Agent Oriented Modeling and Programming: the case of SARL programming language

Shanghai University - September 22, 2019

Prof. Dr. Stéphane GALLAND

Université de Technologie de Belfort Montbéliard - UTBM
1 Reminders on Multiagent Systems
2 Programming Multiagent Systems with SARL
3 Execution Environment
4 Overview of a MABS Architecture
5 Simulation with a Physic Environment
1. Reminders on Multiagent Systems

2. Programming Multiagent Systems with SARL

3. Execution Environment

4. Overview of a MABS Architecture

5. Simulation with a Physic Environment
Agent (Wooldridge, 2001)

An agent is an entity with (at least) the following attributes / characteristics:

- Autonomy
- Reactivity
- Pro-activity
- Social Skills - Sociability

No commonly/universally accepted definition.
An agent:

- is located in an environment (situatedness)
- perceives the environment through its sensors.
- acts upon that environment through its effectors.
- tends to maximize progress towards its goals by acting in the environment.
Reminders on Multiagent Systems

Programming Multiagent Systems with SARL

Execution Environment

Overview of a MABS Architecture

Simulation with a Physic Environment
Agent: a new paradigm?

- Agent-Oriented Programming (AOP) reuses concepts and language artifacts from Object-Oriented Programming (OOP).
- It also provides a higher-level abstraction than the other paradigms.
Language

- **All agents are holonic (recursive agents).**
- There is not only one way of interacting but infinite.
- Event-driven interactions as the default interaction mode.
- Agent/environment architecture-independent.
- Massively parallel.
- Coding should be simple and fun.

Execution Platform

- Clear separation between Language and Platform related aspects.
- Everything is distributed, and it should be transparent.
- Platform-independent.
### COMPARING SARL TO OTHER FRAMEWORKS

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GAMA</td>
<td>Spatial simulations</td>
<td>✔</td>
<td></td>
<td></td>
<td>GAML, Java</td>
<td><strong>[*]</strong></td>
<td>✔</td>
</tr>
<tr>
<td>Jade</td>
<td>General</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>Java</td>
<td>*</td>
<td>✔</td>
</tr>
<tr>
<td>Jason</td>
<td>General</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>Agent-Speaks</td>
<td>*</td>
<td>✔</td>
</tr>
<tr>
<td>Madkit</td>
<td>General</td>
<td>✔</td>
<td></td>
<td></td>
<td>Java</td>
<td>**</td>
<td>✔</td>
</tr>
<tr>
<td>NetLogo</td>
<td>Social/natural sciences</td>
<td>✔</td>
<td></td>
<td></td>
<td>Logo</td>
<td>***</td>
<td>✔</td>
</tr>
<tr>
<td>Repast</td>
<td>Social/natural sciences</td>
<td>✔</td>
<td></td>
<td></td>
<td>Java, Python, .Net</td>
<td>**</td>
<td></td>
</tr>
<tr>
<td>SARL</td>
<td>General</td>
<td>✔</td>
<td>✔️</td>
<td>✔</td>
<td>SARL, Java, Xtend, Python</td>
<td><strong>[*]</strong></td>
<td>✔</td>
</tr>
</tbody>
</table>

- **a**: Native support of hierarchies of agents.
- **b**: Could be used for agent-based simulation.
- **c**: Could be used for cyber-physical systems, or ambient systems.
- **d**: *: experienced developers; **: for Computer Science Students; ***: for others beginners.
- **e**: Ready-to-use Library: [Jaak Simulation Library](#)
```java
public class ExampleOfClass
    extends SuperClass
    implements SuperInterface {
    // Field
    private int a;
    // Single-initialization field
    private final String b = "example";
    // Constructor
    public ExampleOfClass(int p) {
        this.a = p;
    }
    // Function with return value
    public int getA() {
        return this.a;
    }
    // Simulation of default parameter value
    public void increment(int a) {
        this.a += a;
    }
    public void increment() {
        increment(1);
    }
    // Variadic parameter
    public void add(int... v) {
        for (value : v) {
            this.a += value;
        }
    }
}
```

```java
class ExampleOfClass
    extends SuperClass
    implements SuperInterface {
    // Field
    var a : int
    // Single-initialization field
    // automatic detection of the field type
    val b = "example"
    // Constructor
    new(p : int) {
        this.a = p
    }
    // Function with return value
    def getA : int {
        this.a
    }
    // Real default parameter value
    def increment(a : int = 1) {
        this.a += a
    }
    // Variadic parameter
    def add(v : int*) {
        for (value : v) {
            this.a += value
        }
    }
}
```
Calling getter and setter functions is verbose and annoying.

