SARL and Janus: State of the works and Perspectives
EuSarlCon 2019 - May 2nd 2019
Prof. Dr. Stéphane GALLAND
1 State of SARL and Janus

2 Perspectives for SARL and Janus
1 State of SARL and Janus

2 Perspectives for SARL and Janus
   Metamodel and Language Evolutions
   Run-time Framework Evolution
Agent Programming

C++  SARL  Java  Gaml  Logo

Swarm  Ad-hoc  Janus  Repast  Jade  Matsim  Gama  NetLogo
**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

This table was done according to experiments with my students.

<table>
<thead>
<tr>
<th>Name</th>
<th>Domain</th>
<th>Hierar.(^a)</th>
<th>Simu.(^b)</th>
<th>C.Phys.(^c)</th>
<th>Lang.</th>
<th>Beginners(^d)</th>
<th>Free</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAMA</td>
<td>Spatial simulations</td>
<td>✓</td>
<td></td>
<td>GAML, Java</td>
<td>✓</td>
<td>**[*]</td>
<td>✓</td>
</tr>
<tr>
<td>Jade</td>
<td>General</td>
<td>✓</td>
<td>✓</td>
<td>Java</td>
<td>✓</td>
<td>*</td>
<td>✓</td>
</tr>
<tr>
<td>Jason</td>
<td>General</td>
<td>✓</td>
<td>✓</td>
<td>Agent-Speaks</td>
<td>✓</td>
<td>*</td>
<td>✓</td>
</tr>
<tr>
<td>Madkit</td>
<td>General</td>
<td>✓</td>
<td></td>
<td>Java</td>
<td>✓</td>
<td>**</td>
<td>✓</td>
</tr>
<tr>
<td>NetLogo</td>
<td>Social/natural sciences</td>
<td></td>
<td>✓</td>
<td>Logo</td>
<td>✓</td>
<td>***</td>
<td>✓</td>
</tr>
<tr>
<td>Repast</td>
<td>Social/natural sciences</td>
<td>✓</td>
<td></td>
<td>Java, Python, .Net</td>
<td>✓</td>
<td>**</td>
<td></td>
</tr>
<tr>
<td>SARL</td>
<td>General</td>
<td>✓</td>
<td>✓(^e)</td>
<td>C.Phys.(^c)</td>
<td>SARL, Java, Xtend, Python</td>
<td>**[*]</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](#)
### Overview of SARL Concepts

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**.

```java
agent HelloAgent {
  on Initialize {
    println("Hello World!")
  }
  on Destroy {
    println("Goodbye World!")
  }
}
```
**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.
# Space as the Support of Interactions Between Agents

## 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**

- 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**

**Space 1**

**Space 2**

**Space 3**
HOLONS OR AGENTS COMPOSED BY AGENTS

Horizontal

Group 0
Role 3
Holon 1
Role 1
Role 2
Group 1
Group 2

Vertical

Level 0

Level 1

Level 2

Level 3

Level 4

Level 5

Level 6

Production Groups

g3: Holonic Group

g4: Recruitment Group

Representative
Head
Peer
Peer
Head

Stand-Alone
Representative

Role 4
Role 4
Role 5
Role 6
Role 7
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.
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

Other SREs may be defined.
1 State of SARL and Janus

2 Perspectives for SARL and Janus
   Metamodel and Language Evolutions
   Run-time Framework Evolution
### Action Selection Architecture ≈ BDI
- BDI-like framework: Goals, Believes, Actions.
- Definition of statements.

### Environment Metamodel ≈ Artifact
- Artifact-like framework: artifact, use interaction, endogeneous dynamics.
- Definition of specific statements.

### Organizational Modeling ≈ CRIO
- Definition of the mapping between the CRIO concepts and the SARL concepts.
- Definition of statements for roles and interaction definitions.
**Action Selection Architecture ≈ BDI**

- BDI-like framework: Goals, Believes, Actions.
- Definition of statements.

**Design by contract with SARL**

- Formal properties into the SARL concepts: invariant, post-, pre-conditions.
- Formal properties for interaction protocols.
- Enforcement of the property validation during run-time.
<table>
<thead>
<tr>
<th>SIMULATION FRAMEWORK</th>
</tr>
</thead>
</table>

### Time Management
- Management of the simulation time by the run-time framework.
- Management of the simulation time over a network of computers.

### Agent Environment
- Definition of tools for defining the agent environment: artifacts, smart objects...
- Addition of modules for agent-based simulation of drones, road traffic, crowd, autonomous cars, IOT...

### User Interface
- UI tools for simulators, like Netlogo of Gama.
<table>
<thead>
<tr>
<th>Janus for Embedded Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real-time implementation of Janus for embedded systems.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>New Run-time Environments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Akka for creating a new SARL run-time Environment dedicated to Cloud computing.</td>
</tr>
<tr>
<td>GAMA for running SARL agents.</td>
</tr>
<tr>
<td>Extending MATSIM with SARL capacities.</td>
</tr>
</tbody>
</table>
Thank you for your attention...
Appendix
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.
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 {
  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:

\[
\left[ \text{paramName} : \text{paramType}, \ldots \mid \text{code} \right]
\]

Parameters’ names may be not typed. If single parameter, \textit{it} is used as name.

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

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).
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((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 a `if`-statement, it is not necessary to cast the variable inside the inner blocks.

```java
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 else a default value is used</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>col += value</td>
<td>operator_add(Collection, Object)</td>
<td>Add an value into a collection.</td>
</tr>
<tr>
<td>a ** b</td>
<td>operator_power(Number, Number)</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
    }
}
```
1 About the Author

2 Bibliography
Full Professor

Université de Bourgogne Franche-Comté
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/