Artificial Intelligence for Serious Game

Revision 14

Prof.Dr. Stéphane GALLAND

Module MA76

Université de Bourgogne Franche-Comté - UTBM
90010 Belfort cedex, France - `http://www.multiagent.fr`

Chapter 1
Introduction and Overview

Prof.Dr. Stéphane GALLAND

GOALS OF THIS LECTURE

During this lecture, I will present:

1. the basics of the Serious Game;
2. the keys concepts from Virtual Reality, ALife, Game;
3. the basics of the multiagent systems;
4. the basics of the multiagent simulation;

5. models of physic environment,
   - Examples: highway and crowd simulations;

OUTLINE

OUTLINE

(Bergeron, 2006)

A serious game is an interactive computer application, with or without a significant hardware component, that:

- has a challenging goal,
- is fun to play and/or engaging,
- incorporates some concept of scoring, and
- imparts to the user a skill, knowledge, or attitude that can be applied in the real world.

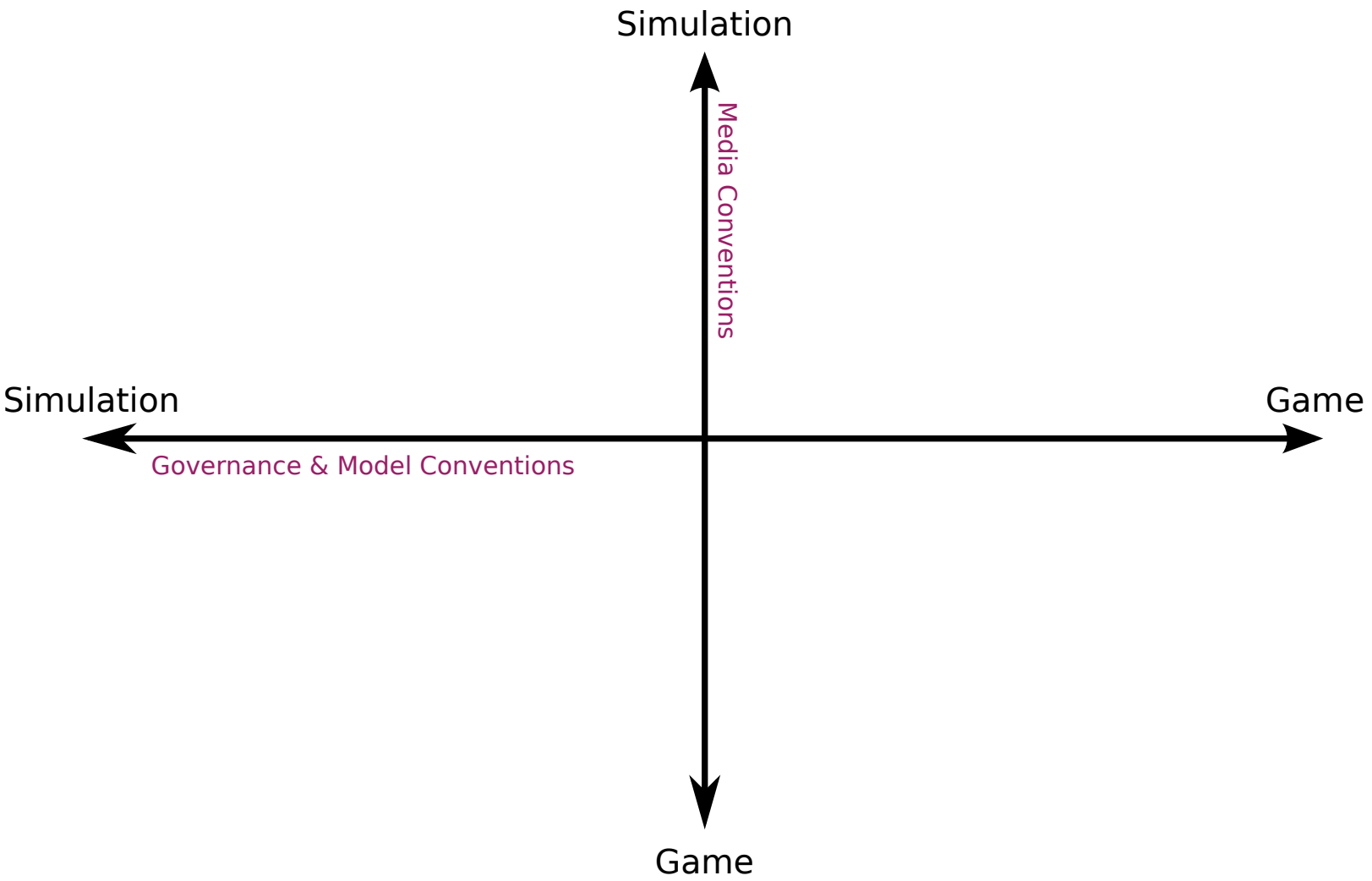
ADVANTAGES OF GAME

- Motivation/Engagement.
- Interactivity.
- Mechanic is the Learning:
  - To beat the game is to learn the message/skill.
  - But only when done right, very difficult.
  - Beyond content to problem solving/systems learning.
- Adaptive to the Learner.
- Real-Time Assessment:
  - Analytics/Data/Log Files.

