the urban traffic settings
  - Ad-hoc communication
  - http://self-adaptive.org/exemplars/v2v-DEECo



- ROS-based
  - Simulates mobile robots on a 2D map
  - Ad-hoc communication
  - https://github.com/d3scomp/deeco-adaptation-testbed

Department of Distributed and Dependable Systems

Expressivity of Ensembles

More Expressivity: Intelligent Ensembles

- Establishing ensembles can be perceived as a constraint solving problem

- This allows rich declarative specification of ensembles …

  … and translation of the specification to constraint solving problem that can be consumed by an existing solver

Using External DSL

```
ensemble ProtectionTeam
 id fireLocation: EntityID

 membership
  roles
   brigades [2..3] : FireBrigade where
    (it.state == State.Idle || it.state == State.Protecting) &&
    it.location == location  &&
    FirePredictorValueAt(fireLocation,   RouteTime(it.position, fireLocation)) < 0.9
  fitness sum brigades RouteCost(it.position, fireLocation)

 knowledge exchange
  brigades.assignedFireLocation = fireLocation
end
```

Multi-paradigm Modelling

- Increase the level of abstraction in ensembles by including domain-specific functions (models)

- E.g.:
  - Reachability on a 2D map
  - Reasoning about the potentiality

Using Internal DSL (in Scala)

```
class FireBrigade(val entityID: EntityID) extends Component {
  val Protecting, Refilling, Idle = State
  val Operational = StateOr(Protecting, Refilling, Idle)

  var brigadePosition: Position
  var assignedFireLocation: EntityID

  sensing {
    brigadePosition = agent.getPosition
  }

  constraints {  Operational && (Protecting -> (assignedFireLocation != 0)) && ...   }

  utility {  states.sum(s => if (s == Protecting) 1 else 0  }

  actuation {
    state match {
      case Protecting =>
        if (inExtinguishingDistanceFromFire) extinguish() else moveTo(assignedBuildingOnFire)
      ...
    }
    ...
    sendMessages()
  }
}
```

```
class ProtectionTeam(fireLocation: EntityID) extends Ensemble {
 ...
 val brigades = role("brigades",components.select[FireBrigade])

 val routesToFireLocation = map.shortestPath.to(fireLocation)
 val firePredictor = statespace(burnModel(fireLocation, currentFieriness), currentTime)

 membership {
  brigades.all(brigade => brigade.state == Idle
   || (brigade.state == Protecting && brigade.assignedFireLocation == fireLocation)) &&
  brigades.all(brigade => routesToFireLocation.timeFrom(mapPosition(brigade)) match {
   case None => false
   case Some(travelTime) => firePredictor.valueAt(travelTime) < 0.9
  }) && brigades.cardinality >= 2 && brigades.cardinality <= 3
 }

 utility {
  brigades.sum(b => travelTimeToUtility(routesToFireLocation.timeFrom(mapPosition(b))))
 }

 coordination {
  for (brigade <- brigades.selectedMembers) {
   brigade.assignedFireLocation = Some(fireLocation)
  }
 }
}
```

```
class ProtectionTeam(fireLocation: EntityID) extends Ensemble {
  ...
  val brigades = role("brigades",components.select[FireBrigade])

  val routesToFireLocation = map.shortestPath.to(fireLocation)
  val firePredictor = statespace(burnModel(fireLocation, currentFieriness), currentTime)

  membership {
    brigades.all(brigade => brigade.state == Idle
      || (brigade.state == Protecting && brigade.assignedFireLocation == fireLocation)) &&
    brigades.all(brigade => routesToFireLocation.timeFrom(mapPosition(brigade)) match {
      case None => false
      case Some(travelTime) => firePredictor.valueAt(travelTime) < 0.9
    }) && brigades.cardinality >= 2 && brigades.cardinality <= 3
  }

  utility {
    brigades.sum(b => trave
  }

  coordination {
    for (brigade <- brigades.selectedMembers) {
      brigade.assignedFireLocation = Some(fireLocation)
    }
  }
}
```

High-level specs:
"Select only those firefighters that can reach the building before it burns almost completely"

Goal-based design (via IRM)

Design Process

- Problem:
  - Component ensembles have relatively exotic computational model
    - Very far from classical procedure call-based sequential programming
    - Much closer to design of real-time systems – but also adaptivity and open-endedness
  - Method for high-level design are necessary
    - To help developers "think" about such systems
    - Requirements ⟹ … ⟹ Components + Ensembles

Challenge

High-level
System Requirements

All vehicles meet their
route/parking calendars

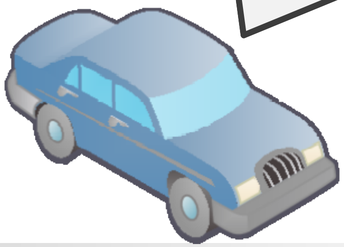
?

Formally grounded, rigorous refinement

Low-level Design
DEECo Processes/Ensembles

- Compute a route plan
- Every 5s check the plan feasibility
- Re-compute the plan if infeasible or once every 60s.

- Every 15s monitor the availability



Every 10s update the vehicle’s belief about availability of nearby parking lots

P

Classical Approaches

Use-cases, User stories, …

Use-case Example

Schedule meeting

1. User enters the possible dates of the meeting
2. Use enters e-mails of the participants
3. System validates the e-mail addresses
4. System sends an e-mail with an invitation to each participant
5. System confirms e-mails being sent

…

Describes "how" instead of "what".

Inherently less adaptable/evolvable.

Goal-oriented model-building at RE time

Goal-orientation enables:

- early, incremental analysis
- completeness and pertinence of the model
- reasoning about alternative options
- validation by stakeholders
- backward traceability

Thinking about goals in the early phases of software development is a natural thing; in GORE it is just made explicit

Approaches in GORE

- **KAOS**

"a GORE approach with a rich set of formal analysis techniques"

→ Axel van Lamsweerde et al.

- **i***

"an agent-oriented modeling framework that can be used for requirements engineering"

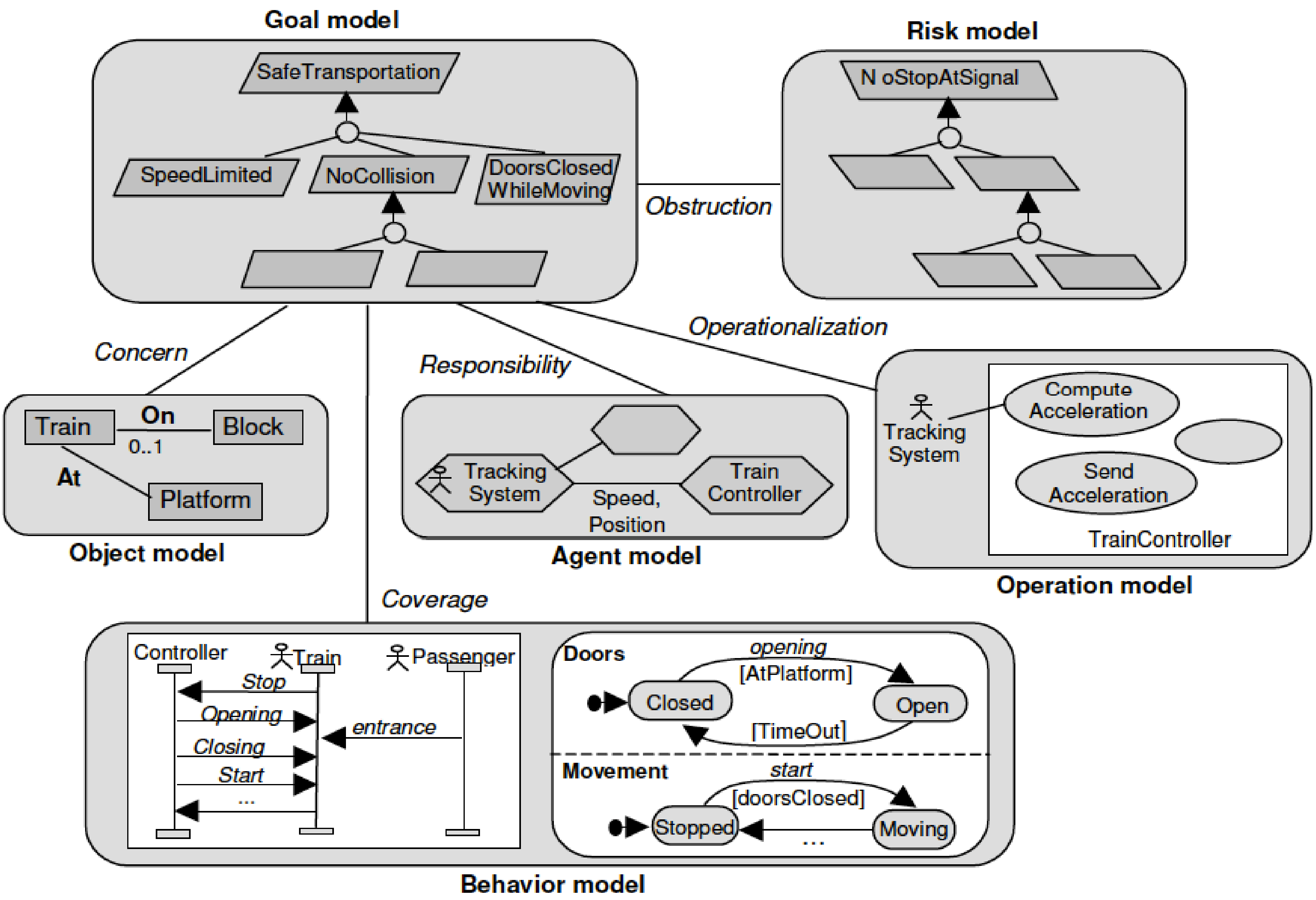
→ Eric Yu

- **TROPOS**

"an agent-oriented software engineering methodology"

→ John Mylopoulos et al.

