DEECo Component Model – Quick Start

A component model is a collection of rules according to which a componentized software system is structured. The component model defines and constrains how software components are implemented, how they are executed and how they are composed together. *DEECo* is a component model that is proposed for the development of *cyber-physical systems (CPS)*. CPS are networked systems that consist of multiple interacting elements: heterogeneous sensing & actuating devices, mobile devices, portable computers. CPS are typically very dynamic (their structure changes in time) and large (they consist of many devices and involve many users). The approach of DEECo was to study the problems in the development of CPS and propose a way CPS should be structured (architectural *rules* or *abstractions*) and operate (operational *semantics*) so that some of these problems are solved.

Components in DEECo are the basic units of development and deployment. They correspond to physical entities, e.g. the business code of a GPS sensor device would probably be encapsulated in a "GPSSensor" DEECo component. When a developer wants to use the GPS device in a CPS he/she builds, he/she has to first code the "GPSSensor" component. A DEECo component contains *processes* and *knowledge*, which roughly correspond to an object's *methods* and *state (fields)* in object-oriented languages, e.g. in Java or C#. However, there are specific rules that apply in DEECo components:

- **Processes** cannot be invoked by other processes or by user actions, but are executed periodically (e.g., a process "readCoordinates" that reads the position in the "GPSSensor" may run every 5 seconds). The middleware of DEECo takes care to execute each process at the right time slot. A component can of course have many processes. Regarding DEECo processes, the following rules apply:
  - When a process is invoked, it atomically reads its input from the component knowledge, executes the process body (the "business logic") and finally (again atomically) writes its output to the component's knowledge.
  - Every process executes independently from the others (in its own thread); this means that many processes can run concurrently.
  - The output for each process has to be unique. E.g. if "readCoordinates" writes its output to a knowledge field called "GPScoordinates", no other process is allowed to use "GPScoordinates" in its output (with the exception of processes writing to the same field but being mutually exclusive, see "condition" field later), although other processes are allowed to read the data. Since read/write of all inputs/outputs happens atomically (see subpoint#1), no race condition can occur in their concurrent execution.
- **Knowledge** is the internal state of a component (its "data"). The important rule here is that a component cannot access (read or write) **in any way** the data of another component. This is why DEECo components are considered "autonomous".
- DEECo processes (even in the same component) cannot call each other. The only way they can communicate is through writing and reading to the component's knowledge.

Ensembles are the only mechanism through which DEECo components can interact, so they are the only form of *component connectors* in DEECo. The following component interaction rules apply:

- The interaction (communication) among DEECo components is based **exclusively** on the *exchange of their knowledge*. Code running within a process of component A is *not allowed* to invoke a process/send a message to component B.
- Knowledge exchanging is a particular kind of business logic which is handled by a particular type of processes called ensembles. Ensembles have a single purpose: to copy knowledge from a component (called *coordinator*) to another component (called *member*) or vice verca. An ensemble specification has two parts:
  - "Membership" is a specification of what the knowledge of the member and coordinator is expected to be like, in order for the two of them to interact.
  - "Knowledge Exchange" is the specification of the mapping between the knowledge of the coordinator and the member.
- Ensembles are not specific to components; they are "interaction templates". They are applied whenever a component can play the role of coordinator or member (this is determined by the knowledge of every component).

- Ensembles, just like DEECo processes, are executed periodically by the DEECo middleware.

DEECo architectures: An Example

Let's see how to apply the above rules on a simplified version of a system of Electric Car Navigation and Parking (ECNP).

System story

Summary:

The main objective of this system is to allow e-cars to coordinate with parking stations and have an adequately up-to-date view of the availability of parking spaces at each time point. At the same time, e-cars should monitor their battery level and choose a different trip plan (e.g. which involves picking a parking place which is closer to the e-car) if the existing plan is not possible to follow any more.

