DEECo

de3s.mff.cuni.cz

ASCENS Brainstorming

CHARLES UNIVERSITY IN PRAGUE
faculty of mathematics and physics
SCEL recall

• flat components
  ▪ interface, structured knowledge, processes, policies
  ▪ explicit inter-component dataflow in processes (get, put, qry)

• ensembles
  ▪ implicit connections (ensemble & membership predicates)
  ▪ a single ensemble per coordinator
  ▪ coordinator-centric communication

• asynchronous semantics
  ▪ processes
  ▪ blocking (get)
  ▪ interleaving (policies)
ASCENS Task Recap

- **WP1 - SCEL**
  - T1.4: Extensions for Implementation-level Conformance Checking and Performance Monitoring

- **WP4 - Adaptation + Self-Expression**
  - T4.4: Models and Techniques for Performance Monitoring and Prediction

- **WP5 - Correctness of SCs and SCEs**
  - T5.4: Verification of SC's implementation

- **WP6 - Tools**
Simplified development of distributed ubiquitous systems with emergent connections

- Problem:
  - the system's environment (i.e. physical environment and available services) may change at any point of time

- Contribution:
  - Suitable component paradigm
  - Methods for human-friendly specification of component's operation and inter-component contracts
  - Methods for exploring system's behavior depending on the environment
  - ...
Case Study

Robotic Playground Demo
DEECo - Main Concepts

- **interface**
  - semantics: view on knowledge

- **functions**
  - body: single threaded java code, nothing blocking

- **component**
  - knowledge (hierarchically structured - tree)
    - typed data
    - function variables
    - process definitions
  - semantics:
    - access to knowledge via a predefined interface(s) (matched by duck typing)
    - processes (triggered by knowledge modification or scheduled)
      - operate upon local knowledge visible at the level of their definition
      - process calls its assigned (“main”) function, assigns its actual parameters from its visible knowledge, on return it maps the out parameters back to the knowledge
      - not aware of being in an ensemble
  - synchronization: always multiple readers and single writer
  - function variables facilitate adaptation/ code mobility
DEECo - Main Concepts II

- composition
  - implicit flat via dynamic involvement in an ensemble
  - determined via interface matching and membership predicate of an ensemble

- ensemble
  - coordinator
  - member
  - semantics: (functions)
    - specification for the pair coordinator - member, instance for single coordinator, multiple members
    - membership predicate
    - mapping function (ensuring consistency among the coordinator’s and member’s knowledge visible via an interface)
    - update of knowledge is an atomic action
    - synchronization: always multiple readers and single writer

- priority of ensembles (partial order)
  - the ensembles bound by priority are mutually excluded in terms of knowledge mapping,
  - always the ensemble with the highest priority takes over the knowledge mapping
DEECO - Main Concepts III

- environment
  - supporting framework
  - watchdog for membership evaluation and ensuring knowledge mapping consistency
    - among coordinators and members
interface IProcess = {
    func: fun ();
    inMapping: list;
    outMapping: list;
    sched: SchedType;
}

interface IPosition = {
    x: int;
    y: int;
}

component Robot = {
    id: string;
    position: IPosition;
    path: list IPosition;
    nextPosition: IPosition;
    nextPositionAlongPath: IPosition;
    stepf: fun (in IPosition, in out list IPosition, in out IPosition) = RobotStepF;
    drivef: fun (in list IPosition, out IPosition) = RobotDriveF;
    processes = {...};
};
component Robot = {
    ...
    processes = {
        step: IProcess = {
            func = stepf;
            inMapping = [nextPosition, path];
            outMapping = [position, path];
            sched = PROCESS_PERIODIC;
            period = 100ms;
        },
        drive: IProcess = {
            func = drivef;
            inMapping = [path, nextPosition];
            outMapping = [nextPositionAlongPath];
            sched = PROCESS_TRIGGERED;
        }
    }
};
function RobotStepF(
    in nextPosition: IPosition, in out path : list IPosition,
    in out position: IPosition) = {
    if (nextPosition.equals(path.first()))
        path = path.dropFirst();
    position = nextPosition;
}

function RobotDriveF(
    in path: list IPosition, in nextPosition: IPosition,
    out newPositionAlongPath: IPosition) = {
    newPositionAlongPath = path.getFirst();
}
interface IRobot = {
    id: string;
    nextPosition: IPosition;
}

interface IRobotDrive = {
    id: string;
    nextPositionAlongPath: IPosition;
}

ensemble RobotDriveEnsemble {
    member: IRobot;
    coordinator: IRobotDrive;

    membership: fun (in robot: IRobot, in drive: IRobotDrive, out membership: boolean) = {
        membership = (robot.id == drive.id);
    };

    mapping: fun (in out robot: IRobot, in out drive: IRobotDrive) = {
        robot.nextPosition = drive.nextPositionAlongPath;
    };
}
Example - Crossing Component 1

```haskell
component Crossing = {
    position: IPosition;
    area: list IPosition;
    -- intended path of robot
    robots: map string -> {position: IPosition, path: list IPosition};
    -- next position for each robot suggested by the crossing
    nextPositions: map string -> IPosition;
    stepf: fun (...) = CrossingStepF;
    processes = {
        step = {
            func = stepf;
            inMapping = [robots, area];
            outMapping = [nextPositions];
            sched = PROCESS_PERIODIC;
            period = 50ms;
        }
    }
}

function CrossingStepF(
    in robots: map string -> {position: IPosition, path: list IPosition},
    in area: list IPosition;
    out nextPositions: map string -> IPosition) = {
    // guide robots through the crossing via setting their nextPositions
}
```
interface ICrossingRobot = {
    id: string;
    position: IPosition;
    nextPosition: IPosition;
    path: list IPosition;
}

interface ICrossing = {
    area: list IPosition;
    robots: map string -> {position: IPosition, path: list IPosition};
    nextPositions: map string -> IPosition;
}

ensemble CrossingEnsemble {
    member: ICrossingRobot;
    coordinator: ICrossing;
    membership: fun (in r: ICrossingRobot, in c: ICrossing, out m: boolean)={
        m = positionInArea(r.position, c.area)
    };

    mapping: fun (in out robot: ICrossingRobot, in out crossing: ICrossing) = {
        crossing.robors[robot.id] = {position: robot.position, path: robot.path};
        robot.nextPosition = crossing.nextPositions[robot.id]
    };
}
interface IGenRobot {
    id: string;
    crossingData: ICrossingData;
    f: fun (out IPosition, in ICrossingData)
}

interface IGenCrossing {
    crossingData: map string -> ICrossingData;
    f: map string -> (fun (out IPosition, in ICrossingData))
}

ensemble GenCrossingEnsemble {
    member: IGenRobot;
    coordinator: IGenCrossing;

    membership: fun(...) = {...}

    mapping: fun (in out robot: IGenRobot, in out crossing: IGenCrossing) = {
        robot.f = crossing.f[robot.id];
        robot.crossingData = coordinator.crossingData[robot.id];
    }
}
Example - Adaptable Robot 2

```javascript
component Robot {
    id: string;
    nextPosition: IPosition;
    crossingData: ICrossingData;
    f: fun (in ICrossingData, out IPosition);
    processes = {
        ... 
        genericRobotStep = {
            func = GenericRobotStepF;
            inMapping = [crossingData, f];
            outMapping = [nextPosition];
            sched = PROCESS_TRIGGERED;
        };
    };
}

function GenericRobotStepF(
    in crossingData: ICrossingData,
    in f: fun (in ICrossingData, out IPosition),
    out nextPosition: IPosition) = {
    nextPosition = f(crossingData);
}
```