KAOS multi-view modeling



Goal model - I

behavioral (hard) goals:

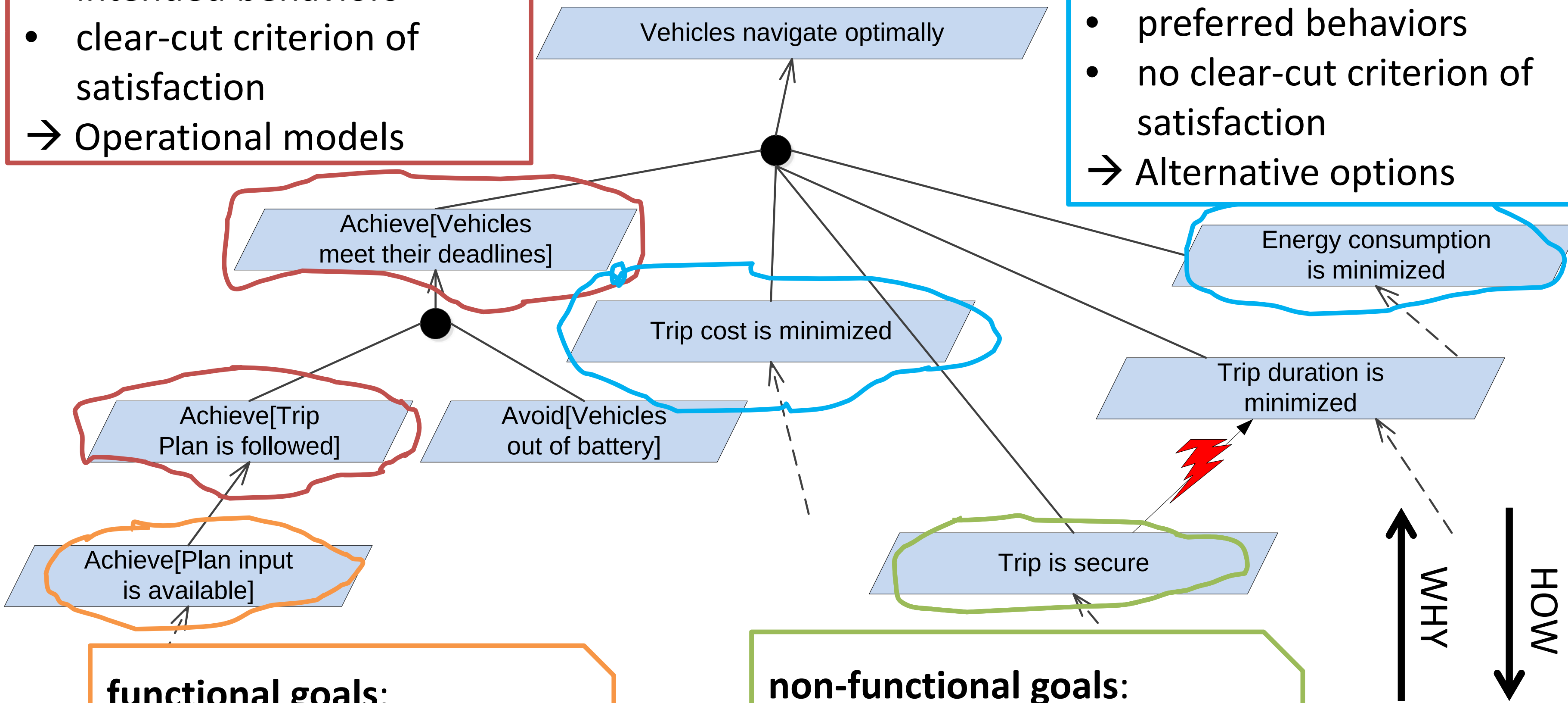
- intended behaviors
- clear-cut criterion of satisfaction

→ Operational models

soft goals:

- preferred behaviors
- no clear-cut criterion of satisfaction

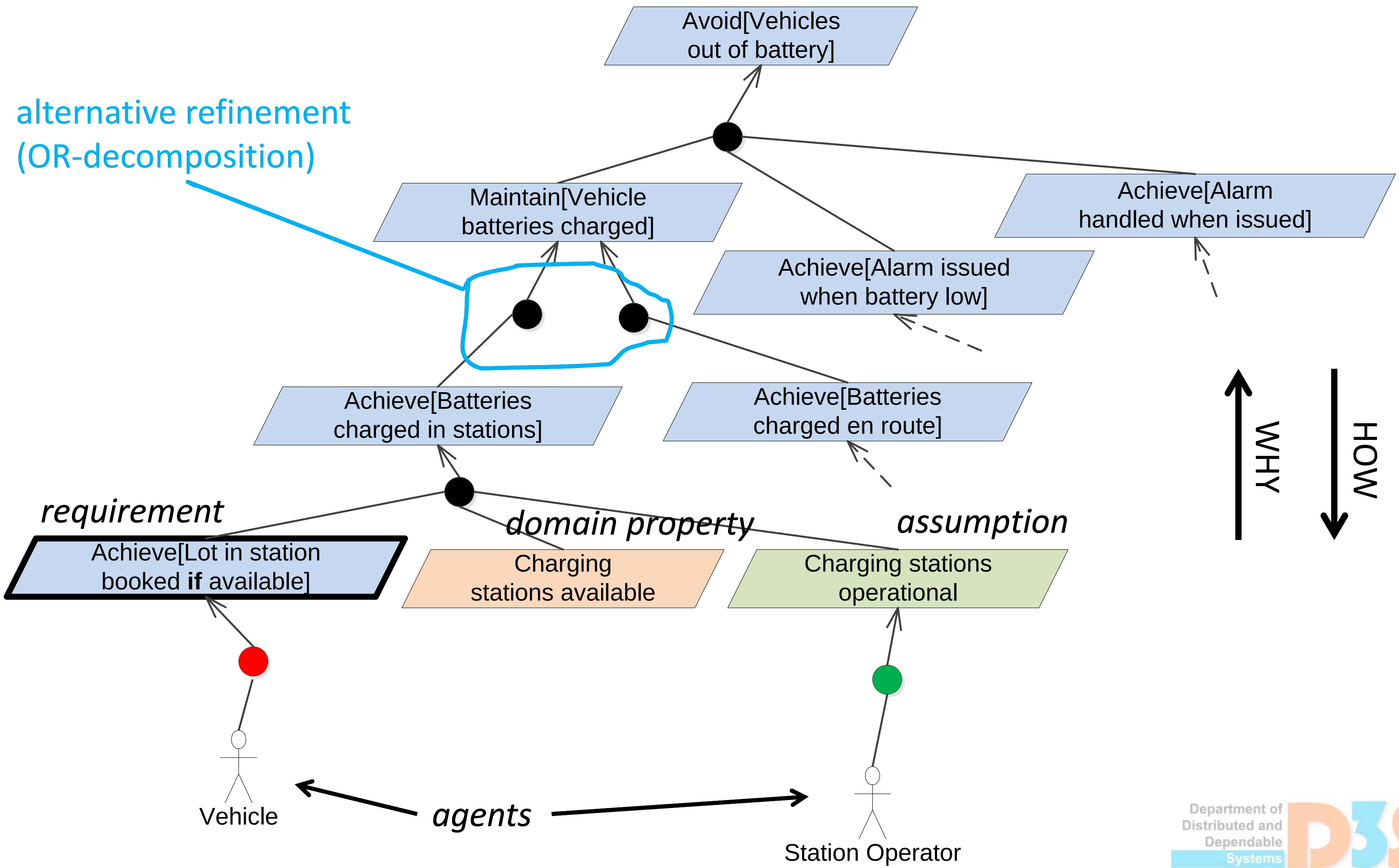
→ Alternative options



functional goals:
Underlying operation, feature, service, task

non-functional goals:
quality goals e.g. security, accuracy, architectural, development goals, etc.

