Behavior models and verification

Lecture 13

Jan Kofroň, František Plášil
Efficient verification algorithms can extend applicability of formal methods

Many systems can be decomposed into parts

- verifying each property of each part separately
- if conjunction of parts’ properties implies satisfaction of overall specification, we are done
- the entire system never analyzed as the whole
Example

- Communication protocol: Sender, network, receiver
- Overall specification:
  - Data correctly transmitted from sender to receiver
- Partial specification checking
  - Data correctly sent from sender to network
  - Data correctly transmitted via network
  - Data correctly transmitted from network to receiver
- Illustrates: Verification of partial specifications typically much easier
  - sum of the state spaces much smaller than the state space of entire system (recall state space explosion)
Background: Assume-guarantee principle

- Verify each component separately
- Specify
  - **Assumptions** – requirements on behavior of environment
  - **Guarantees** – provisions offered to environment if assumptions are met
  - environment = the other components

- By combining assumptions and guarantees of particular parts, it is possible to infer correctness of the entire system
  - Full transition graph never constructed
Assume – guarantee formally

- Formula is triple $\langle g \rangle M \langle f \rangle$ where $g$ and $f$ are temporal formulae and $M$ is a program
  - If true -> whenever $M$ is part of a system satisfying $g$, the system also guarantees $f$

- Typically: proof of $\langle g \rangle M' \langle f \rangle$ and $\langle true \rangle M \langle g \rangle$
  $\rightarrow$ then we have $\langle true \rangle M \parallel M' \langle f \rangle$
  - $\parallel$ a composition operator

- Can be expressed as inference rule:

$$
\begin{align*}
\langle true \rangle M \langle g \rangle \\
\langle g \rangle M' \langle f \rangle \\
\hline
\langle true \rangle M \parallel M' \langle f \rangle
\end{align*}
$$
Formally

- Necessary to avoid circular dependencies:
  \[
  \begin{align*}
  &\langle f \rangle M \langle g \rangle \\
  &\langle g \rangle M' \langle f \rangle \\
  \hline
  M \parallel M' \models f \land g
  \end{align*}
  \]

- This is unsound!
- Should be avoided
Compositional reasoning - formally

- Composing n components

(Premise 1) \(<A_1> M_1 <P>\)
(Premise 2) \(<A_2> M_2 <A_1>\)
\[\vdots\]
(Premise n) \(<true> M_n <A_{n-1}>\)
\[\begin{array}{c}
<true> M_n \parallel M_{n-1} \parallel \cdots \parallel M_1 <P>
\end{array}\]
Each component specifies not only provided (implemented) interfaces,
- as objects do
- But also required interfaces

Syntactic (type) information
- may or may not consider interface/type inheritance

Semantic (behavior) specification
- verification techniques, usually model checking or pre-order/equivalence checking
  - (simulation, bisimulation,...)
In hierarchical component models (SOFA, Fractal)
- i.e., components are nested (primitive vs. composite)
- two directions of composability
  - horizontal
    - composition of subcomponents of a component
  - vertical (refinement)
    - composition of subcomponents yields a refinement of behavior and the parent component
- architecture description in the form of
  - IDL/ADL (interface/architecture description language)
  - (UML/EMF) model of application
  - usually includes many aspects (static – connections, dynamic – behavior, extra-functional – performance, ...)

SW components
Example of SOFA application – TIR browser
Example of SOFA application – TIR browser

Controller → TIRQuery → TIRQuery Body → Cache

Provisions → TIR
Example of SOFA application – TIR browser

Controller → TIRQuery → TIRQuery Body → Cache

Requirements
Type information

- Assumptions
  - requirements on types of required interfaces
- Provisions
  - types of provided interfaces
- Trivial to check composability
  - even if type hierarchy (inheritance) is considered
  - both horizontal and vertical directions straightforward
Can cover various aspects of the components
- functional – sequences of messages/calls
- extra-functional
  - timing
  - reliability
  - resource usage
  - security
  - ...