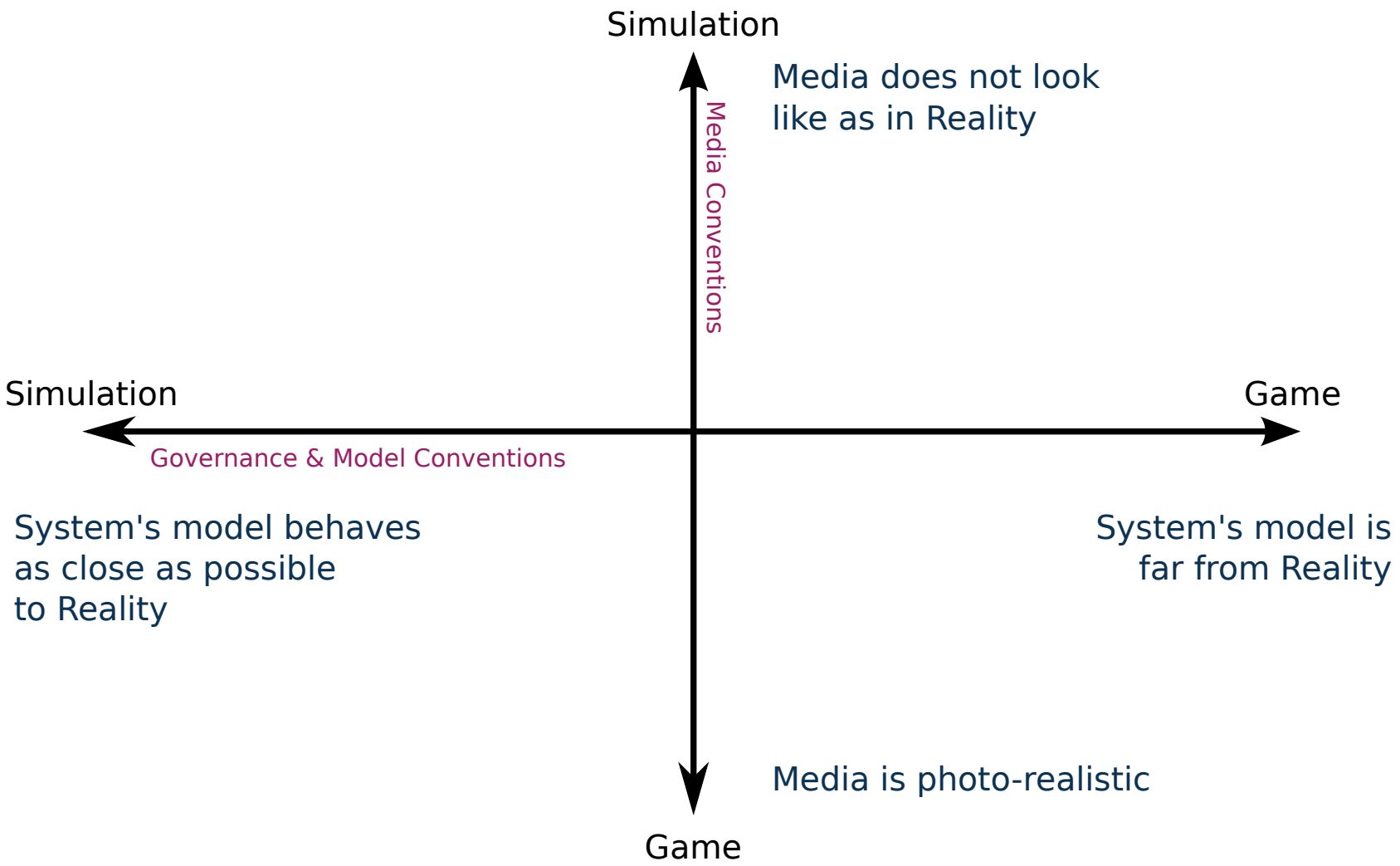
[Serious] Game = Simulation + Media

- Simulation: Software that reproduces:
  - the processes and behaviors of the game's universe, and
  - manage the players' interaction in order to enforce the game's universe consistency.
- Media: Software engine that displays realistic images, and play sounds. Outside the scope of MA76.

Sawyer (2008)


Simulation
Media Conventions
Simulation
Game
Governance & Model Conventions
Game

Sawyer (2008)


Simulation
Media does not look like as in Reality
Media Conventions
Simulation
Game
Governance & Model Conventions
System's model behaves as close as possible to Reality
System's model is far from Reality
Media is photo-realistic
Game

SIMULATIONS AND GAMES

Packet Tracer

Virtual U

Bridge Builder

Federal Budget Game

Simulation

Media Conventions

Simulation

Game

Governance & Model Conventions

Sim City

Half-Life 2

MS Flight Simulator

Every Day Shooter

Game

OUTLINE

VIRTUAL REALITY: FUNCTIONAL DEFINITIONS

Definition

The goal of Virtual Reality is to allow people's sensori-motor activity inside an artificial world which could be imaginary, symbolic or a simulation of several aspects of the Reality.

Definition

Virtual Reality allows people to be extracted from physical reality to virtually change of time, of place and/or of type of interaction: interaction with an environment simulating the Reality, or interaction with an imaginary world or a symbolic system.

Fuchs (2006)

Definition

Virtual reality is a scientific and technical field, which is exploiting computer science and behavioral interfaces in order to simulate in a virtual world the behavior of 3D entities, which are in real time interaction with users in pseudonatural immersion via sensorimotor channels.

Fuchs (2006)

BRIEF HISTORY OF VIRTUAL REALITY

Sensorama
1962

Simulator &
Headscreen
1970

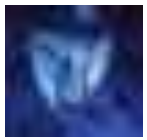
Computer-
Aided Design
1965

Business
Companies
1980-1985

Data glove
1980

People
knows VR
1990-1995

VR Platform
in France
2000



VR Startups
2005

Wii:VR for
Gamers
2006

VR for
all
2009

VIRTUAL REALITY: KEY CONCEPTS

- Immersion:
  - Override human senses with dedicated devices to immerse the user inside a virtual world:
    - Vision: stereoscopic devices.
    - Hearing: 7.1 devices.
    - Touch: 3D mouses and haptic devices.
  - Simulate the virtual world in a real way to obtain a cognitive immersion.
- Interaction:
  - Allow the human to move or change the virtual objects, to speak to the virtual characters, etc.

VIRTUAL WORLDS

Definition

A virtual world is a computer-based simulated environment intended for its users to inhabit and interact via avatars.