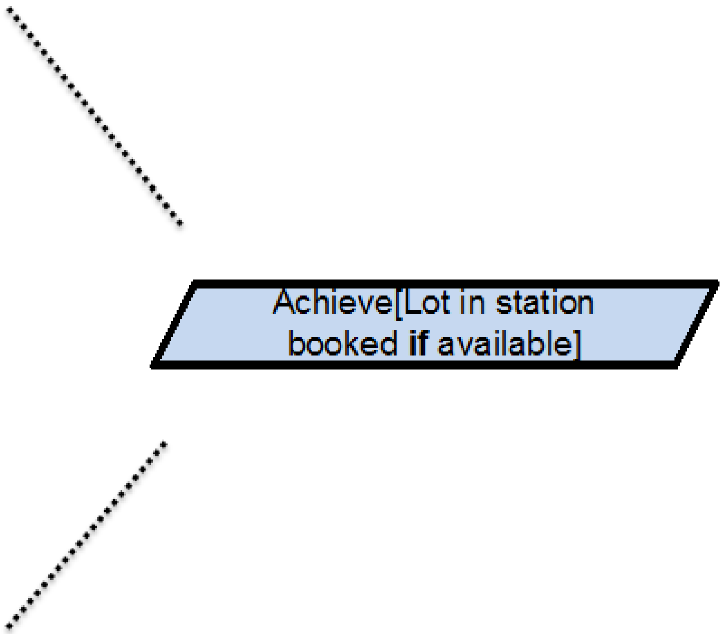
Goal model - II



Department of Distributed and Dependable Systems D3S

Formal Specification of Goals

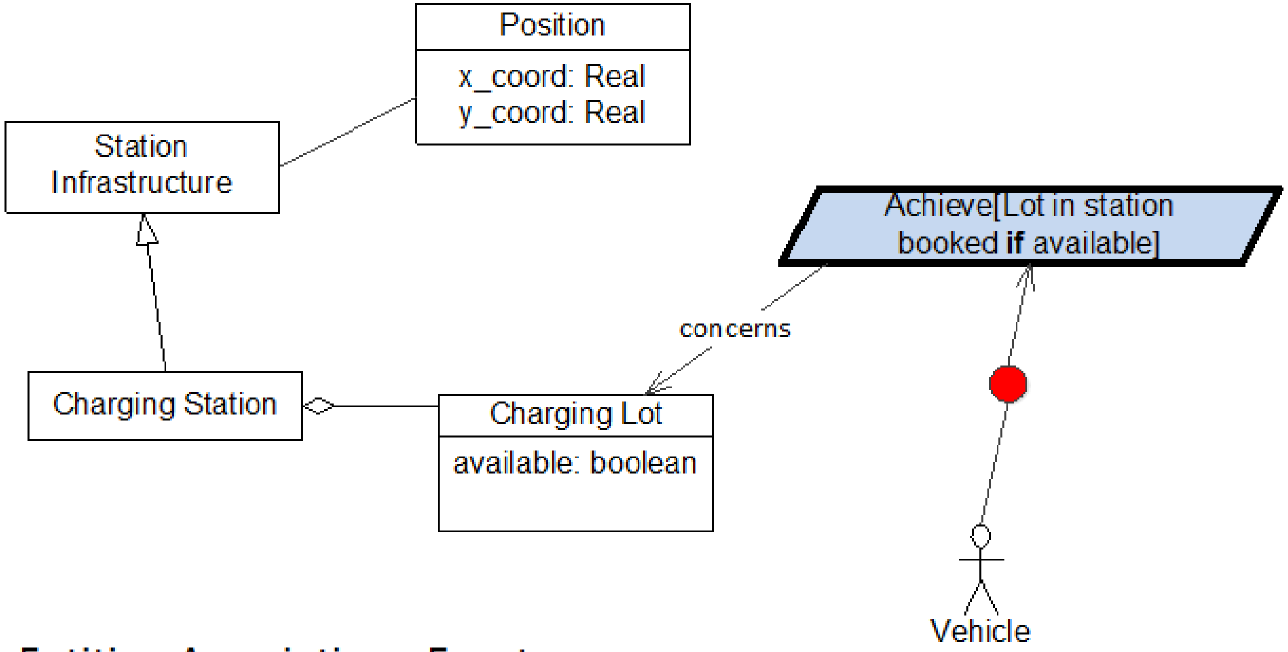
Name Lot in Station booked **if** available
Def If a place is available, then it must be booked by the vehicle in order to recharge
Type Achieve
Category Satisfaction
Source interview with VW
Priority Medium
FormalSpec
$\forall$ v: Vehicle, cl: ChargingLot:
LowBattery(v) $\wedge$ Available(cl) $\Rightarrow$
$\Diamond_{\leq T}$ Booked(v,cl)



Real-time linear temporal logic:

$\circ$ P, $\Diamond$ P, P U N, P W N, *and operators on past*

Object model



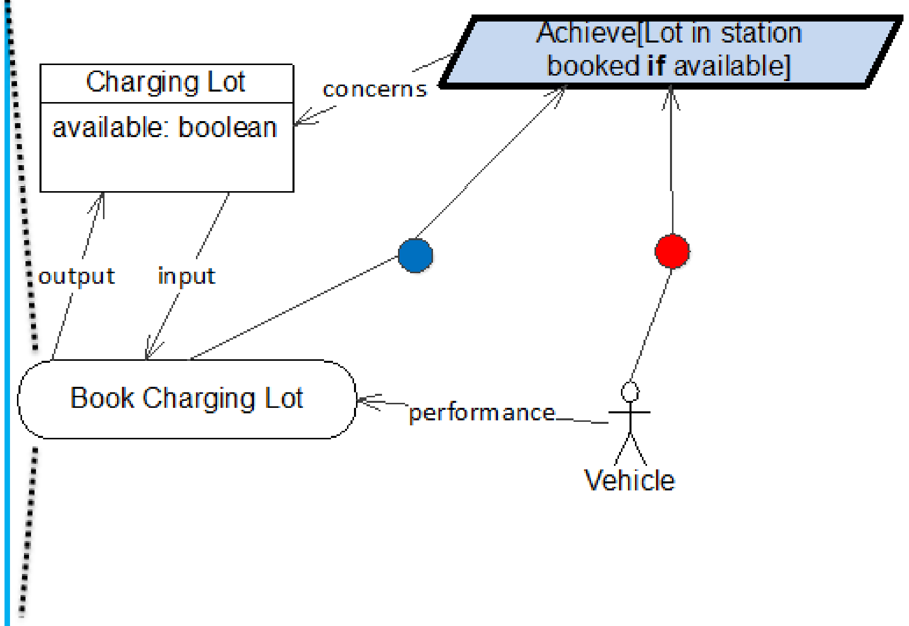
Objects: Entities, Associations, Events

Structure/Object model: UML class diagram notation

Only objects concerned in/referenced by goals are described

Operations model

Operation BookChargingLot
Def If a place is available, then it must be booked by the vehicle in order to recharge
Input cl: ChargingLot, v: Vehicle
Output cl
DomPre cl.available = true
DomPost cl.available = false
ReqPre ...
ReqTrig for LotInStationBookedIfAvailable: LowBattery(v) ∧Close(v,cl)
ReqPost ...



DomPre, *DomPost*: what the operation means in the domain

ReqPre, ReqTrig, ReqPost: additional strengthening to ensure the associated goal

What Goals provide in KAOS

sufficient completeness criterion:

A specification is complete **with respect to a set of goals** if all the goals can be proven to be achieved from the specification and the properties known about the domain.

pertinence criterion:

A requirement is pertinent **with respect to a set of goals** if its specification is used in the proof of at least one goal.

Goals refinement checking

A refinement of goal G into subgoals SG_1, …, SG_n is correct, when it is

- complete: $\{SG_1, \ldots, SG_{N,}$ DOM$\} \models G$
- consistent: $\{SG_1, \ldots, SG_{N,}$ DOM$\} \not\models$ false
- minimal: $\{SG_1, \ldots, SG_{j-1}, SG_{j+1}, \ldots, SG_n$, DOM$\} \not\models G$

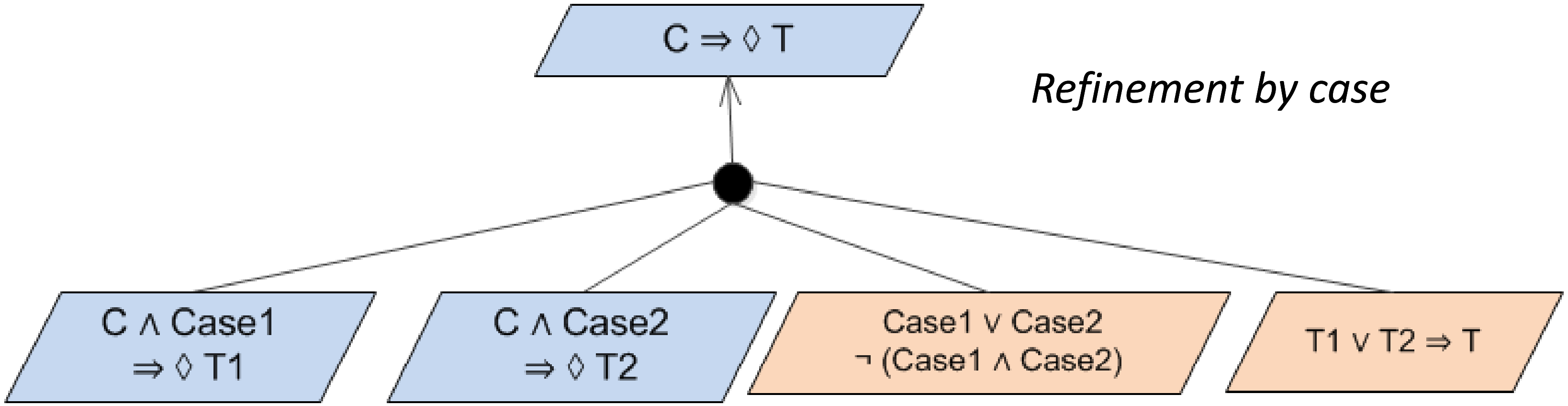
How to check goal refinements?

1. Use LTL theorem prover
   - heavyweight, nonconstructive
2. Use bounded SAT solver
   - input: $SG_1 \wedge \ldots \wedge SGn \wedge Dom \wedge \neg G$
   - incremental check/debug
3. Reuse refinement patterns