Composability verification based on the same principle
- component should provide at least as much (as good, fast, reliable, ...) as its counterpart requires
Functional aspects

- At provided side – specification of allowed sequences of method calls/messages
  - interfaces (horizontally) compatible if subset of provided functionality is required
- At required side similarly
  - in reaction to (part of) provided functionality
  - spontaneous – internal threads

- Can be extended to entire component
  - e.g. frame protocol in SOFA
  - specifies behavior of entire component
  - simplifies reuse and automatic selection of components
  - characterize component semantics at abstract level
Functional aspects – Example

- Behavior protocols
  - specifies behavior of components via expressions:
    - symbols
      - method names
      - ? (accept call) and ! Issue call
    - Operators
      - + (alternative) ; (sequence) * (repetition)
      - {x} (nested action x)
      - | (parallel composition)
        expressing independence/parallel actions
  - Example: Component for accessing data in a file
    (?f.open; (?f.read {!fs.read} + ?f.write{!fs.write})* ; ?f.close) | ?m.status*

- See: http://sofa.ow2.org
Example of SOFA application – TIR browser
Behavior protocols philosophy

- Behavior specification for
  - single component - boundary of component
    - FRAME protocol
  - horizontal composition:
    - parallel composition of the frame protocols of the components at the first level of nesting
    - ARCHITECTURE protocol
interface TIRAccessInterface {
    void init();
    string query(in string name);
    void finish();
};

frame TIRQueryBody {
    provides:
        TIRQueryInterface query;
    requires:
        TIRAccessInterface tir;
        CacheInterface cache;
    protocol:
        !tir.init;
        query.query{ !cache.get;
            (!tir.query+NULL) }* ;
        !tir.finish
};

frame TIRQuery {
    ...
    protocol ...
}

architecture CUNI TIRQuery implements TIRQuery {
    inst TIRQueryBody body;
    inst TIR tir;
    bind body:tir to tir:access;
    delegate query to body:query;
    subsume body:cache to cache;
    -- protocol: auto generated as parallel composition of the horizontally composed components
}
Protocol checks  Vertical & Horizontal Compliance

Controller → TIRQuery Body → Cache

TIRQuery

TIR

Jan Kofroň, František Plášil, Lecture 13
Protocol checks Vertical & Horizontal Compliance

Controller → TIRQuery → TIRQuery Body → Cache

TIR
Protocol checks Vertical & Horizontal Compliance

Controller → TIRQuery → TIRQueryBody

TIRQueryBody

Frame protocol

TIR

Cache
Protocol checks Vertical & Horizontal Compliance

Controller → TIRQuery → TIRQueryBody → TIR → TIRFrame protocol

→ Cache

TIRQuery

TIRQueryBody

Frame protocol

TIR

Frame protocol
Protocol checks  Vertical & Horizontal Compliance

Horizontal composition:
 □ ▼ □
Protocol checks Vertical & Horizontal Compliance

Controller → TIRQuery
TIRQuery → TIRQueryBody
TIRQueryBody → TIR
TIR → TIRFrame protocol
TIRFrame protocol → Cache
Protocol checks  Vertical & Horizontal Compliance

Vertical Composition:
(□ ▼ □) refines □
Protocol checks Vertical & Horizontal Compliance

Vertical Composition: $(\square \downarrow \square)$ refines $\square$

Horizontal composition: $\square \downarrow \square \downarrow \square$
Applying protocols: Checks options

Design

- Adjacent levels of abstraction
  - *Do the cooperating children do what the parent expects?*
  - Vertical Compliance
    - refinement check

- The same level of abstraction
  - *Do the children cooperate with no conflict?*
  - Horizontal compliance
    - *Checking errors in parallel composition*
Applying protocols: Checks options

- Design
  - Adjacent levels of abstraction
    - *Do the cooperating children do what the parent expects?*
  - Vertical Compliance
    - refinement check
  - The same level of abstraction
    - *Do the children cooperate with no conflict?*
    - Horizontal compliance
      - *Checking errors in parallel composition*

- Runtime
  - Checking real behavior against frame protocols
    - *Runtime checking by monitoring*
      - Important for primitive components
Features:

- **Parallelism**
  - $M \parallel (a.b.\varepsilon + b.c.\varepsilon)$
  - Actions are atomic (interleaving ... )!