- This habitation is usually done in the form of two or three-dimensional graphical representations of humanoids (or other graphical or text-based avatars).
- Some, but not all, virtual worlds allow multiple users a time.

BRIEF HISTORY OF VIRTUAL WORLDS

Flight Simulator
1970

First online VW
(Habitat)
1987

Interactive communities
(WorldsAway and
Dreamscape)
1987-1995

First realistic
virtual world
(Helsinski Arena 2000)
2000

Huge VW
(WoW)
2004

3D computer graphics
(Cityspace)
1993

Community driven
virtual world
(The Palace)
1995

Infinite & Business
(Second Life)
2003

VIRTUAL WORLDS: KEY CONCEPTS (1/2)

- Avatar:
  - Each user has its representative inside the virtual world.
  - An avatar permits to its owning human to interact with virtual world's entities.
- Animat, Virtual Character:
  - An artificial life's avatar.
- Shadow Avatar:
  - A user's interaction-less representative.

VIRTUAL WORLDS: KEY CONCEPTS (2/2)

- Interaction:
  - Human user can interact with the other world's entities via its avatar:
    - speak/text based interaction,
    - action proceeding and world's perception.
- Persistent:
  - World is (mostly) always available and world events happen continually.
- Environment Dynamics:
  - Dynamics of the world must be implemented (rain, wind, etc.) It cannot be controled by avatars.

Definition

Artificial Life (or Alife) is a field of study and associated art form which examine systems related to life, its processes, and its evolution through simulations using computer models, robotics, and biochemistry.

Artificial life studies the evolution of agents, populations of computer-simulated life forms, in artificial environments.

There are three main kinds of alife:

- soft-alife, from software;
- hard-alife, from hardware and robostic; and
- wet-alife, from biochemistry.

ALIFE KEY ELEMENTS

- We are not concerned with the outcome of an action, but how that outcome was reached.
- Nature is fundamentally parallel.
- Our goal is to imitate natural systems.

- We can achieve interactive behaviour simulation through behaviour generation code - serial programming techniques.
- Bio-Genetic-Cognitive techniques can allow software to imitate natural systems.

BRIEF HISTORY OF ALIFE

"Unconscious Memory"
Samuel Butler
1880

"Darwin Among the Machines"
Samuel Butler
1859

"Theory and Organisation of Complicated Automata"
Self-replicating machines
John von Neumann
1949

"Life Game"
Cells' division and replication
John Conway
1970

Jacques de Vaucanson's
Mechanical Duck
1753

Programmable
controllers
1900's

"Computing machinery and intelligence"
Alan Turing
1948/1950

"Symbiogenesis"
Symbiotic genes
Nils Barricelli
1953

"Adaptation in Natural
nd Artificial Systems"
1st PhD in Computer Science
John Holland
1975

FROM ALIFE TO MULTIAGENT SYSTEMS

- ALife may consist of populations of simple programs or specifications.
- There is no single program that directs/controls all of the other programs.
- Each program details the way in which a simple entity reacts to local situations in its environment, including interactions with other entities.
- There are no coded rules in the system that dictate a global behaviour.
- Any behaviour at higher level than the individual programs is therefore named emergent.

Multiagent Systems is a computer-science approach that is suitable for Alife simulation.

Definition

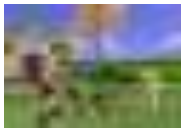
Artificial Intelligence for Games is the field studying the enhancement of games with autonomous and human-like entities.

Multiagent Systems is a computer-science approach that is suitable for Games Artificial Intelligence.

BRIEF HISTORY OF G-AI

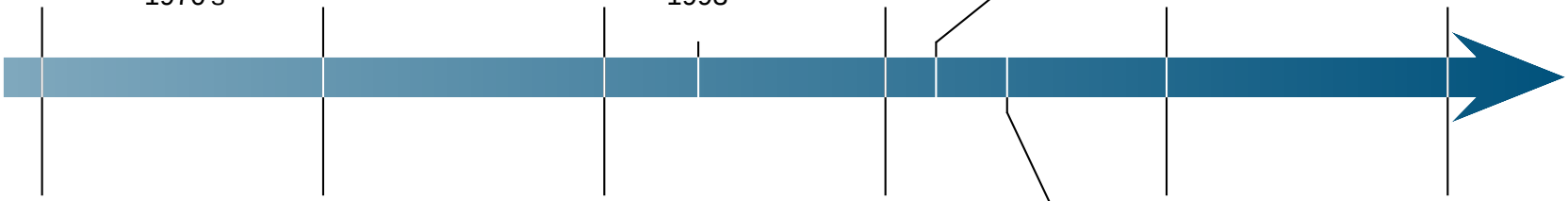
Pacman
Pong
Space Invader
1970's

Creatures
1993



The Sims
2000

Black & White
2001



Beneath a steel sky
Thief: the Dark Project
Metal Gear Solid
1990's

Full Spectrum Warriors
2004

KEYS POINTS FOR G-AI

- AI in most modern games addresses three basic needs:
  - the ability to move characters,
  - the ability to make decisions about where to move,
  - the ability to think tactically or strategically.

- Even though we've gone from using state-based AI every-where (they are still used in most places) to a broad range of techniques, they all fulfill the same three basic requirements.

OUTLINE

Five ongoing trends have marked the history of computing

- Ubiquity;
- Interconnection;
- Intelligence;
- Delegation;
- Human-orientation: easy/natural to design/implement/use.

Other Trends in Computer Science

- Grid Computing;
- Ubiquitous Computing;
- Semantic Web.

AGENT: A FIRST DEFINITION

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.

Autonomy

Agents encapsulate their internal state (that is not accessible to other agents), and make decisions about what to do based on this state, without the direct intervention of humans or others;

- Able to act without any direct intervention of human users or other agents.
- Has control over his own internal state.
- Has control over his own actions (no master/slave relationship)
- Can, if necessary/required, modify his behavior according to his personal or social experience (adaptation-learning).