Refinement patterns - I

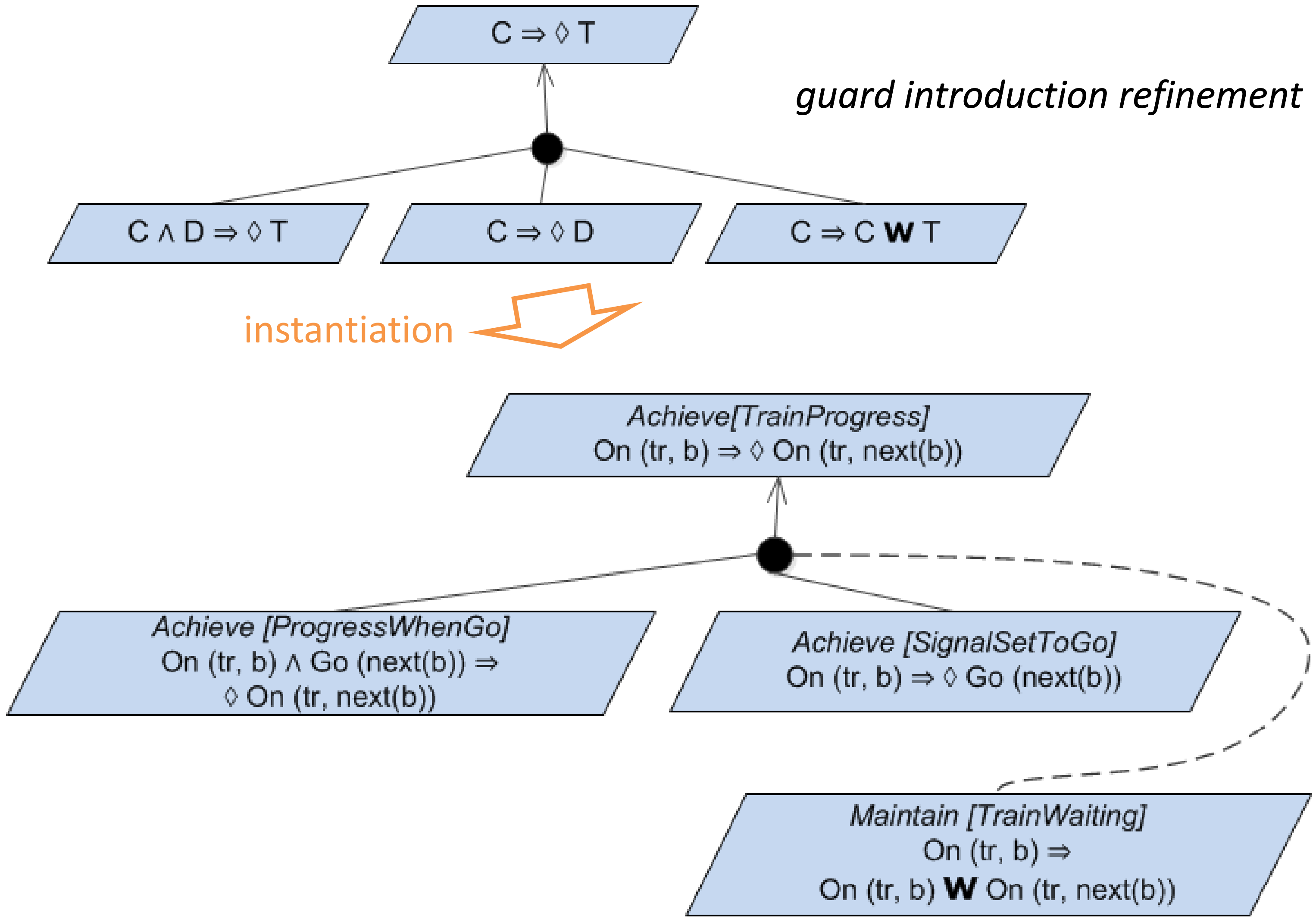
- Catalogue of patterns encoding *refinement tactics*
- Generic refinements proved formally, once for all
- Reuse through instantiation, in matching situation

Examples:



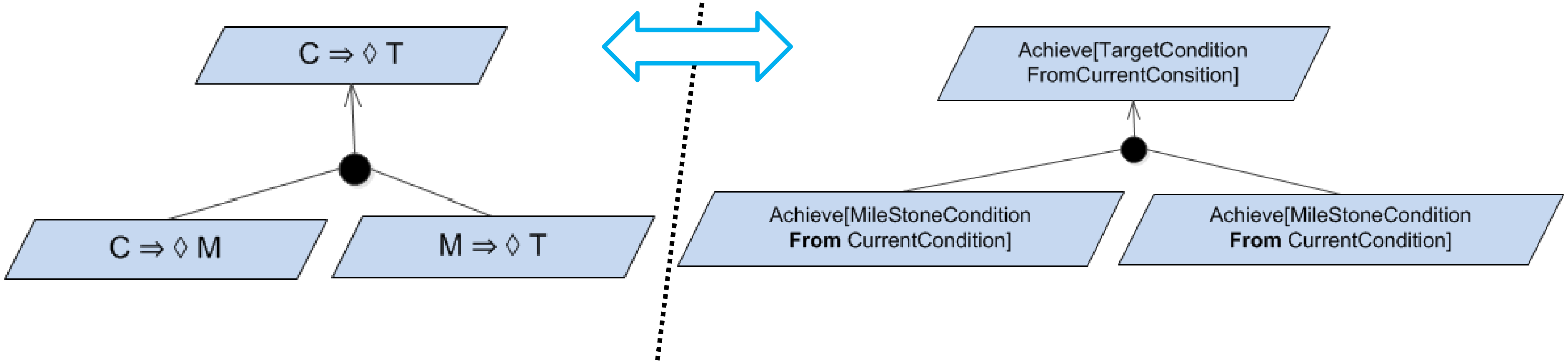
Refinement by case

Refinement patterns - II



Refinement patterns - III

milestone refinement



Formal pattern vs informal guideline

Apart from goal refinement, patterns can be applied:

- *Goal operationalization*
- *Obstacle analysis*

Department of Distributed and Dependable Systems D3S

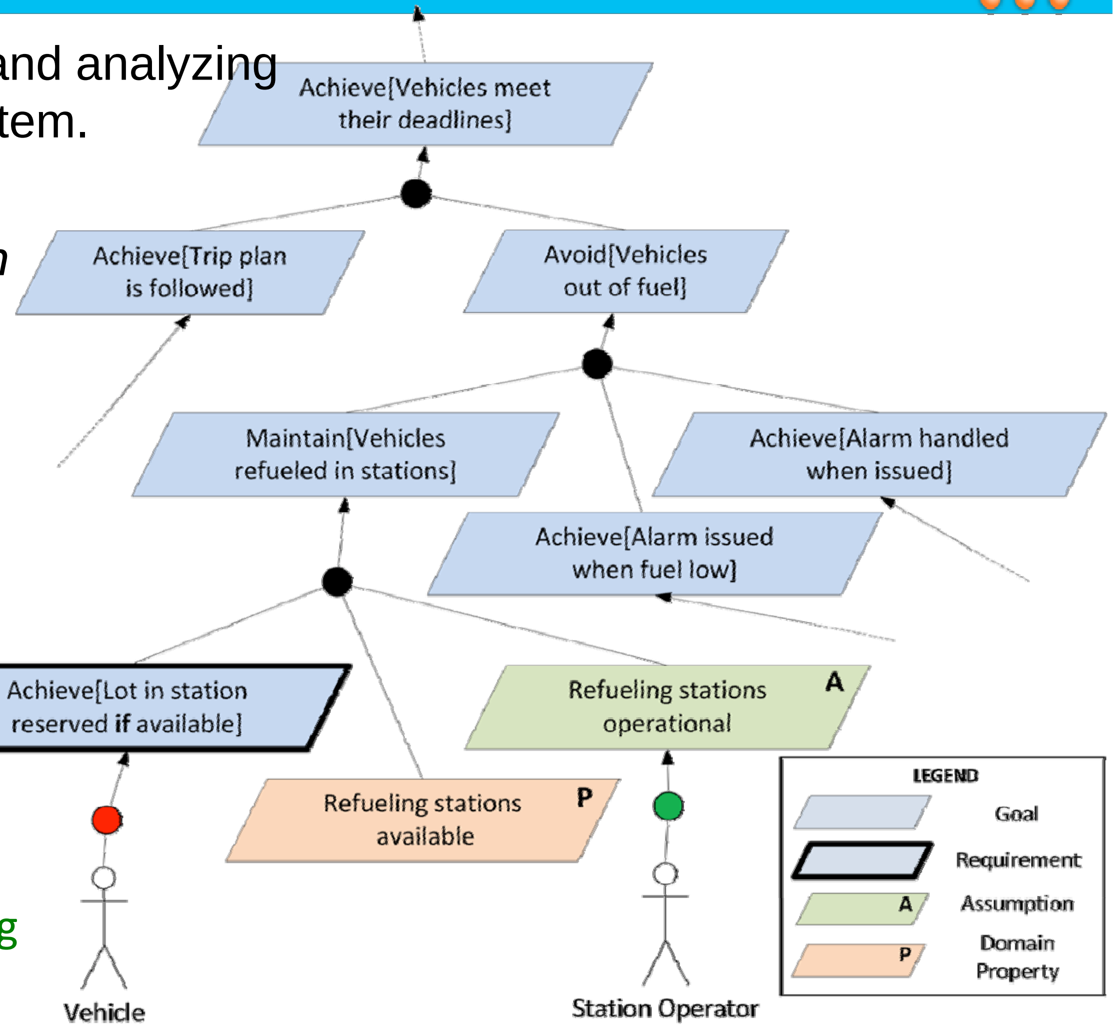
Requirements modeling – KAOS

Goal-oriented method for eliciting and analyzing the requirements of a software system.

- Goals have a prominent role
- Formal methods are used *when* and *where* needed

Goal model
Agent model
Object model
Operation model
Behavior model

} KAOS specification



Applicability in design of EBCS:

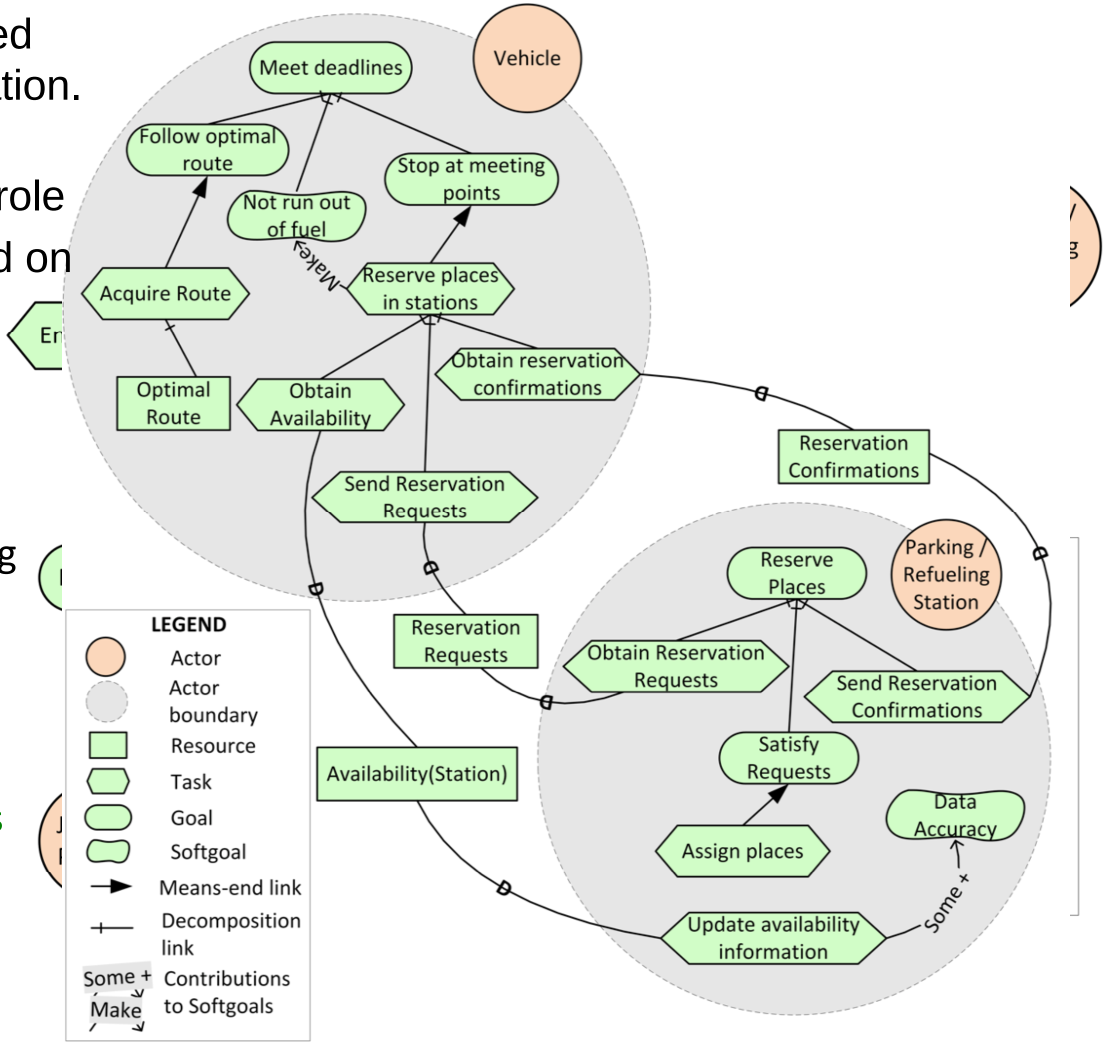
+ captures the (intended) system behavior at a high level
+ allows for automatic formal reasoning
– does not align requirements with architecture
– is intended for requirements analysis and documentation, not system design

Requirements modeling – Tropos

Methodology for building agent-oriented software systems that uses the i* notation.

- Agent and related notions (goals, plans, intentions) have prominent role
- Focus on early stages of SWD and on the organizational context

Goal models ⇨ Enhanced goal models ⇩ Actor-Agent mapping ⇦ BDI architecture (JACK, JADE)



Applicability in design of EBCS:

+ aligns the requirements phase with architecture and implementation phases
+ preserves a manageable set of concepts throughout the software development phases
– typically assumes static architecture (speaks about fixed instances)
– a bit ambiguous (goal or task?)

Detour: Resilient Systems

"A resilient control system is one that maintains state awareness and an accepted level of operational normalcy in response to disturbances, including threats of an unexpected and malicious nature"

[wikipedia]

Resilience

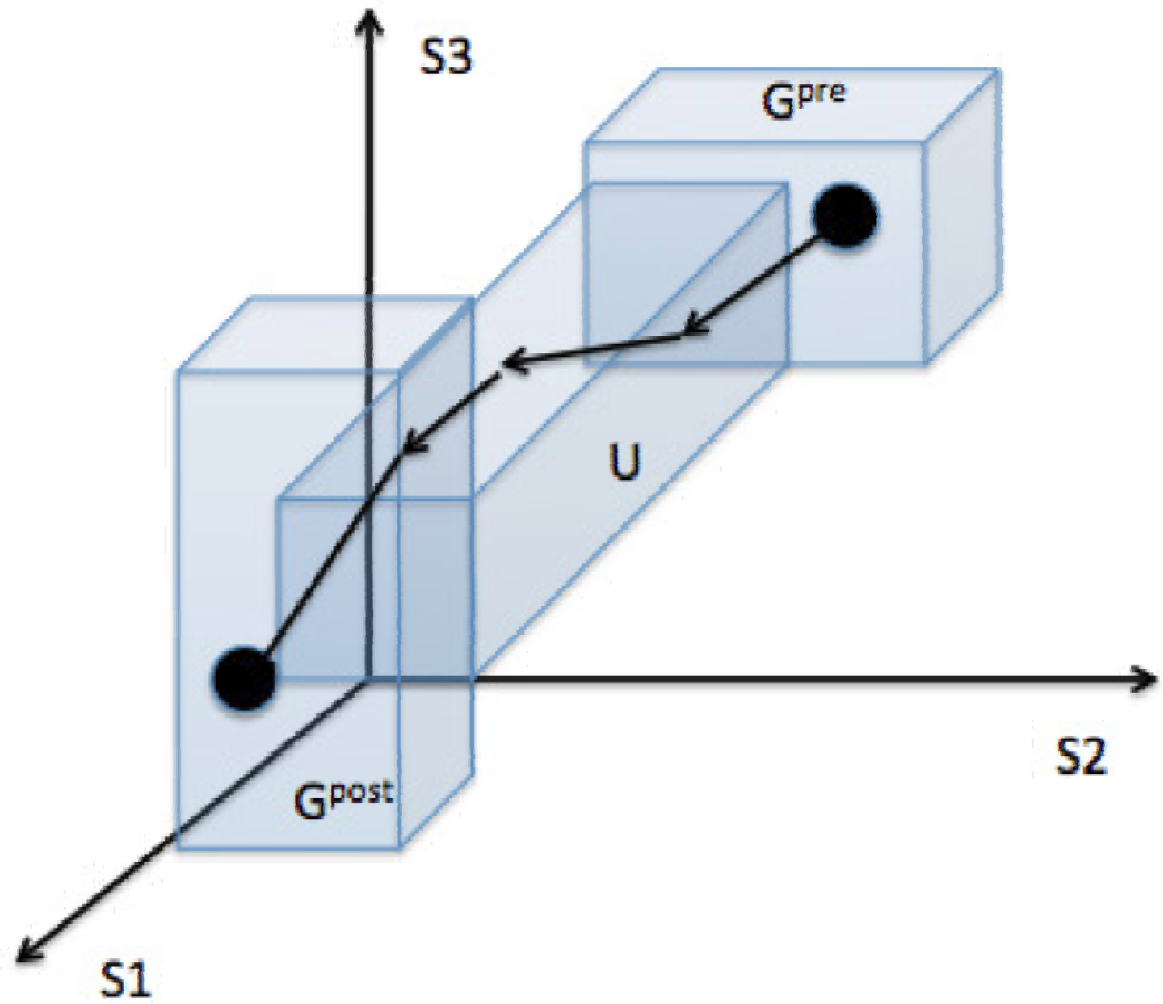
System adaptability

System evolvability

Impact on external environment

- Cooperative aspects

SOTA Model

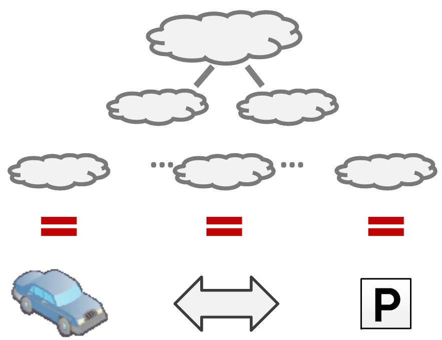


IRM – Invariant Refinement Method

Goals and high-level specification → IRM → Detailed system design

Systematic gradual refinement

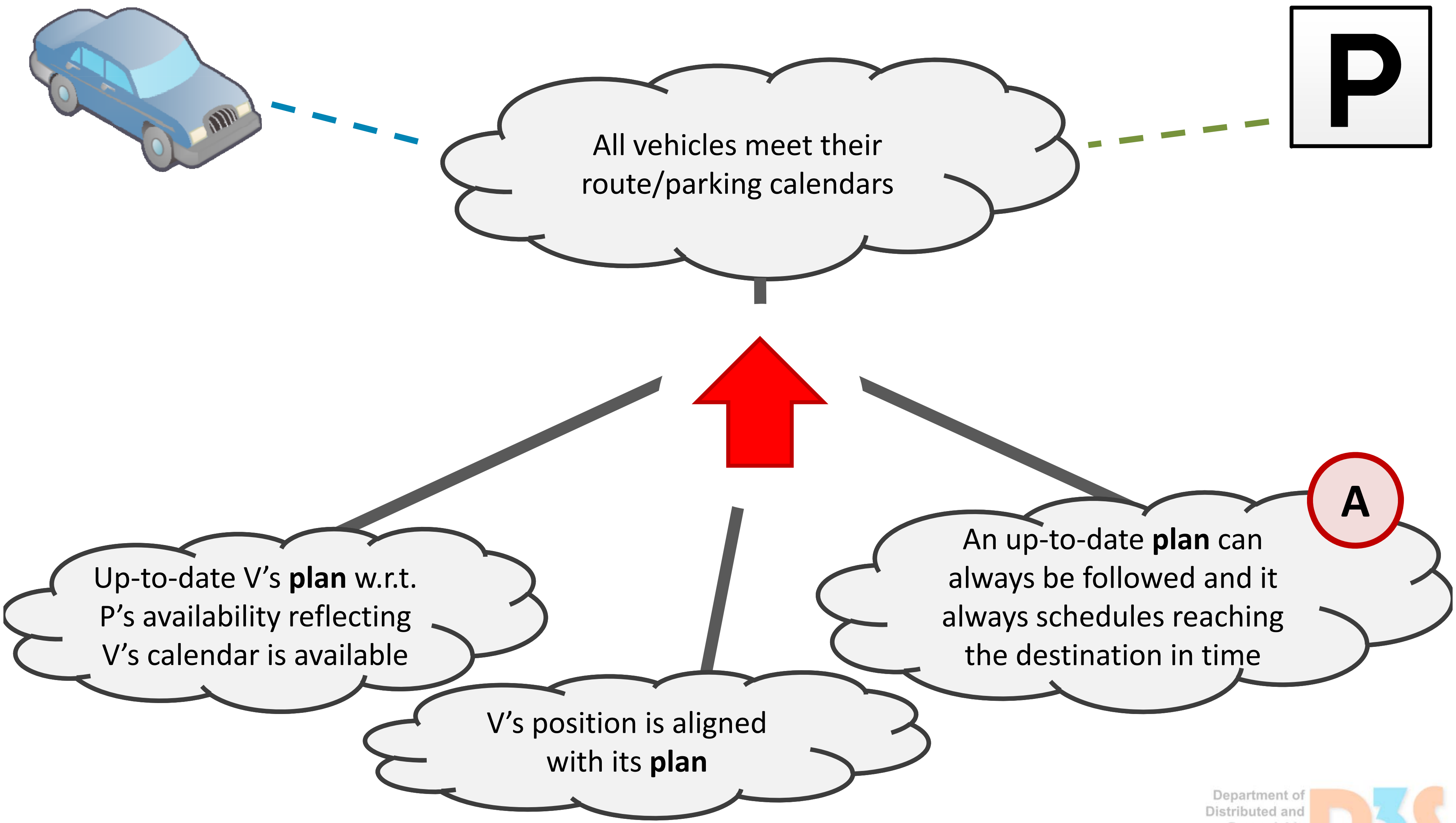
- Architecture design
- Conceptual framework & guidelines
- Borrows from goal-based requirements elaboration
  - KAOS, i*



Invariant (What is to be refined?)