Syntax for field getting and setting is better.

SARL compiler implicitly calls the getter/setter functions when field syntax is used.

With call: `variable.field`; SARL search for:
1. the function `getField` defined in the variable’s type,
2. the accessible field `field`.

If the previous syntax is left operand of assignment operator, SARL search for:
1. the function `setField` defined in the variable’s type,
2. the accessible field `field`.

```java
class Example {
    private var a : int

    def getA : int {
        this.a
    }
    def setA (a : int) {
        this.a = a
    }
}
class Caller {
    def function (in : Example) {
        // Annoying calls
        in.setA(in.getA + 1)
        // Implicit calls by SARL
        in.a = in.a + 1
    }
}
```
Goal: Extension of existing types with new methods.

Tool: Extension methods.

Principe: The first argument could be externalized prior to the function name.

Standard notation:
function(value1, value2, value3)

Extension method notation:
value1.function(value2, value3)

class Example {
    // Compute the Levenstein distance between two strings of characters
    def distance(s1 : String, s2 : String) : int {
        // Code
    }

    def standardNotation {
        var d = distance("abc", "abz")
    }

    def extensionMethodNotation {
        var d = "abc".distance("abz")
    }
}
Lambda expression: a piece of code, which is wrapped in an object to pass it around.

Notation:

```
[ paramName : paramType, ... | code ]
```

Parameters’ names may be not typed. If single parameter, `it` is used as name.

Parameters’ types may be not typed. They are inferred by the SARL compiler.

```java
class Example {
    def example1 {
        var lambda1 = [
            a : int, b : String |
            a + b.length
        ]
    }

    def example2 {
        var lambda2 = [
            it.length
        ]
    }
}
```
Type for a lambda expression may be written with a SARL approach, or a Java approach.

Let the example of a lambda expression with:
- two parameters, one int, one String, and
- a returned value of type int.

**SARL notation:** $(\text{int}, \text{String}) \Rightarrow \text{int}$

**Java notation:** `Function2<Integer, String, Integer>`
Problem: Giving a lambda expression as function’s argument is not friendly (see example1).

Goal: Allow a nicer syntax.

Principle: If the last parameter is a lambda expression, it may be externalized after the function’s arguments (see example2).

class Example {
    def myfct(a : int, b : String, c : (int) => int) {
        // Code
    }
    def example1 {
        myfct(1, "abc", [it * 2])
    }
    def example2 {
        myfct(1, "abc") [it * 2]
    }
}
Usually, the OO languages provide special instance variables.

SARL provides:

- **this**: the instance of current type declaration (class, agent, behavior...)
- **super**: the instance of the inherited type declaration.
- **it**: an object that depends on the code context.

```java
class Example extends SuperType {
    var field : int

    def thisExample {
        this.field = 1
    }

    def superExample {
        super.myfct
    }

    def itExample_failure {
        // it is unknown in this context
        it.field
    }

    def itExample_inLambda {
        // it means: current parameter
        lambdaConsumer [ it + 1 ]
    }

    def lambdaConsumer(
        x : (int) => int)
    }
}
```
- **Type**: Explicit naming a type may be done with the optional operator: `typeof(TYPE)`.

- **Casting**: Dynamic change of the type of a variable is done with operator: `VARIABLE as TYPE`.

- **Instance of**: Dynamic type testing is supported by the operator: `VARIABLE instanceof TYPE`.