REACTIVITY OF AN AGENT

Reactivity

Agents are situated in an environment, (physical world, a user via a GUI, a collection of other agents, Internet, or perhaps many of these combined), are able to perceive this environment (through the use of potentially imperfect sensors), and are able to respond in a timely fashion to changes that occur in it;

- Environment static $\Rightarrow$ the program can execute itself blindly.
- Real world as a lot of systems are highly dynamic: constantly changing, partial/incomplete information
- Design software in dynamic environment is difficult: failures, changes, etc.
- A reactive system perceives its environment and responds in a timely appropriate fashion to the changes that occur in this environment (Event-directed).

PRO-ACTIVITY OF AN AGENT

Pro-activity

Agent do not simply act in response to their environment, they are able to exhibit goal-directed behavior by taking the initiative; They pursue their own personal or collective goals.

- Reactivity is limited (e.g. Stimulus $\Rightarrow$ Response).
- A proactive system generates and attempts to capture objectives, it is not directed only by events, take the initiative.
- Recognize/Identify opportunities to act/trigger something.

SOCIABILITY OF AN AGENT

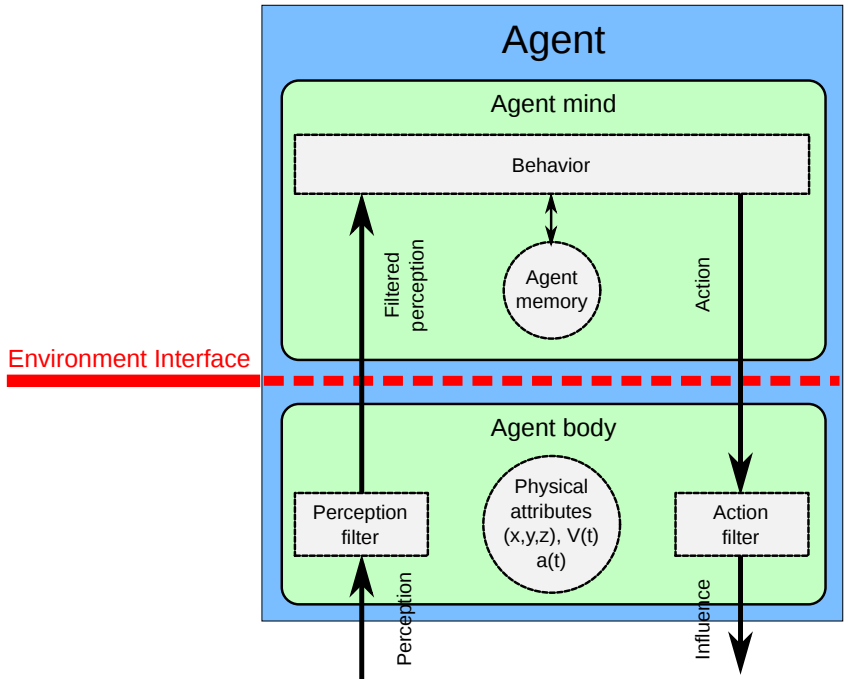
Sociability - Social Ability

Agents interact with other agents (and possibly humans), and typically have the ability to engage in social activities (such as cooperative problem solving or negotiation) in order to achieve their goals. Unity is strength.

- Many tasks can only be done by cooperating with others
- An agent must be able to interact with virtual or/and real entities
- Require a mechanism to exchange information either directly (Agent-to-Agent) or indirectly (through the environment).
- May require a specific (agent-communication) language.

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.



More details are given in Chapter #2

OTHER AGENT'S PROPERTIES

- Mobility: agent's ability to move through different nodes of a network/grid.
- Adaptability: ability to modify his actions/behavior according to external conditions and perceptions.
- Versatility: ability to perform different tasks or to meet different objectives.
- Trustiness: level of confidence that inspires the agent to delegate tasks, perform action, collaborate with other agents.
- Robustness: ability to continue to operate in fault situations, even with lower performances
- Persistence: Ability to keep continuously running by retrieving or saving their internal state even after a crash or unexpected situations.
- Altruism: disposition of an agent to assist other agents in their tasks.

AGENT: ANOTHER DEFINITION

Agent (Ferber, 1999)

Agent is a virtual (software) or physical entity which:

- is capable of acting in an environment;
- can communicate directly with other agents;
- is driven by a set of tendencies (in the form of individual objectives or of a satisfaction/survival function which it tries to optimize);
- possesses resources of its own;
- is capable to perceive its environment (but up to a limited extent);
- has only a partial representation of this environment (and perhaps none at all);
- possesses skills and can offer services; Add a comment to this line
- may be able to reproduce itself;
- whose behavior tends towards satisfying its objectives, taking account of the resources and skills available to it and depending on its perception, its representation and the communication it receives.

AGENT TYPOLOGY

- Reactive: Each agent has a mechanism of **reaction to events**, without having an explanation/understanding of the objectives nor planning mechanisms. Typical Example: The ant colony.
- Cognitive/Deliberative: Each agent has a **knowledge** base that contains all information and skills necessary for the accomplishment of their tasks/goals and the management of his interactions with other agents and his environment: reasoning, planning, normative. Typical Example: Multi-Expert Systems.

AGENT VS OBJECT

- Agents are autonomous, they decide on the execution of services: "*Objects do it for free; agents do it because they want to*".

- The agents allow flexible (reactive, pro-active, social) and autonomous behavior.

- Agent-based systems are inherently distributed and/or parallel ("multi-thread"). Each agent has its own resource of execution.

Wooldridge (2001)

AGENT VS EXPERT SYSTEM

Expert System Principle

An Expert System (ES) maintains various facts about the world that are used to draw conclusions.