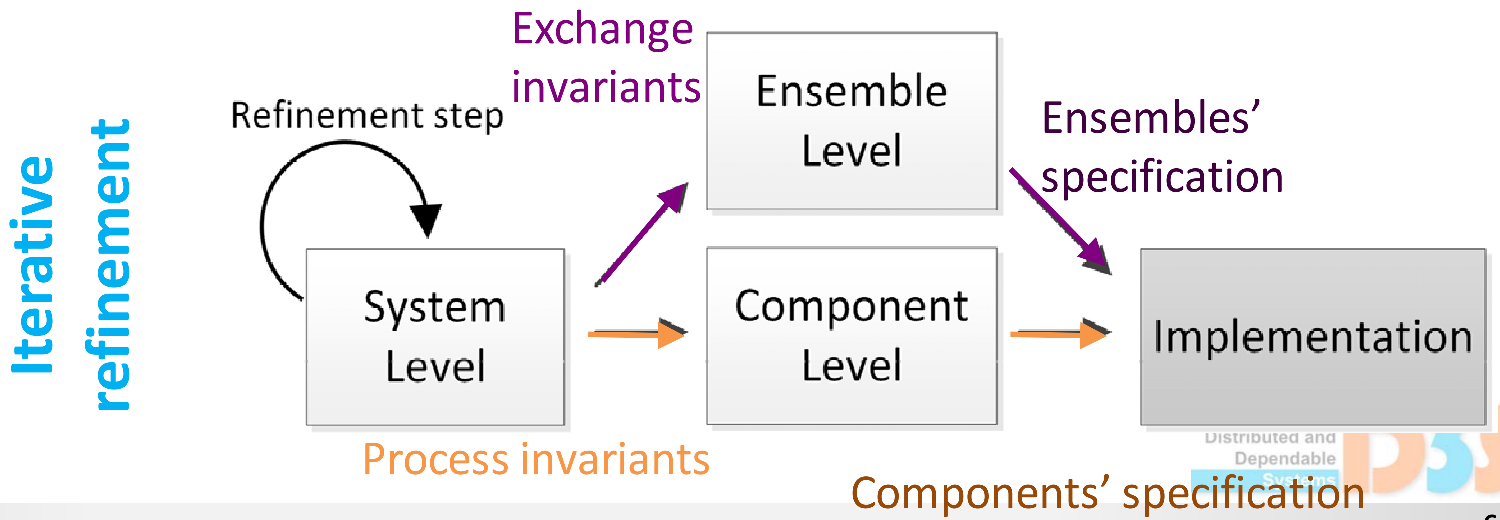
- Describes the **operational normalcy** of a (sub)system
  - i.e., the desired (global) state of the system that should be preserved as the knowledge valuation evolves in time
- Suitable for expressing **both goals and low-level concepts**
- Syntactically a condition on knowledge valuation of a set of components

Refinement

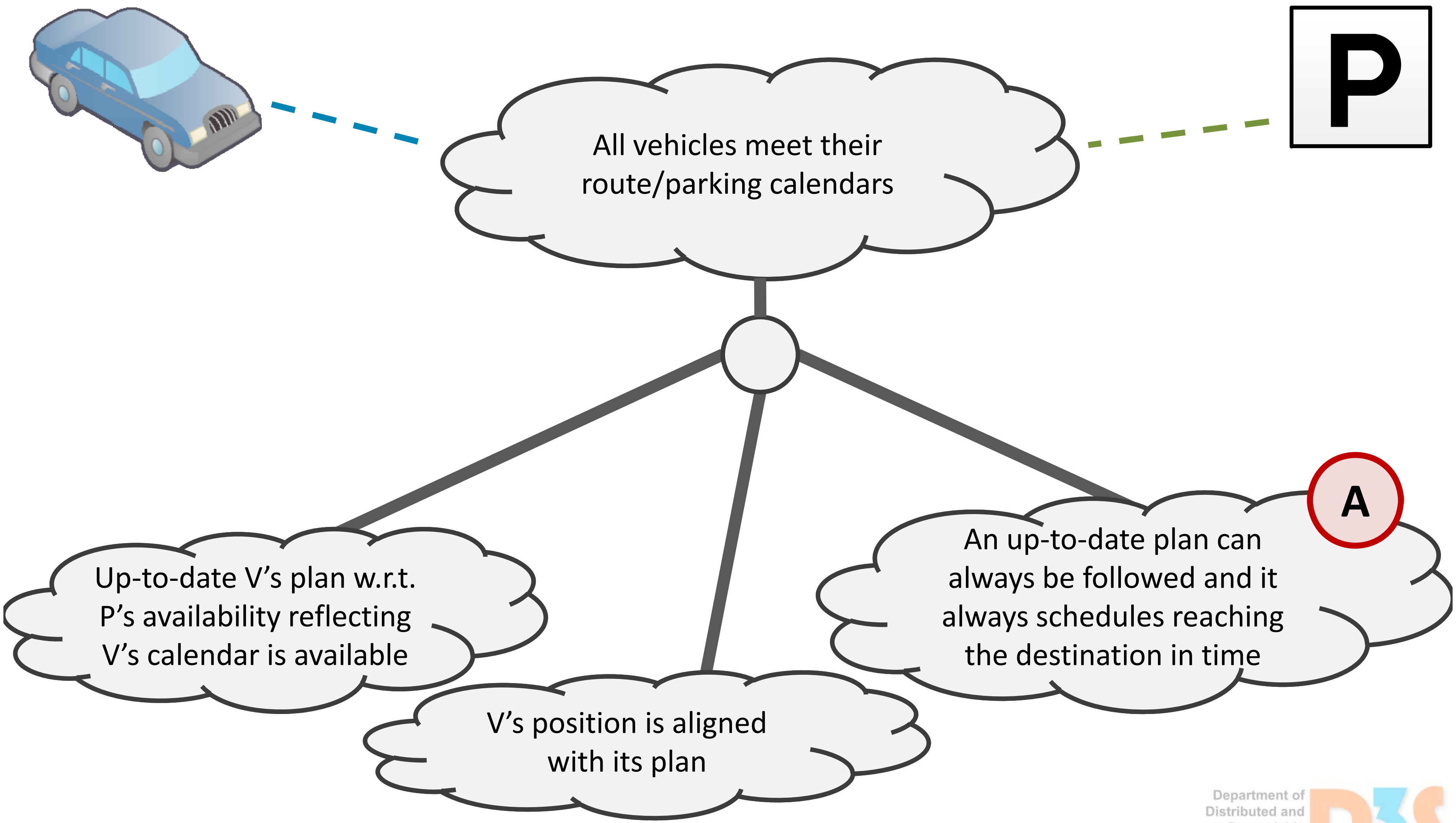

P
All vehicles meet their route/parking calendars
A
Up-to-date V's **plan** w.r.t. P's availability reflecting V's calendar is available
V's position is aligned with its **plan**
An up-to-date **plan** can always be followed and it always schedules reaching the destination in time

Leaves of Refinement (When to Stop?)