If the test is done in an if-statement, it is not necessary to cast the variable inside the inner blocks.

class Example {
    def typeofExample {
        var t : Class<?>
        t = typeof(String)
        t = String
    }

    def castExample {
        var t : int
        t = 123.456 as int
    }

    def instanceExample(t:Object) {
        var x : int
        if (t instanceof Number) {
            x = t.intValue
        }
    }
}
SARL provides special operators in addition to the classic operators from Java or C++:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Semantic</th>
<th>Java equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>a == b</td>
<td>Object equality test</td>
<td>a.equals(b)</td>
</tr>
<tr>
<td>a != b</td>
<td>Object inequality test</td>
<td>!a.equals(b)</td>
</tr>
<tr>
<td>a === b</td>
<td>Reference equality test</td>
<td>a == b</td>
</tr>
<tr>
<td>a !== b</td>
<td>Reference inequality test</td>
<td>a != b</td>
</tr>
<tr>
<td>a &lt;== b</td>
<td>Compare a and b</td>
<td>Comparable interface</td>
</tr>
<tr>
<td>a .. b</td>
<td>Range of values ([a, b])</td>
<td>n/a</td>
</tr>
<tr>
<td>a..&lt; b</td>
<td>Range of values ([a, b))</td>
<td>n/a</td>
</tr>
<tr>
<td>a &gt;.. b</td>
<td>Range of values ((a, b])</td>
<td>n/a</td>
</tr>
<tr>
<td>a ** b</td>
<td>Compute (a^b)</td>
<td>n/a</td>
</tr>
<tr>
<td>a -&gt; b</td>
<td>Create a pair ((a, b))</td>
<td>n/a</td>
</tr>
<tr>
<td>a ?: b</td>
<td>If a is not null then a else b</td>
<td>a == null ? b : a</td>
</tr>
<tr>
<td>a?.b</td>
<td>If a is not null then a.b is called</td>
<td>a == null ? defaultValue : a.b</td>
</tr>
<tr>
<td>if (a) b else c</td>
<td>Inline condition</td>
<td>a ? b : c</td>
</tr>
</tbody>
</table>
SARL allows overriding or definition operators.

Each operator is associated to a specific function name that enables the developer to redefine the operator’s code.

Examples of operators in SARL:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Function name</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>col += value</code></td>
<td><code>operator_add(Collection, Object)</code></td>
<td>Add an value into a collection.</td>
</tr>
<tr>
<td><code>a ** b</code></td>
<td><code>operator_power(Number, Number)</code></td>
<td>Compute the power b of a.</td>
</tr>
</tbody>
</table>

```java
class Vector {
    var x : float
    var y : float
    new (x : float, y : float) {
        this.x = x ; this.y = y
    }
    def operator_plus(v: Vector) : Vector {
        new Vector(this.x + v.x, this.y + v.y)
    }
}

class X {
    def fct {
        var v1 = new Vector(1, 2)
        var v2 = new Vector(3, 4)
        var v3 = v1 + v2
    }
}
```
Multiagent System in SARL

A collection of agents interacting together in a collection of shared distributed spaces.

4 main concepts
- Agent
- Capacity
- Skill
- Space

3 main dimensions
- Individual:: the Agent abstraction (Agent, Capacity, Skill)
- Collective:: the Interaction abstraction (Space, Event, etc.)
- Hierarchical:: the Holon abstraction (Context)


http://www.sarl.io
Agent

- An agent is an autonomous entity having some intrinsic skills to implement the capacities it exhibits.
- An agent initially owns native capacities called Built-in Capacities.
- An agent defines a Context.
package org.multiagent.example

agent HelloAgent {

    var myvariable : int
    val myconstant = "abc"

    on Initialize {
        println("Hello World!")
    }

    on Destroy {
        println("Goodbye World!")
    }

}
package org.multiagent.example

agent HelloAgent {

  var myvariable : int
  val myconstant = "abc"

  on Initialize {
    println("Hello World!")
  }

  on Destroy {
    println("Goodbye World!")
  }

}
package org.multiagent.example

agent HelloAgent {

    var myvariable : int
    val myconstant = "abc"

    on Initialize {
        println("Hello World!")
    }

    on Destroy {
        println("Goodbye World!")
    }
}

This block of code contains all the elements related to the agent.
package org.multiagent.example