Key Differences

- Traditional ES are not situated in an environment. There is no direct coupling with the environment and requires a user that acts as an intermediary ue.
- ES are usually not capable of exhibiting flexible behavior (reactive and proactive).
- ES are usually not provided with social skills (communication and interaction with other agents).

TWO PERSPECTIVES ON MAS

Mono-agent approach

- The system is composed of a single agent.
- Example: Personal Assistant

Multi-agent approach

- The system is composed of multiple agents.
- The realization of global/collective task relies on a set of agents, on the composition of their actions.
- The solution emerges from the interactions of agents in an environment.

Multiagent systems

An MultiAgent Systems (MAS) is a system composed of agents that interact together and through their environment.

Interactions:

→ Direct, agent to agent

→ Indirect, Stigmergy, through the Environment

MULTIAGENT SYSTEMS: FROM LOCAL TO GLOBAL

Micro perspective (local): Agent

Individual level

- Reactivity - Pro-activity
- Autonomy
- Delegation

Macro perspective (global): Multiagent systems

Society/Community level

- Distribution
- Decentralization (control and/or authority)
- Hierarchy
- Agreement technologies (coordination)
- Emergence, social order/pattern, norms

Multiagent Systems (Ferber, 1999)

System comprising the following elements:

- An environment E, usually a space (with volume, 3D).
- An array of objects, O. These objects are situated.
- A set of agents, A, which are specific objects.
- A set of relations, R, which links the objects (and thus agents).
- A set of operations, Op, making it possible for agents to receive, produce, process and manipulate the objects in O.
- Operators with the task of representing the application of these operations and the reaction of the world to this attempt of modification.

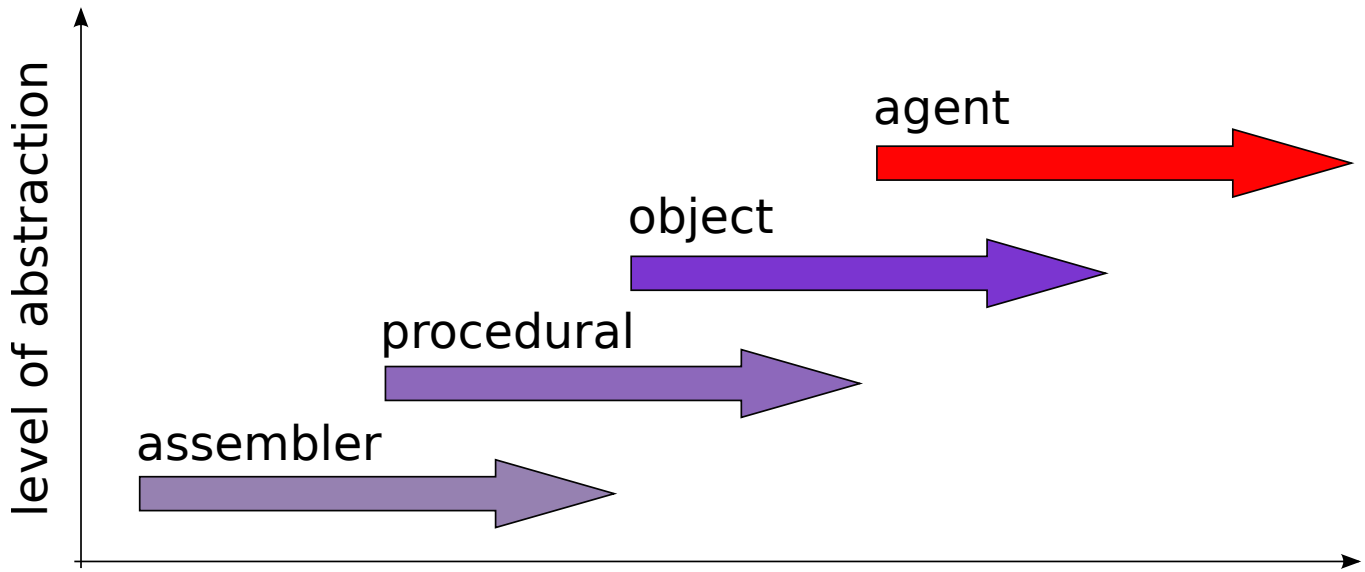
MAS INTERESTS

What is the interest of MAS? (Wooldridge, 2001)

- Natural metaphors
- Distribution of Data or Control
- Legacy Systems: Wrap/Encapsulate these systems into an agent for enabling interactions
- Open Systems requires flexible autonomous decision making
- Complex Systems

Agent: a new paradigm ?

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



DESIGN PRINCIPLES OF SARL

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

Name	Domain	Hierar.[a]	Simu.[b]	C.Phys.[c]	Lang.	Beginners[d]	Free
GAMA	Spatial simulations	✓	✓		GAML, Java	**[*]	✓
Jade	General		✓	✓	Java	*	✓
Jason	General		✓	✓	Agent-Speaks	*	✓
Madkit	General		✓		Java	**	✓
NetLogo	Social/ natural sciences		✓		Logo	***	✓
Repast	Social/ natural sciences		✓		Java, Python, .Net	**	
SARL	General	✓	✓[e]	✓	SARL, Java, Xtend, Python	**[*]	✓

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

This table was done according to experiments with my students and ?