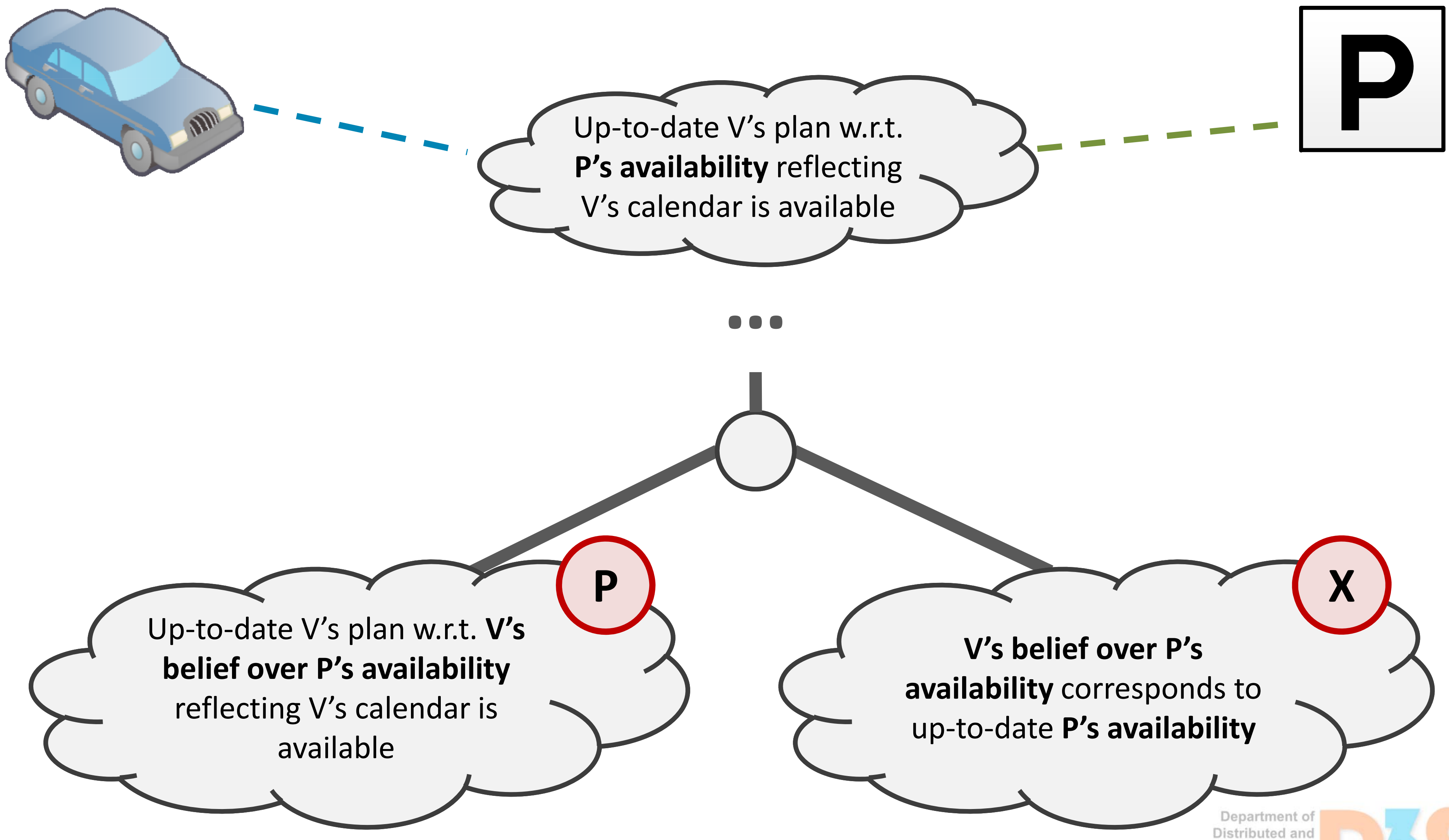
- Stop when an invariant is
  - Assumption
  - Can be “easily” mapped to a low-level execution concept
- **Process invariant**
  - Condition on knowledge of a single component
- **Exchange invariant**
  - A **belief** of a component vs. **knowledge** of another



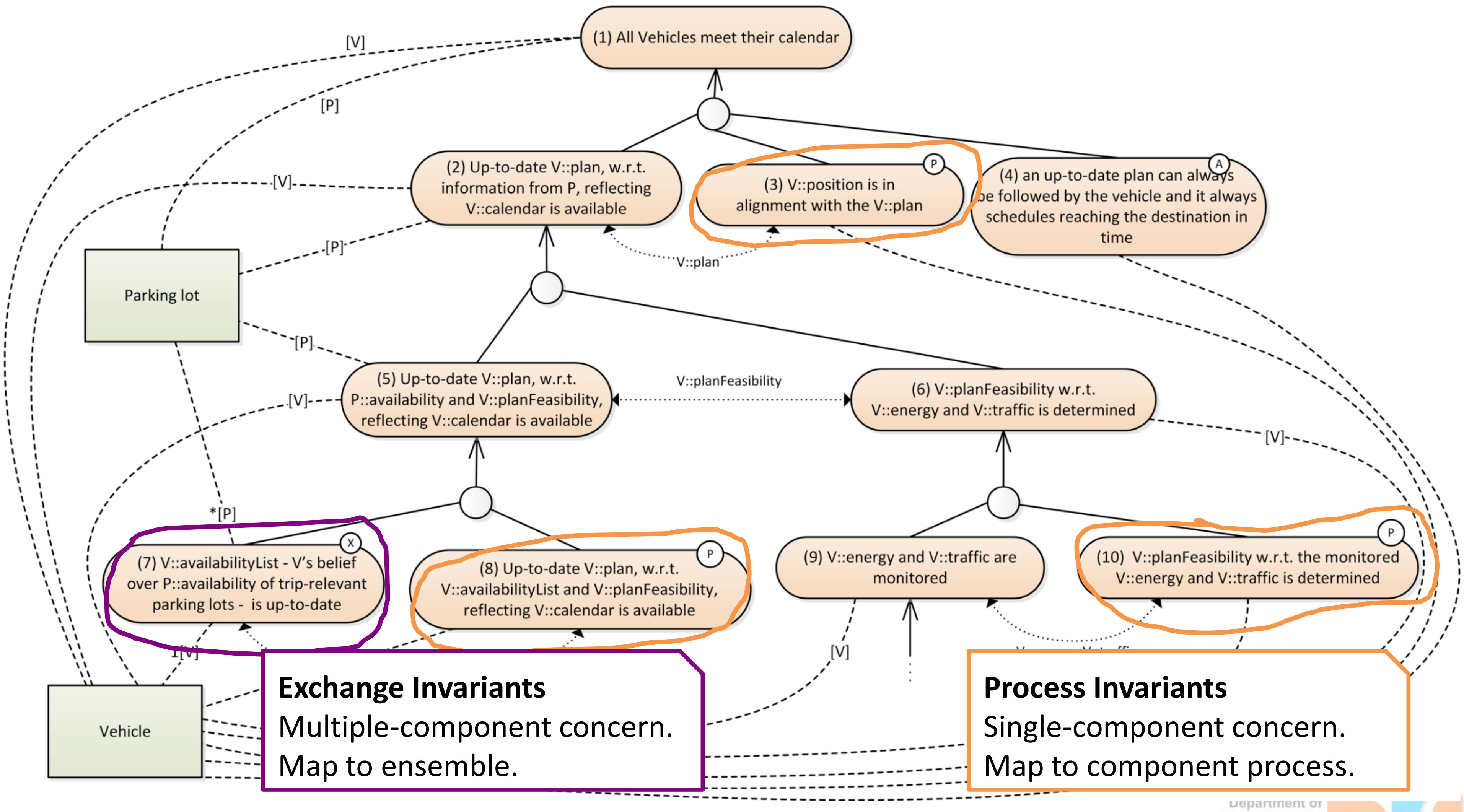
Leaves of Refinement



Leaves of Refinement



IRM refinement tree



Exchange Invariants
Multiple-component concern.
Map to ensemble.

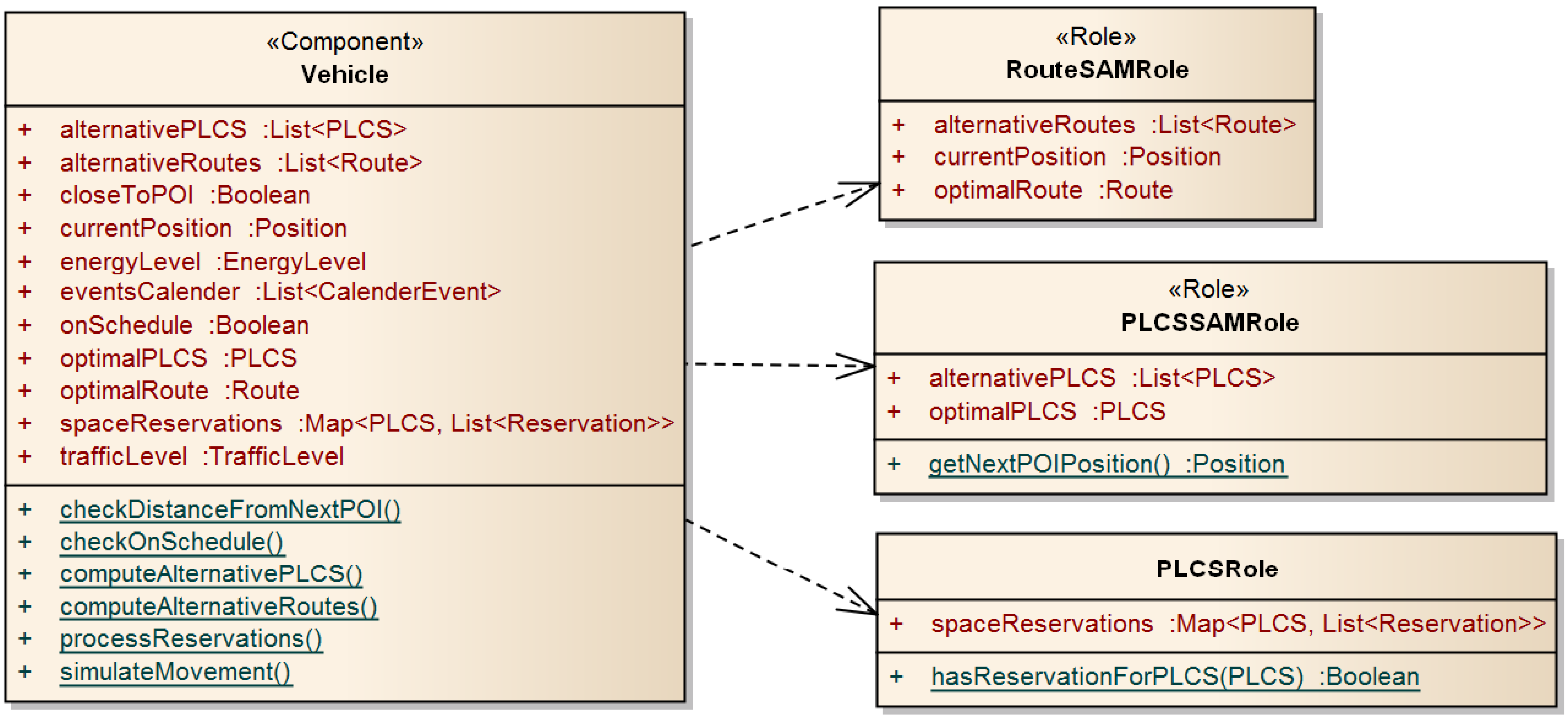
Process Invariants
Single-component concern.
Map to component process.