- **Communication (synchronization)**
  - $\gamma(b, c) = d$
  - particular pair of actions becomes another action

- **Abstraction**
  - $\tau_{\{i_1,i_2\}}(a.i_1.i_2.b.\varepsilon)$
  - listed actions become internal

The resulting theory

- **Algebra of Communicating Processes (ACP)**
Parallel composition

- Recall process algebra
  - Parallel composition
    - Complementary actions synchronize (yielding $\tau$ action)
- Parallel composition in behavior protocols
  - ▼ Consent operator
    - Via parallel composition synchronizes complementary actions
      - roughly $?x$ with $!x$
    - The parallel composition captures communication errors
      - Bad Activity
      - No Activity
      - Infinite Activity
        - no $?$ for $!$
        - deadlock
        - livelock
Bad activity

\[ \text{Prot}_X = !n \; ; \; ( !o + !n) \]

\[ \text{Prot}_Y = ?n \; ; \; ?o \]
Bad activity

\[ \text{Prot}_X = !n \ ; \ ( !o + !n) \]

\[ \text{Prot}_Y = ?n \ ; \ ?o \]

\[ !n = !n\uparrow \ ; \ ?n\downarrow \]

\[ \text{Prot}_X = !n\uparrow \ ; \ ?n\downarrow \ ; \ (( !o\uparrow \ ; \ ?o\downarrow ) + ( !n\uparrow \ ; \ ?n\downarrow )) \]

\[ \text{Prot}_Y = ?n\uparrow \ ; \ !n\downarrow \ ; \ ?o\uparrow \ ; \ !o\downarrow \]
Bad activity

\[ \text{Prot}_X = \neg n \uparrow ; ?n \downarrow ; (( \neg o \uparrow ; ?o \downarrow ) + ( \neg n \uparrow ; ?n \downarrow )) \]

\[ \text{Prot}_Y = ?n \uparrow ; !n \downarrow ; ?o \uparrow ; !o \downarrow \]

\[ \text{Prot}_X \bigtriangledown \{ n \uparrow, n \downarrow, !o \uparrow, o \downarrow \} \]

\[ \text{Prot}_Y = \{ < \tau n \uparrow ; \tau n \downarrow ; \tau o \uparrow ; \tau o \downarrow >, < \tau n \uparrow ; \tau n \downarrow ; \times n \uparrow > \} \]
Summary of compositional reasoning

• Recall

(Premise 1) \( <A_1> M_1 <P> \)
(Premise 2) \( <A_2> M_2 <A_1> \)
\[ \vdots \]
(Premise \( n \)) \( <\text{true}> M_n <A_{n-1}> \)
\( <\text{true}> M_n \parallel M_{n-1} \parallel \vdots \parallel M_1 <P> \)

• Compositional reasoning for behavior protocols

1) For each a primitive component \( C_{\text{prim}} \) check that

\( <\text{true}> C_{\text{prim}} <\text{guarantees compliance with its frame protocol}> \)

2) For all levels of component nesting (bottom up):

2.1) Check horizontal compliance in architecture protocol

2.2) Check vertical compliance

\[ \text{architecture protocol refines frame protocol of} \ C_{\text{par}} <\text{Cpar behaves as frame protocol specifies}> \]
Extra-functional aspects

- Similar in many views to functional case
  - “provide at least what is required”
- Granularity from method/service/function to component/set of components
- Can cover various dimensions
  - performance – execution time (average/worst)
  - Resource consumption – CPU time/memory space
  - reliability – likelihood of failure, time to recovery
  - specification of HW for deployment
  - ...

Extra-functional aspects – Example

- Palladio Component Model (PCM)
  - required resources
  - performance
  - deployment
  - usage profile

- Many types of analysis
  - worst/mean execution time for composed services
  - cumulated resource requirements
  - possible deployment targets for given usage profile
  - reliability analysis

- See: http://www.palladio-simulator.com/
Code level
Code level

- Types – usually checked by compiler
  - no additional effort required

- Semantics – code contracts
  - at level of functions/methods
  - assumptions – preconditions
    - on environment
  - guarantees – postconditions
    - for environment
  - usually also invariants
    - helps with verification
    - loop invariants
• Verification is compositional (*modular*)
  ▪ each function is verified separately – whether the code really guarantees postcondition once precondition is satisfied at function entry
  ▪ if function is called from within other function, its contract is assumed (precondition is checked, postcondition is assumed)
void add(int index, Object obj) { ... }
public int size() { ... }

• “Value of the index parameter has to be greater than or equal to zero. Successful call of add increases the size of the array by one.”