agent HelloAgent {

  var myvariable : int
  val myconstant = "abc"

  on Initialize {
    println("Hello World!")
  }

  on Destroy {
    println("Goodbye World!")
  }

}
package org.multiagent.example

agent HelloAgent {

  var myvariable : int

  val myconstant = "abc"

  on Initialize {
    println("Hello World!")
  }

  on Destroy {
    println("Goodbye World!")
  }
}

Define a constant with name "myconstant" and the given value.
package org.multiagent.example

agent HelloAgent {
  var myvariable : int
  val myconstant = "abc"

  on Initialize {
    println("Hello World!")
  }

  on Destroy {
    println("Goodbye World!")
  }
}
package org.multiagent.example

agent HelloAgent {

  var myvariable : int
  val myconstant = "abc"

  on Initialize {
    println("Hello World!")
  }

  on Destroy {
    println("Goodbye World!")
  }

}
**CAPACITY AND SKILL**

**Action**
- A specification of a transformation of a part of the designed system or its environment.
- Guarantees resulting properties if the system before the transformation satisfies a set of constraints.
- Defined in terms of pre- and post-conditions.

**Capacity**

**Skill**
A possible implementation of a capacity fulfilling all the constraints of its specification, the capacity.

Enable the separation between a generic behavior and agent-specific capabilities.
EXAMPLE OF CAPACITY AND SKILL

```java
capacity Logging {
    def debug(s : String)
    def info(s : String)
}
```

```java
skill BasicConsoleLogging implements Logging {
    def debug(s : String) {
        println("DEBUG:" + s)
    }
    def info(s : String) {
        println("INFO:" + s)
    }
}
```

```java
agent HelloAgent {
    uses Logging
    on Initialize {
        setSkill(new BasicConsoleLogging)
        info("Hello World!")
    }
}
```

Definition of a capacity that permits to an agent to print messages into the log system.
EXAMPLE OF CAPACITY AND SKILL

capacity Logging {
    def debug(s : String)
    def info(s : String)
}

skill BasicConsoleLogging implements Logging {
    def debug(s : String) {
        println("DEBUG: " + s)
    }
    def info(s : String) {
        println("INFO:" + s)
    }
}

agent HelloAgent {
    uses Logging
    on Initialize {
        setSkill(new BasicConsoleLogging)
        info("Hello World!")
    }
    on Destroy {
        info("Goodbye World!")
    }
}

Define a function that could be invoked by the agent.
Define the skill that implements the Logging capacity.

capacity Logging {
    def debug(s : String)
    def info(s : String)
}

skill BasicConsoleLogging implements Logging {
    def debug(s : String) {
        println("DEBUG: " + s)
    }
    def info(s : String) {
        println("INFO: " + s)
    }
}

uses Logging

on Initialize {
    setSkill(new BasicConsoleLogging)
    info("Hello World!")
}

on Destroy {
    info("Goodbye World!")
}
capacity Logging {
    def debug(s : String)
    def info(s : String)
}

skill BasicConsoleLogging implements Logging {
    def debug(s : String) {
        println("DEBUG: " + s)
    }

    def info(s : String) {
        println("INFO: " + s)
    }
}

Every function declared into the implemented capacity must be implemented in the skill. The current implementations output the message onto the standard output stream.
The use of a capacity into the agent code is enabled by the "uses" keyword.

```scala
agent HelloAgent {
  uses Logging

  on Initialize {
    setSkill(new BasicConsoleLogging)
    info("Hello World!")
  }

  on Destroy {
    info("Goodbye World!")
  }
}
```

```scala
capacity
  def debug(s : String) {
    println("DEBUG: " + s)
  }

  def info(s : String) {
    println("INFO:" + s)
  }
}
```

```scala
skill BasicConsoleLogging
  implements Logging {
    def debug(s : String) {
      println("DEBUG:" + s)
    }

    def info(s : String) {
      println("INFO:" + s)
    }
  }
```
capacity Logging {
    def debug(s : String)
    def info(s : String)
}

skill BasicConsoleLogging {
    def debug(s : String) {
        println("DEBUG: \$s")
    }
    def info(s : String) {
        println("INFO: \$s")
    }
}

agent HelloAgent {
    uses Logging
    on Initialize {
        setSkill(new BasicConsoleLogging)
        info("Hello World!")
    }
    on Destroy {
        info("Goodbye World!")
    }
}