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)

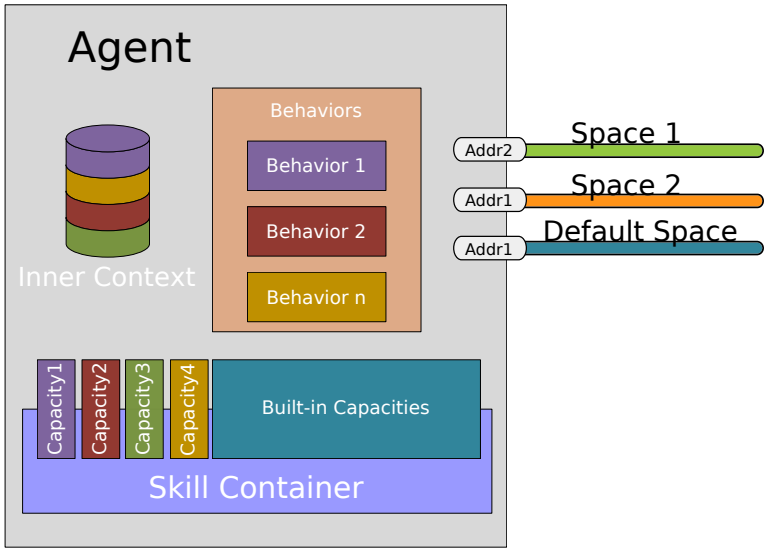
SARL: a general-purpose agent-oriented programming language. Rodriguez, S., Gaud, N., Galland, S. (2014) Presented at the The 2014 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, IEEE Computer Society Press, Warsaw, Poland. (Rodriguez, 2014)

http://www.sarl.io

AGENT

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.




```
agent HelloAgent {
  on Initialize {
   println("Hello World!")
  }
  on Destroy {
   println("Goodbye World!")
  }
}
```

EXAMPLE OF AGENT CODE


```
package org.multiagent.example

agent HelloAgent {

  var myvariable : int
  val myconstant = "abc"

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

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

}
```

The content of the file will be assumed to be in the given package.

EXAMPLE OF AGENT CODE


```
package org.multiagent.example

agent HelloAgent {

  var myvariable : int
  val myconstant = "abc"

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

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

}
```

Define the code of all the agents of type HelloAgent

EXAMPLE OF AGENT CODE


```
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.

EXAMPLE OF AGENT CODE


```
package org.multiagent.example

agent HelloAgent {

  var myvariable : int
  val myconstant = "abc"

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

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

}
```

Define a variable with name "myvariable" and of type integer

EXAMPLE OF AGENT CODE


```
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.

EXAMPLE OF AGENT CODE


```
package org.multiagent.example

agent HelloAgent {

  var myvariable : int
  val myconstant = "abc"

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

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

}
```

Execute the block of code when an event of type "Initialize" is received by the agent.

EXAMPLE OF AGENT CODE


```
package org.multiagent.example

agent HelloAgent {

    var myvariable : int
    val myconstant = "abc"

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

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

}
```

Events predefined in the SARL language:
- When initializing the agent
- When destroying the agent

CAPACITY AND SKILL

Action

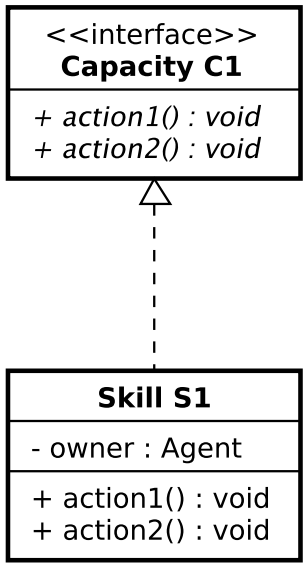
- A specification of a transformation of a part of the designed system or its environment.
- Guarantees resulting properties if the system before the transformation satisfies a set of constraints.
- Defined in terms of pre- and post-conditions.

Capacity

Specification of a collection of actions.

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


```
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!")
  }

}
```

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!")
  }

}
```

Define a function that could be invoked by the agent.

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)
  }

}
```

Define the skill that implements the Logging capacity.

```
  uses Logging

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

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

}
```

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)
  }

}
```

Every function declared into the implemented capacity must be implemented in the skill. The current implementations output the message onto the standard output stream.

```
                new BasicConsoleLogging)
    info("Hello World!")
  }

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

}
```

EXAMPLE OF CAPACITY AND SKILL

The use of a capacity into the agent code is enabled by the "uses" keyword.

```
capaci

  def

  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!")
  }

}
```

EXAMPLE OF CAPACITY AND SKILL


```
capacity Logging {

  def debug(s : String)

  def info(s : String)

}
```

```
ski

  d
    println("DEBUG:" + s)
  }

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

}
```

All functions defined into the used capacities are directly callable from the source code.

```
agent HelloAgent {

  uses Logging

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

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

}
```

EXAMPLE OF CAPACITY AND SKILL


```
capacity Logging {

  def debug(s : String)

  def info(s : String)