• Formally:

```java
def public void add(int index, Object obj)
    requires index >= 0;
    ensures size = old(size) + 1;
    {
        ... 
    }
```
public class ArrayList {
    public void add(int index, Object obj) { ... }
    public int size() { ... }
}

• “Value of the index parameter has to be greater than or equal to zero. Successful call of add increases the size of the array by one.”

• Formally:
  public void add(int index, Object obj)
  \text{requires index} >= 0;
  \text{ensures size} = \text{old(size)} + 1;
  { ... }
public class ArrayList {
    public void add(int index, Object obj) {
        ...
    }
    public int size() {
        ...
    }
}

• “Value of the index parameter has to be greater than or equal to zero. Successful call of `add` increases the size of the array by one.”

• Formally:
  
  public void add(int index, Object obj)
  
  requires index >= 0;
  
  ensures size = old(size) + 1;
  
  { ...
  }
Contract specification languages
- Spec#, JML, Code Contracts, ...

There are tools to verify contracts
- model checkers, SAT/SMT solvers, theorem provers

NSWI132 – Program analysis and code verification
It is not easy to specify the contracts

- preconditions
  - too weak to guarantee postconditions
  - too strong to be satisfied by caller

- postconditions
  - too strong to be proven
  - too weak to “satisfy” caller
    - illustrated by two next slides

One has to know and tune up ...
int[N] field;
int swapMin(int from)
{
    swaps the min value beyond from with the one at from and returns its index
}

int main()
{
    // really sorted:
    ensures (forall int i : 0<i<N-2 : field[i] <= field[i+1]);
    // the original values kept
    ensures (forall int i : 0<i<N-1 && exists int j : old(field[j]) == field[i]);
    
    for (int i = 0; i < N; i++)
    {
        swapMin(i);
    }
}
int[N] field;
int swapMin(int from)
{
    \textit{swaps the min value beyond from with the one at from and returns its index}
}

int main()
{
    // really sorted:
    \texttt{ensures} (\forall \text{int } i: 0 < i < N-2 : \text{field}[i] \leq \text{field}[i+1]);
    // the original values kept
    \texttt{ensures} (\forall \text{int } i: 0 < i < N-1 \&\& \exists \text{int } j : \text{old(field[j])} == \text{field}[i]);
    
    \texttt{for} (\text{int } i = 0; i < N; i++)
    {
        \text{swapMin}(i);
    }
}

We need some guarantees from swapMin to prove this
int[N] field;
int swapMin(int from)
    ensures ((field[return] == old(field[from])) && field[from] == old(field[return]));
{
    swaps the min value beyond from with the one at from and returns its index
}

int main()
    // sorted
    ensures (forall int i : 0<i<N-2 : field[i] <= field[i+1]);
    // the original values
    ensures (forall int i : 0<i<N-1 && exists int j : old(field[j]) == field[i]);
{
    for (int i = 0; i < N; i++)
        swapMin(i);
}
int[N] field;  

Holds, but too weak to ensure postconditions in main() !

int swapMin(int from)

ensures ((field[return] == old(field[from]) && field[from] == old(field[return])));
{
  swaps the min value beyond from with the one at from and returns its index
}

int main()

// sorted
ensures (forall int i : 0<i<N-2 : field[i] <= field[i+1]);

// the original values
ensures (forall int i : 0<i<N-1 && exists int j : old(field[j]) == field[i]);
{
  for (int i = 0; i < N; i++)
    swapMin(i);
}
int[N] field;
int swapMin(int from)
    ensures (forall int i<from: field[i]<=field[from]);
{
    swaps the min value beyond from with the one at from and returns its index
}

int main()
    // sorted
    ensures (forall int i : 0<i<N-2 : field[i] <= field[i+1]);
    // the original values
    ensures (forall int i : 0<i<N-1 && exists int j : old(field[j]) == field[i]);
{
    for (int i = 0; i < N; i++)
        swapMin(i);
}
int[N] field;

This is enough for an SMT solver to proof ensures of main()

int swapMin(int from)
{
  ensures (forall int i<from: field[i] <= field[from]);

  swaps the min value beyond from with the one at from and returns its index
}

int main()
{
  for (int i = 0; i < N; i++)
    swapMin(i);
}
Code-to-Model Compliance

- Based on checking model level specification with code
  - whether code complies to model spec
  - again modular – at granularity of
    - functions/methods
    - objects
    - sw components
  - similar to code contracts but usually coarser granularity
    - e.g., limited to sequences of method calls