All functions defined into the used capacities are directly callable from the source code.
capacity Logging {
    def debug(s : String)
    def info(s : String)
}

agent HelloAgent {
    uses Logging
    on Initialize {
        setSkill(new BasicConsoleLogging)
        info("Hello World!")
    }
    on Destroy {
        info("Goodbye World!")
    }
}

An agent MUST specify the skill to use for a capacity (except for the buildin skills that are provided by the execution framework)
### Space

Support of interaction between agents respecting the rules defined in various Space Specifications.

### Space Specification

- Defines the rules (including action and perception) for interacting within a given set of Spaces respecting this specification.
- Defines the way agents are addressed and perceived by other agents in the same space.
- A way for implementing new interaction means.

The spaces and space specifications must be written with the Java programming language.
**Context AND SPACES**

- Defines the boundary of a sub-system.
- Collection of Spaces.
- Every Context has a Default Space.
- Every Agent has a Default Context, the context where it was spawned.
Default Space: an Event Space

- Event-driven interaction space.
- Default Space of a context, contains all agents of the considered context.
- Event: the specification of some occurrence in a Space that may potentially trigger effects by a participant.
EXAMPLE OF INTERACTIONS: PING - PONG

Default Space

PingAgent

PongAgent

Wait for partner
Send Ping
Send Pong
Send Ping
Send Pong
Wait 1 second
Wait 1 second
EXAMPLE OF INTERACTIONS: PING - PONG

```java
event Ping {
    var value : Integer
    new (v : Integer) {
        value = v
    }
}

event Pong {
    var value : Integer
    new (v : Integer) {
        value = v
    }
}

agent PongAgent {
    uses DefaultContextInteractions
    on Initialize {
        println("Waiting for ping")
    }
    on Ping {
        println("Recv Ping: "+ occurrence.value)
        println("Send Pong: "+ occurrence.value)
        emit(new Pong(occurrence.value))
    }
}

agent PingAgent {
    uses Schedules
    uses DefaultContextInteractions
    var count : Integer
    on Initialize {
        println("Starting PingAgent")
        count = 0
        in(2000) [ sendPing ]
    }
    def sendPing {
        if (defaultSpace.participants.size > 1) {
            emit(new Ping(count))
            count = count + 1
        } else {
            in(2000) [ sendPing ]
        }
    }
    on Pong {
        in(1000) [
            println("Send Ping: "+count)
            emit(new Ping(count))
            count = count + 1
        ]
    }
}
```

Multiagent Systems  Programming MAS with SARL  Execution Environment
Overview of a MABS Architecture  Physic Environment
The SARL syntax is explained into the “General Syntax Reference” on the SARL website.

http://www.sarl.io/docs/

SARL is 100% compatible with Java

- Any Java feature or library could be included and called from SARL.
- A Java application could call any public feature from the SARL API.
1. Reminders on Multiagent Systems

2. Programming Multiagent Systems with SARL

3. Execution Environment

4. Overview of a MABS Architecture

5. Simulation with a Physic Environment
### Runtime Environment Requirements

- Implements SARL concepts.
- Provides Built-in Capacities.
- Handles Agent's Lifecycle.
- Handles resources.

### Janus as a SARL Runtime Environment

- Fully distributed.
- Dynamic discovery of Kernels.
- Automatic synchronization of kernels’ data (easy recovery).
- Micro-Kernel implementation.
- Official website: [http://www.janusproject.io](http://www.janusproject.io)

Other SREs may be defined.
1. Reminders on Multiagent Systems

2. Programming Multiagent Systems with SARL

3. Execution Environment

4. Overview of a MABS Architecture

5. Simulation with a Physic Environment
GENERAL ARCHITECTURE

Agent

Environment

- Resources, services, objects
- Rules, laws
- Physical structures (spatial and topological)
- Communication structures (stigmergy, implicit communication)
- Social structure