}

ski

  d

  }

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

}
```

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

```
agent HelloAgent {

  uses Logging

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

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

}
```

SPACE: 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.

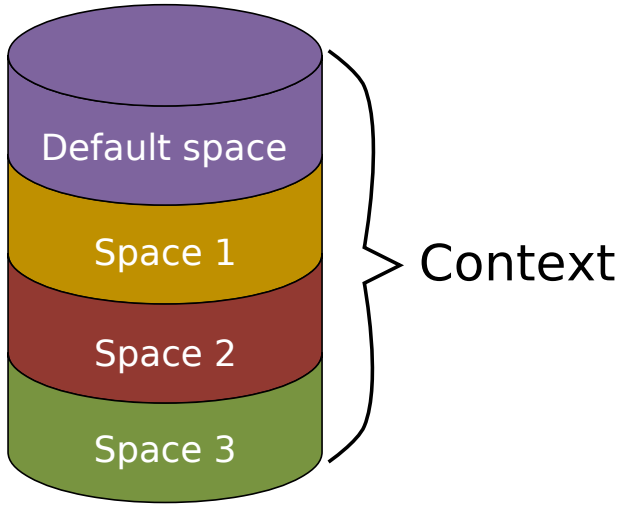
.

The spaces and space specifications must be written with an Object-Oriented Programming language, e.g. Java; or with the basic OO statements within SARL

CONTEXT AND SPACES

Context

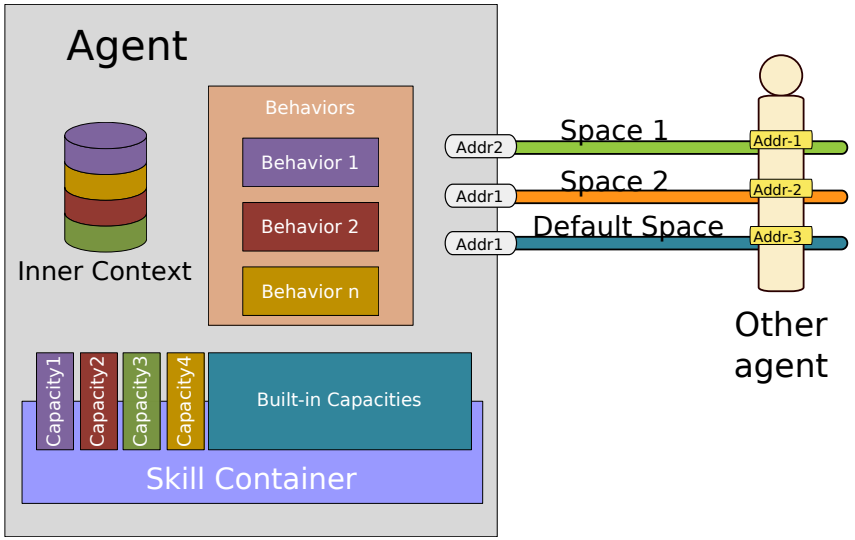
- Defines the boundary of a sub-system.
- Collection of Spaces.
- Every Context has a Default Space.
- Every Agent has a Default Context, the context where it was spawned.



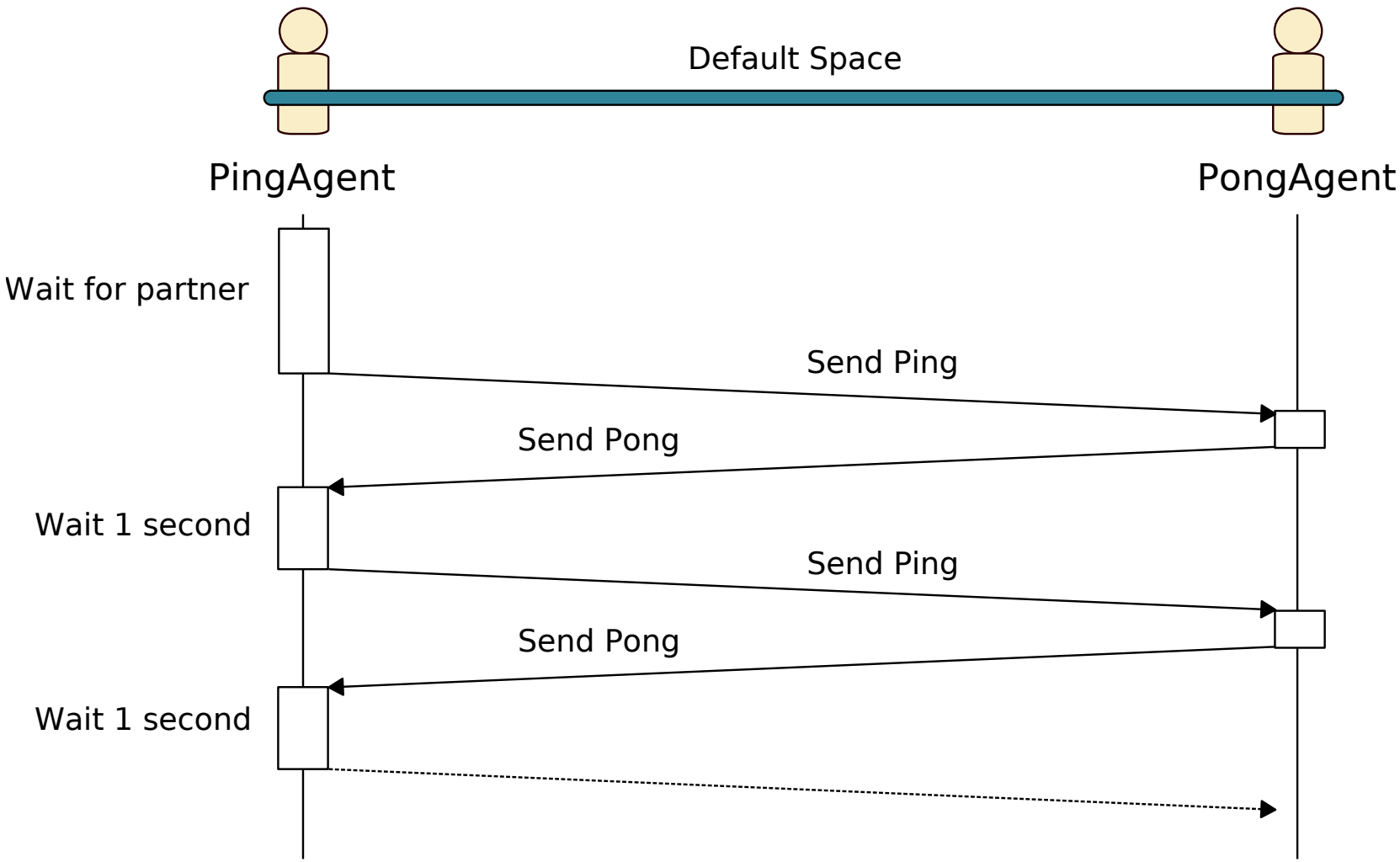
DEFAULT SPACE IN EVERY CONTEXTS

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



EXAMPLE OF INTERACTIONS: PING - PONG

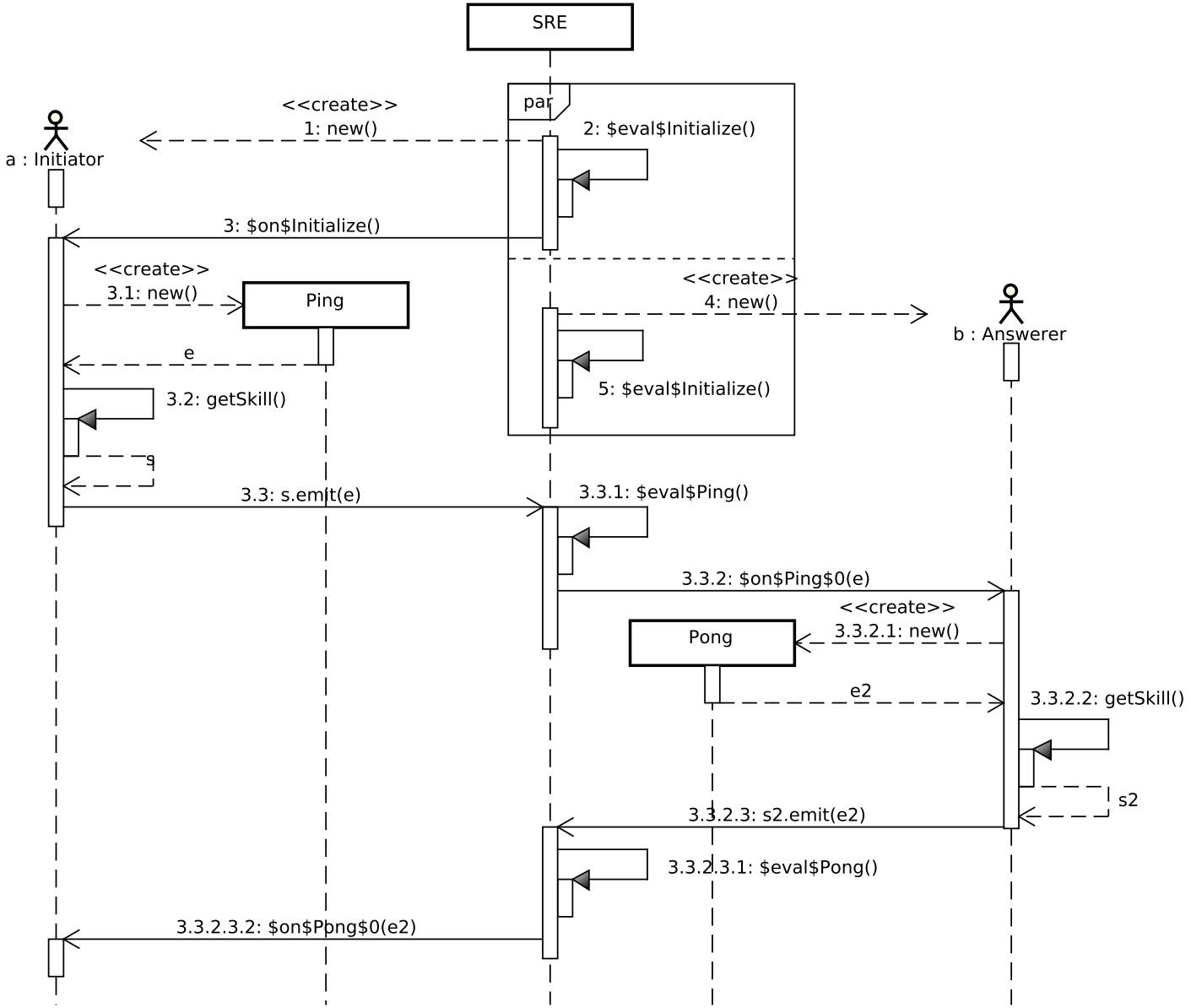