- Allows for checking compositionality at model level
  - which is usually easier than at code level
  - handling components as annotated black boxes
  - if strong enough, entire problem **undecidable**
    - code model checking \(\Rightarrow\) halting problem
Can employ functional and extra-functional aspects
- call sequences
- resource demands
- worst-case execution time
- reliability
• Model level spec of DB component:
(f.open; (f.read {!fs.read} + f.write{!fs.write})* ; f.close) | m.status*

• Code:

```java
boolean open(FILE *file) { ... }

int read(int n, char *buffer) {
    assert(buffer);         // check that the nested call is OK
    assert(n>0);
    return fs->fread(file, n, buffer);
}

void close(FILE *file) {
    assert(file.isopen());   // check that the nested call is OK
    file.close();
}

// rest of the protocol checked in the caller code
```
• Things to check:
  ▪ no assertion violated in any execution
    ▪ i.e., adherence to specified provisions
  ▪ no required method called out of order
    ▪ i.e., adherence to specified requirements

• If verified, spec used in composition verification instead of code
Smart Cyber-Physical Systems

• Characteristics:
  - Collaborating computational elements controlling physical entities
  - Designed as a network of interacting elements with physical input and output
  - Real-time, mobility, adaptation
Characteristics:

- 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
Smart Cyber-Physical Systems

- Characteristics:
  - 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

- D3S System & component model: DEECo
Smart Cyber-Physical Systems

- Characteristics:
  - 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

- D3S System & component model: DEECo
  - Dependable Emergent Ensembles of Components
Example: E-mobility