From Leaves to Detailed Design/Code

- Straightforward conversion
  - **Cyclic execution of processes/knowledge exchange maintains operational normalcy** (described by invariants)
- **Process**
  - all the inputs/outputs
  - post-condition/guarantee of the process
- **Ensemble**
  - the components/knowledge involved
  - the membership condition
  - post-condition/guarantee of the knowledge exchange

Design of components / ensembles

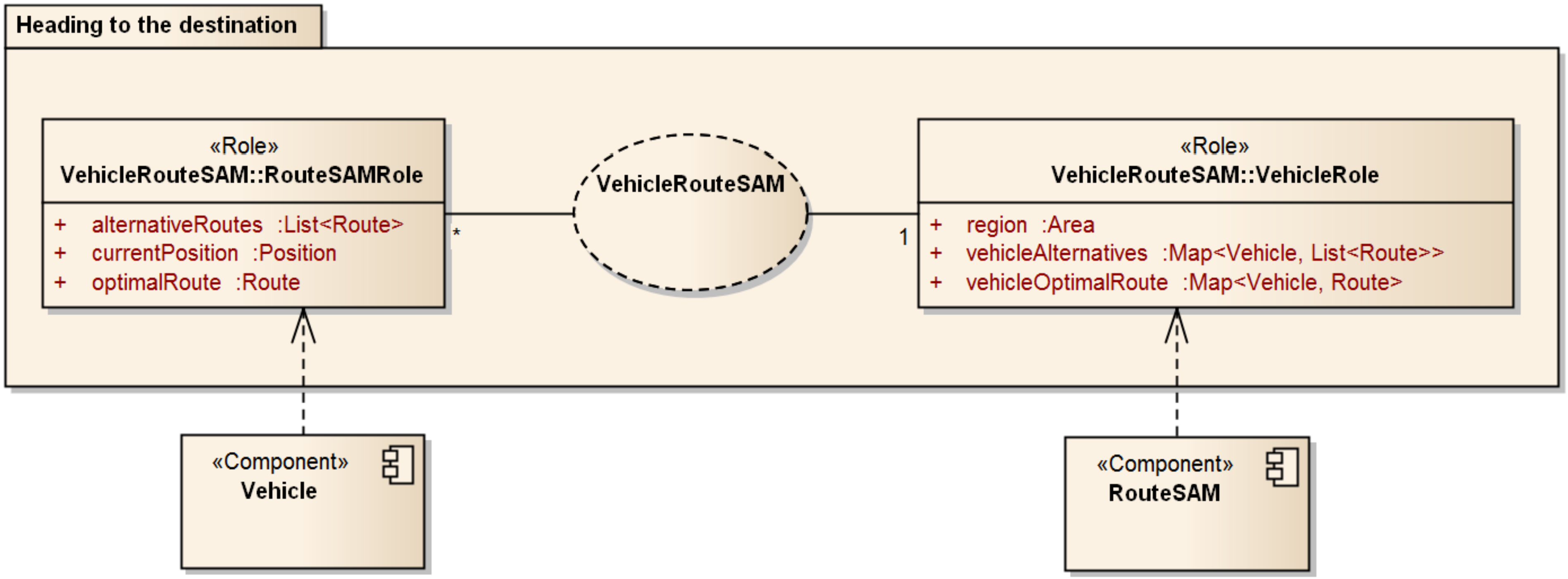
- Design of components and their roles (i.e. knowledge interfaces)



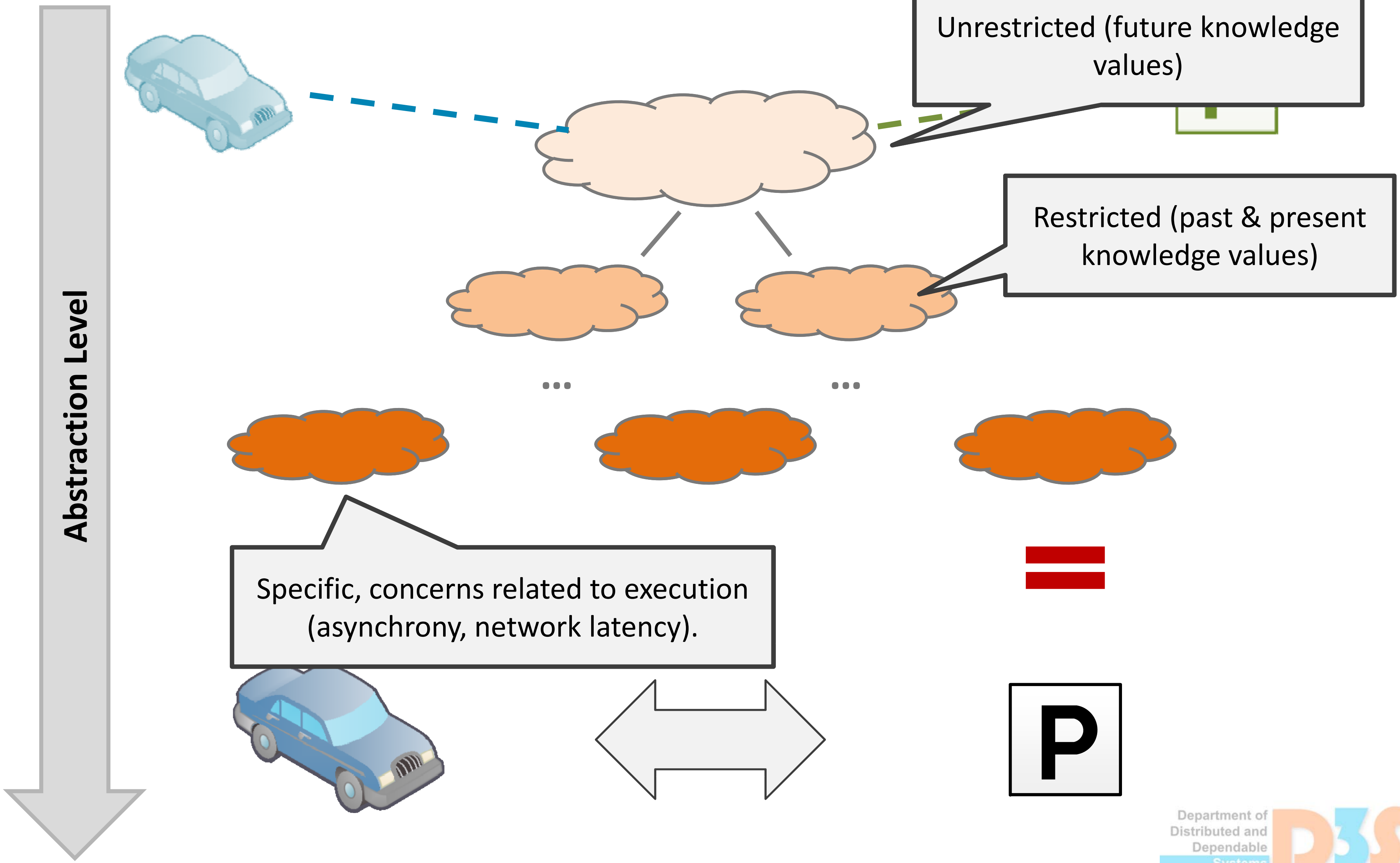
Department of Distributed and Dependable Systems

Design of components / ensembles

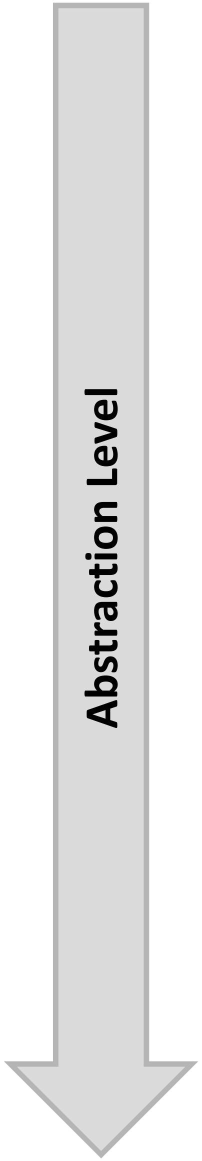
- Design of component interaction patterns (i.e. ensembles)
  - Captured as partial explicit architecture
  - Valid in a particular situation



Invariants on Different Levels of Abstraction



Invariant Patterns



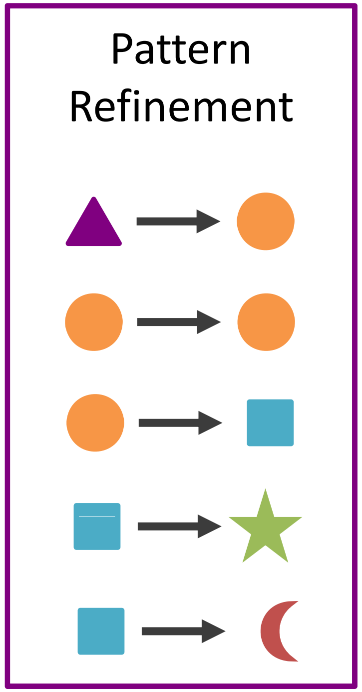
General
- Unrestricted (even future knowledge valuation)

Present-past
- SW system constraints (present/past knowledge valuation)

Activity
- Cyclic computational activity constraints
- Current outputs vs. current/past inputs
- Output changes only as a result of computation

Process
- Periodic execution constraints
- Output is produced once every period

Ensemble
- Periodic distributed execution constraints
- Output produced once every period from input outdated according to the network latency



Interesting Challenges (instead of conclusion)

- High-level of **dynamicity** and **open-endedness**
  - Can we reason about dynamically changing open-ended systems?
- Component **self-awareness** and adaptation based on **current situation**
  - Can we somehow formally reason about the situation and the awareness of it?
- Communication **latency** causes **uncertainty** (the system is almost constantly in de-synchronized state)
  - Can we somehow formally reason about system quality/reliability w.r.t. to communication difficulty?
- Proper level of abstraction for feasible testing and verification of **correctness** of components with **emergent behavior**
  - Can we somehow cope with emergent behavior?
- **Continuous integration** and **regular updates**
  - Can we somehow verify these systems incrementally?
- **Security aspects**