Requirements:

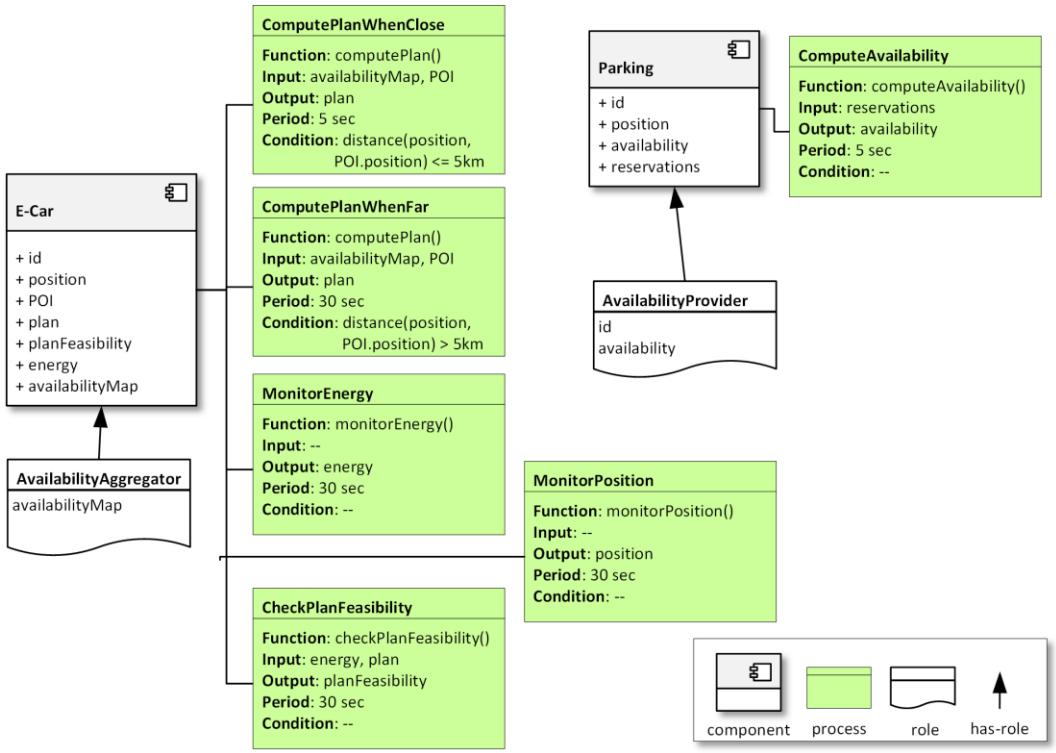
The **general** requirements for the ECNP are:

1. Every e-car has to arrive to its *place of interest* (POI) and park within a radius of 100 meters. In order to do that, every car needs to:
    a. Continuously monitor its energy level (battery);
    b. Continuously monitor its position;
    c. Continuously assess whether its energy level would be enough to complete the trip based on the distance left to cover;
    d. Have a plan to follow, which is based on its energy level and on the available parking slots in the parking places near the POI.
2. Every parking place has to continuously monitor its availability level (e.g. in terms of available parking slots per time slot).
3. The information regarding the availability of the parking slots has to be exchanged with the appropriate e-cars.

The **situation-specific** requirements of the ECNP are:

4. When an e-car is more than 5km far from the POI, it should update its plan *at least once per 60 seconds*.
5. When an e-car is equal to or less than 5km far from the POI, it should update its plan *at least every 10 seconds*.

Given the above requirements, the specification of DEECo components and ensembles (of the "DEECo architecture") can be the following:

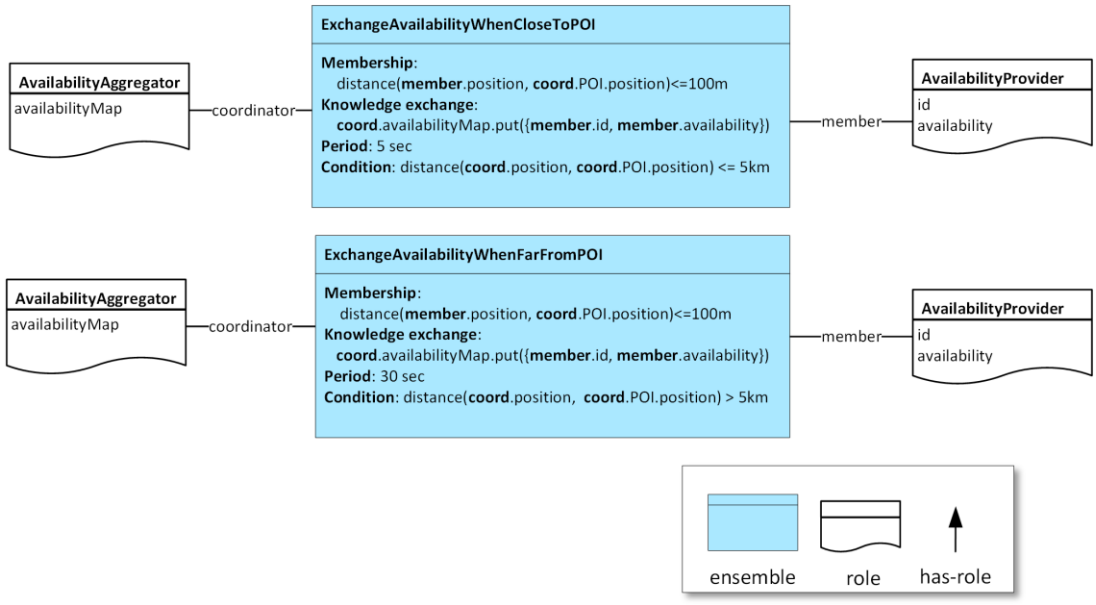


3.3

In the above diagram the two different DEECo components that comprise the architecture of the example are depicted together with their knowledge (lower compartment of each component), their processes and their roles. A role is just a collection of knowledge fields, which have to be included in a component's knowledge when connected with the "has-role" relation. For every process, the name of the function to be executed in the code, its input (set of knowledge fields), output (again set of knowledge fields), period (in msec or sec), and the condition upon which the process is active is depicted. The condition can be formulated in predicate logic with arithmetic: in the example, the predicate "distance" with two parameters is compared to number 50 (distance threshold in km).

Note that "ComputePlanWhenClose" and "ComputePlanWhenFar" differ only in the *Period* and the *Condition* attribute. A switching mechanism in DEECo middleware takes care to carry out the prescribed switching logic between two or more processes. The switching logic is captured in the *Condition* attribute as conditions that have to hold in order for the process to be active (to get periodically invoked by the middleware). In this example, the e-car's distance from its POI is checked and the appropriate process is activated ("ComputePlanWhenClose" if the distance is equal to/less than 5km, "ComputePlanWhenFar" otherwise). The same switching logic is prescribed in the ensembles below. Compared to membership, the condition is treated as the pre-condition for the evaluation of the ensemble (the evaluation comprises checking the membership and, if it holds, performing the knowledge exchange).

The periods of "ComputePlanWhenClose" and "ComputePlanWhenFar" are chosen so that their frequency is double than the required sampling frequency so that the timing requirements are met. The same reasoning was used for the calculation of the ensemble periods depicted below.



In the diagram above, the two ensembles of our example are depicted. They differ only in the *Period* and *Condition* attributes. They both allow knowledge exchange between two components, one taking the role of "AvailabilityAggregator" (coordinator), and the other taking the role of "AvailabilityProvider" (member). In our simplistic system, the only component that can take the first role is "E-Car"; the only component that can take the second role is "Parking". Both ensembles have the same *Membership* and *Mapping* attributes. This means that they both: first (i) check whether the condition specified by *Membership* holds (i.e. if the distance between the e-car and its destination is less than/equal to 100m), and then (ii) copy the "availability" knowledge field from "Parking" to a knowledge field (here, "availabilityMap") of "E-Car". The *Membership* attribute is again formulated in predicate logic with arithmetic, just like the *Condition* attribute.