Direct interaction

Perceptions

Actions

Multiagent Systems Programming MAS with SARL Execution Environment Overview of a MABS Architecture Physic Environment
Overview of a MABS Architecture

Multiagent Systems Programming MAS with SARL Execution Environment

**Environment**
- Resources, services, objects
- Rules, laws
- Physical structures (spatial and topological)
- Communication structures (stigmergy, implicit communication)
- Social structure

**Agent**

**Simulation Controller**

**Direct interaction**

**Perceptions**

**Actions**
**Overview of a MABS Architecture**

**Environment**
- Resources, services, objects
- Rules, laws
- Physical structures (spatial and topological)
- Communication structures (stigmergy, implicit communication)
- Social structure

**Agent**

**Observer**

**Simulation Controller**

**Change events**

**Direct interaction**

**Perceptions**

**Actions**

**Rendering Software Modules**
(1D, 2D or 3D)
GENERAL ARCHITECTURE

Avatar
Immersed User

Observer

Simulation Controller

Agent

Direct interaction

Perceptions
Actions

Environment
- Resources, services, objects
- Rules, laws
- Physical structures (spatial and topological)
- Communication structures (stigmergy, implicit communication)
- Social structure

Change events

Rendering Software Modules (1D, 2D or 3D)

Agent

Multiagent Systems Programming MAS with SARL Execution Environment
Overview of a MABS Architecture Physic Environment
Behaviors
Internal Agent Architecture
Modeling of agent deliberation processes (agent mind)

Environment
Physical objects of the world, their structuring, environment dynamics (evaporation...)

Scheduling
Temporal dynamic of the system
Modeling of the time progress, and of the agent scheduling

Interaction
Modeling the concurrent events
Modeling the results of actions and interactions at a given time

Definitions of action, perception, and conflict resolution
SITUATED ENVIRONMENT MODEL

Agent

Environment Interface

Environment

- Resources
- Physical structures
- Physical rules

Environmental Object Collection

Environment's State

Influence Solver
ensure valid environment state according to environment laws

Perception data structure
spatial tree, grid, graph

Influence

Agent

Behavioral component
Agent mind

Physical component
Agent body

Direct interaction

Multiagent Systems Programming MAS with SARL Execution Environment
Overview of a MABS Architecture Physic Environment Galland (2009)
**State variables of the decisional component**
Readable/modifiable only by the agent

**State variables of the physical component**
Readable by the agent
Modifiable by the environment
How to support simultaneous actions from agents?

1. An agent does not change the state of the environment directly.
2. Agent gives a state-change expectation to the environment: the influence.
3. Environment gathers influences, and solves conflicts among them for obtaining its reaction.
4. Environment applies reaction for changing its state.
- The agent has the capacity to use its body.
- The body supports the interactions with the environment.

```scala
event Perception {
  val object : Object
  val relativePosition : Vector
}

capacity EnvironmentInteraction {
  moveTheBody(motion : Vector)
  move(object : Object,
        motion : Vector)
  executeActionOn(object : Object,
                   actionName : String,
                   parameters : Object*)
}

space PhysicEnvironment {
  def move(object : Object,
            motion : Vector) {
    // ...
  }
}

skill PhysicBody implements EnvironmentInteraction {
  val env : PhysicEnvironment
  val body : Object
  def moveTheBody(motion:Vector) {
    move(this.body, motion)
  }
  def move(object : Object,
           motion : Vector) {
    env.move(object, motion)
  }
}
```
Each vehicle is simulated but road signs are skipped ⇒ mesoscopic simulation.

The roads are extracted from a Geographical Information Database.