- **POI: Work**
  - Time: 7AM-4PM

- **POI: Cinema**
  - Time: 2PM-4PM

- **POI: Shopping**
  - Time: 4PM-6PM

- **POI: Shopping**
  - Time: 4PM-6PM

- **POI: Home**
  - Time: 6:30PM

- **POI: Home**
  - Time: 6:45PM

[FP7 project ASCENS – Deliverable D7.1 (VW Demonstrator)]
Example: E-mobility

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

POI: Work
Time: 7AM-4PM

POI: Home
Time: 6:45PM

POI: Shopping
Time: 4PM-6PM

POI: Shopping
Time: 4PM-6PM

POI: Home
Time: 6:30PM

POI: Cinema
Time: 2PM-4PM
Example: E-mobility

POI: Work
Time: 7AM-4PM

POI: Home
Time: 6:45PM

POI: Shopping
Time: 4PM-6PM

POI: Shopping
Time: 4PM-6PM

POI: Cinema
Time: 2PM-4PM

[FP7 project ASCENS – Deliverable D7.1 (VW Demonstrator)]
Example: E-mobility

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

POI: Work
Time: 7AM-4PM

POI: Cinema
Time: 2PM-4PM

POI: Shopping
Time: 4PM-6PM

POI: Shopping
Time: 4PM-6PM

POI: Home
Time: 6:45PM

POI: Home
Time: 6:30PM
Example: E-mobility

[FP7 project ASCENS – Deliverable D7.1 (VW Demonstrator)]
Example: E-mobility

[FP7 project ASCENS – Deliverable D7.1 (VW Demonstrator)]
Example: E-mobility

POI: Work
Time: 7AM-4PM

POI: Home
Time: 6:45PM

POI: Shopping
Time: 4PM-6PM

POI: Shopping
Time: 4PM-6PM

POI: Home
Time: 6:30PM

POI: Cinema
Time: 2PM-4PM

[FP7 project ASCENS – Deliverable D7.1 (VW Demonstrator)]
Example: E-mobility

[FP7 project ASCENS – Deliverable D7.1 (VW Demonstrator)]
Example: E-mobility

POI: Work
Time: 7AM-4PM

POI: Home
Time: 6:45PM

POI: Shopping
Time: 4PM-6PM

POI: Shopping
Time: 4PM-6PM

POI: Cinema
Time: 2PM-4PM

[FP7 project ASCENS – Deliverable D7.1 (VW Demonstrator)]
Example: E-mobility

- **POI: Work**  
  Time: 7AM-4PM

- **POI: Cinema**  
  Time: 2PM-4PM

- **POI: Shopping**  
  Time: 4PM-6PM

- **POI: Shopping**  
  Time: 4PM-6PM

- **POI: Home**  
  Time: 6:30PM

- **POI: Home**  
  Time: 6:45PM

[FP7 project ASCENS – Deliverable D7.1 (VW Demonstrator)]
Example: E-mobility

POI: Work
Time: 7AM-4PM

POI: Home
Time: 6:45PM

POI: Shopping
Time: 4PM-6PM

POI: Shopping
Time: 4PM-6PM

POI: Home
Time: 6:30PM

POI: Cinema
Time: 2PM-4PM

[FP7 project ASCENS – Deliverable D7.1 (VW Demonstrator)]
Example: E-mobility

- **POI: Work**
  - Time: 7AM-4PM

- **POI: Cinema**
  - Time: 2PM-4PM

- **POI: Shopping**
  - Time: 4PM-6PM

- **POI: Home**
  - Time: 6:45PM

- **POI: Home**
  - Time: 6:30PM

[FP7 project ASCENS – Deliverable D7.1 (VW Demonstrator)]
Example: E-mobility

POI: Work
Time: 7AM-4PM

POI: Home
Time: 6:45PM

POI: Shopping
Time: 4PM-6PM

POI: Shopping
Time: 4PM-6PM

Key Aspects

• Open-ended
• Dynamic

• Autonomous
• Adaptive
• Emergent behavior

[FP7 project ASCENS – Deliverable D7.1 (VW Demonstrator)]
Classical Component-Based Approach

- Centralized ownership & deployment
- Cannot capture dynamic changes in architecture
- Guaranteed communication needed
- Strong reliance on other components
Service-Oriented Approach

- 3-rd party ownership & deployment
- Dynamic architecture (via service-driven discovery)
- Guaranteed communication needed
- Strong reliance on other services
Agent-Based Approach

- 3-rd party ownership & deployment
- Dynamic architecture (via agent-driven discovery)
- Guaranteed communication needed
- Autonomous (beliefs – desires – intentions)
Agent-Based Approach

- 3-rd party ownership & deployment
- Dynamic architecture (via agent-driven discovery)
- Guaranteed communication needed
- Autonomous (beliefs – desires – intentions)

Agents bring conceptual autonomy
But do not sufficiently translate it to proper software engineering constructs
Agent-Based Approach

- 3-rd party ownership & deployment
- Dynamic architecture (via agent-driven discovery)
- Guaranteed communication needed
- Autonomous (beliefs – desires – intentions)
DEECo Strategy: Hybrid approach
Component-Based Engineering

(software engineering concepts)

Ensemble-based Component Systems

(DEECo)
DEECo Strategy: Hybrid approach

Component-Based Engineering
software engineering concepts)

Agent-Oriented Computing
(autonomy)

Ensemble-based Component Systems
(DEECo)
DEECo Strategy: Hybrid approach

- **Component-Based Engineering** (software engineering concepts)
- **Agent-Oriented Computing** (autonomy)
- **Control System Engineering** (operational normacy)
- **Ensemble-based Component Systems** (DEECo)
- **Ensemble-Oriented Systems** (attribute-based communication)
DEECo as a Refinement of EBCS
DEECo as a Refinement of EBCS

- Components
  - Knowledge
    - Local data + belief
  - Processes
    - Cyclic execution
    - Local knowledge only

monitor the position every 1s
- position
- route calendar
DEECo as a Refinement of EBCS

- **Components**
  - **Knowledge**
    - Local data + belief
  - **Processes**
    - Cyclic execution
    - Local knowledge only

- **Ensembles**
  - **Membership**
    - Declarative
  - **Knowledge exchange**
    - Best-effort
    - Stateless
    - Belief

Members are all parking lots close to a vehicle and the vehicle. Update the vehicle’s belief about the parking-lots’ availability.
DEECo – Concepts
Component

Processes

Knowledge
Component Vehicle = {
  position: IPosition
  availableParkingLots: IParkingLot[]
  route: IRoute
  schedule: ISchedule
  ...
  process updatePlan {
    function = updatePlan
    inputKnowledge = [position, availableParkingLots, ...]
    outputKnowledge = [route, ...]
    scheduling = periodic(1s)
  }
}

Component ParkingLot = {
  freePlaces: Int
  position: IPosition
  ...
  process updateFreePlaces {
    ...
  }
}
DEECo – Concepts
AvailableParkingLotsCloseToDestination