The simulation model is composed of two parts (Galland, 2009):
1. the environment: the model of the road network, and the vehicles.
2. the driver model: the behavior of the driver linked to a single vehicle.
### Road Network

- **Road polylines**: \( S = \{ \langle \text{path}, \text{objects} \rangle \mid \text{path} = \langle (x_0, y_0) \cdots \rangle \} \)
- **Graph**: \( G = \{ S, S \mapsto S, S \mapsto S \} = \{ \text{segments, entering, exiting} \} \)

### Operations

- Compute the set of objects perceived by a driver (vehicles, roads...):

\[
P = \left\{ \begin{array}{l}
  o \\
  \text{distance}(d, o) \leq \Delta \land \\
  o \in O \land \\
  \forall (s_1, s_2), \text{path} = s_1.\langle p, O\rangle.s_2
\end{array} \right\}
\]

where \( \text{path} \) is the roads followed by a driver \( d \).

- Move the vehicles, and avoid physical collisions.
ARCHITECTURE OF THE DRIVER AGENT

ENVIRONMENT

Path planning

Collision avoidance

perceived objects

Car

new position

Jasim model (Galland, 2009)
Based on the A* algorithm (Dechter, 1985; Delling, 2009):
- extension of the Dijkstra’s algorithm: search shortest paths between the nodes of a graph.
- introduce the heuristic function $h$ to explore first the nodes that permit to converge to the target node.

Inspired by the D*-Lite algorithm (Koenig, 2005):
- A* family.
- supports dynamic changes in the graph topology and the values of the edges.
- **Principle**: compute the acceleration of the vehicle to avoid collisions with the other vehicles.

- **Intelligent Driver Model** (Treiber, 2000)

  \[
  \text{followerDriving} = \begin{cases} 
  - \frac{(v \Delta v)^2}{4b \Delta p^2} & \text{if the ahead object is far} \\
  -a \frac{(s + vw)^2}{\Delta p^2} & \text{if the ahead object is near}
  \end{cases}
  \]

- **Free driving**:

  \[
  \text{freeDriving} = a \left( 1 - \left( \frac{v}{v_c} \right)^4 \right)
  \]
agent StandardDriver {
  uses DrivingCapacity

  var path : Path

  on Initialize {
    setSkill(DrivingCapacity, IDM_Dstart_DrivingSkill)
  }

  on Perception {
    var stopVehicleInStandardCondition = isVehicleStop(occurrence)
    var siren = occurrence.body.getFirstPerceptionAtCurrentPosition(Siren)
    var stopVehicleForEmergencyVehicle = isStopWhenEmergencyVehicle(siren)

    if (!stopVehicleForEmergencyVehicle&&!stopVehicleInStandardCondition){
      var motion : Vector2i = null
      path = updatePathWithDstart(path, occurrence)

      if (!path.empty) {
        motion = followPathWithIDM(path, occurrence)
      }

      if (motion !== null & motion.lengthSquared > 0) {
        move(motion, true)
        this.previousOrientation = direction
      }
    }
  }
}
SIMULATION OF EMERGENCY SITUATION ON A FRENCH HIGHWAY

Multiagent Systems Programming MAS with SARL Execution Environment
Overview of a MABS Architecture Physic Environment
**SEVERAL PERSPECTIVES FOR SARL**

- **Language:**
  - Statements for Space and Space specification.
  - Statements for organizational concepts.
  - Design by contract with SARL.
  - Ontology support.

- **Development Environment:**
  - UI tools for creating (simulated) universes.
  - IntelliJ support.

- **Run-time Environments:**
  - Real-time implementation of Janus for embedded systems.
  - Addition of modules to Janus for agent-based simulation (drones, traffic, pedestrians)
  - Extension of GAMA for being a SARL Runtime Environment.
  - Extension of MATSIM for being a SARL Runtime Environment.
Agent Oriented Modeling and Programming: the case of SARL programming language

Shanghai University - September 22, 2019

Prof.Dr. Stéphane GALLAND
Appendix
Full Professor

Co-founder and Deputy Director of the CIAD Laboratory
Université de Technologie de Belfort-Montbéliard, France

Topics: Multiagent systems, Agent-based simulation, Agent-oriented software engineering, Mobility and traffic modeling

Web page: http://www.multiagent.fr/People:Galland_stephane
Email: stephane.galland@utbm.fr

Open-source contributions:
- http://www.sarl.io
- http://www.janusproject.io
- http://www.aspecs.org
- http://www.arakhne.org
- https://github.com/gallandarakhneorg/