Ensemble

Membership Condition

Holds

Knowledge Exchange
AvailableParkingLotsCloseToDestination

**Ensemble** AvailableParkingLotsCloseToDestination {
  v: IVehicle
  p: IParkingLot

  membership:
  proximity(p.position, v.route) <= DIST_THR
  && p.freePlaces >= FREE_PLACES_THR

  knowledge exchange {
    v.availableParkingLots <- p.id
  }
}
**Ensemble** `AvailableParkingLotsCloseToDestination` {

`v`: `IVehicle`

`p`: `IParkingLot`

**membership**:

`proximity(p.position, v.route) <= DIST_THR`  
`&& p.freePlaces >= FREE_PLACES_THR`

**knowledge exchange**:

`v.availablePakingLots <- p.id`  
}

---

Jan Kofroň, František Plášil, Lecture 13
**Ensemble** `AvailableParkingLotsCloseToDestination` {

- `v`: `IVehicle`
- `p`: `IParkingLot`

**membership**: 
- `proximity(p.position, v.route) <= DIST_THR` 
- `&& p.freePlaces >= FREE_PLACES_THR`

**knowledge exchange**: 
- `v.availablePakingLots <- p.id`

}`
**DEECo – Concepts**

**Ensemble** `AvailableParkingLotsCloseToDestination` {
  `v: IVehicle`
  `p: IParkingLot`

  membership:
  \[
  \text{proximity}(p.\text{position}, v.\text{route}) \leq \text{DIST}_\text{THR} \\
  \&\& \ p.\text{freePlaces} \geq \text{FREE}_\text{PLACES}_\text{THR}
  \]

  knowledge exchange {
  \[
  v.\text{availablePakingLots} <\!-\! p.\text{id}
  \]
  
  }
}
DEECo – Concepts
DEECo – Concepts

AvailableParkingLotsCloseToDestination

AvailableParkingLotsCloseToDestination
AvailableParkingLotsCloseToDestination

(ensemble)
AvailableParkingLotsCloseToDestination
**Ensemble VehiclesCloseByWithTrafficUpdate** {

v1: IVehicle
v2: IVehicle

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

knowledge exchange {
  v1.trafficInfo <- v2.trafficInfo
}

}
**Ensemble VehiclesCloseByWithTrafficUpdate** {
  v1: IVehicle
  v2: IVehicle

  membership:
  proximity(v1.position, v2.position) \leq \text{DIST\_THR}

  knowledge exchange {
  v1.trafficInfo <- v2.trafficInfo
  }
}
**Ensemble VehiclesCloseByWithTrafficUpdate**

v1: IVehicle
v2: IVehicle

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

knowledge exchange {
  v1.trafficInfo <- v2.trafficInfo
}

**AvailableParkingLotsCloseToDestination**

**VehiclesCloseByWithTrafficUpdate**

**AvailableParkingLotsCloseToDestination**
DEECo – Concepts

AvailableParkingLotsCloseToDestination

AvailableParkingLotsCloseToDestination

(ensemble)

AvailableParkingLotsCloseToDestination

(ensemble)
DEECo – Concepts
**Key features:**

The architecture changes dynamically based on static definition of ensembles.

Weakly-connected components, possible to use gossip protocols

Distributed management of component belief
Monitor -> adaptation ~ MAPE – K loop

- **MAPE – K Loop**
  - **Monitor**
    - DEECo runtime regularly check component internal variables (knowledge)
  - **Analyze**
    - Checking which component satisfies membership cond., checks component knowledge
  - **Plan**
    - Select alternative if analysis is ambiguous
  - **Execute**
    - Updated architecture
JDEECo – Framework

Java Virtual Machine

JDEECo Runtime

Local

Hello World Component

Comp X

Knowledge Dissemination

JDEECo Runtime

Local

Replicas

Comp X

Hello World Component

Knowledge Dissemination

...
Communication matters
- Takes time, is unreliable

Components and ensembles have strict time-based semantics
- Cartesian product of automatons of three types:

\[
A(S) = \prod_{\forall C \in \mathcal{C}} \prod_{\forall p \in \mathcal{P}_C} A(p) \times \prod_{\forall C_i, C_j \in \mathcal{C}, C_i \neq C_j} A\left(Q_{C_i}^{C_j}\right) \times \prod_{\forall E \in \mathcal{E}} A(E)
\]