```
event Ping {
  var value : Integer
  new (v : Integer) {
   value = v
  }
}

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))
  }
}
```

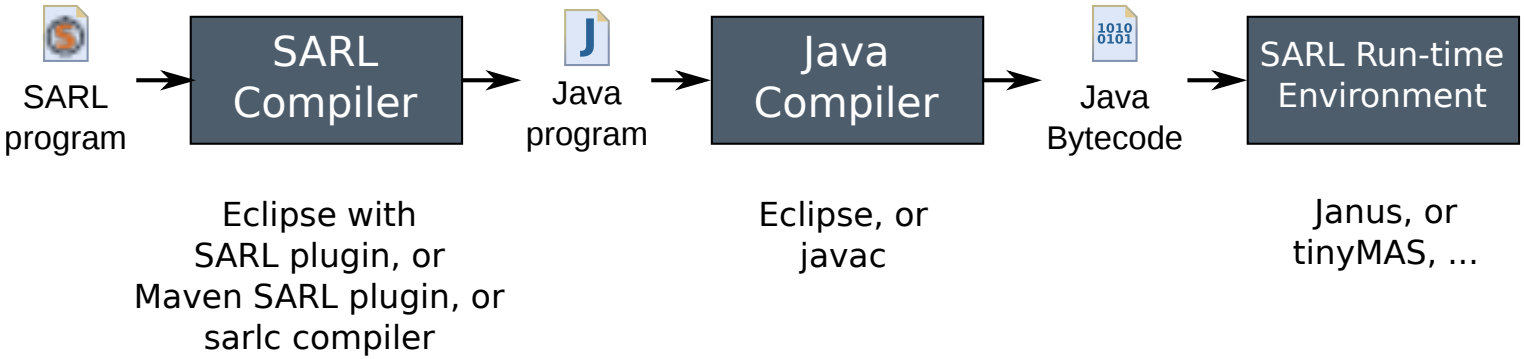
```
agent PingAgent {
  uses Schedules
  uses DefaultContextInteractions
  var count : Integer
  on Initialize {
   println("Starting PingAgent")
   count = 0
   in(2000) [ sendPing ]
  }
  def sendPing {
   if (defaultSpace.
         participants.size > 1) {
     emit(new Ping(count))
     count = count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

EXECUTION SEQUENCE DIAGRAM



COMPATIBILITY BETWEEN SARL AND JAVA

SARL is 100% compatible with Java



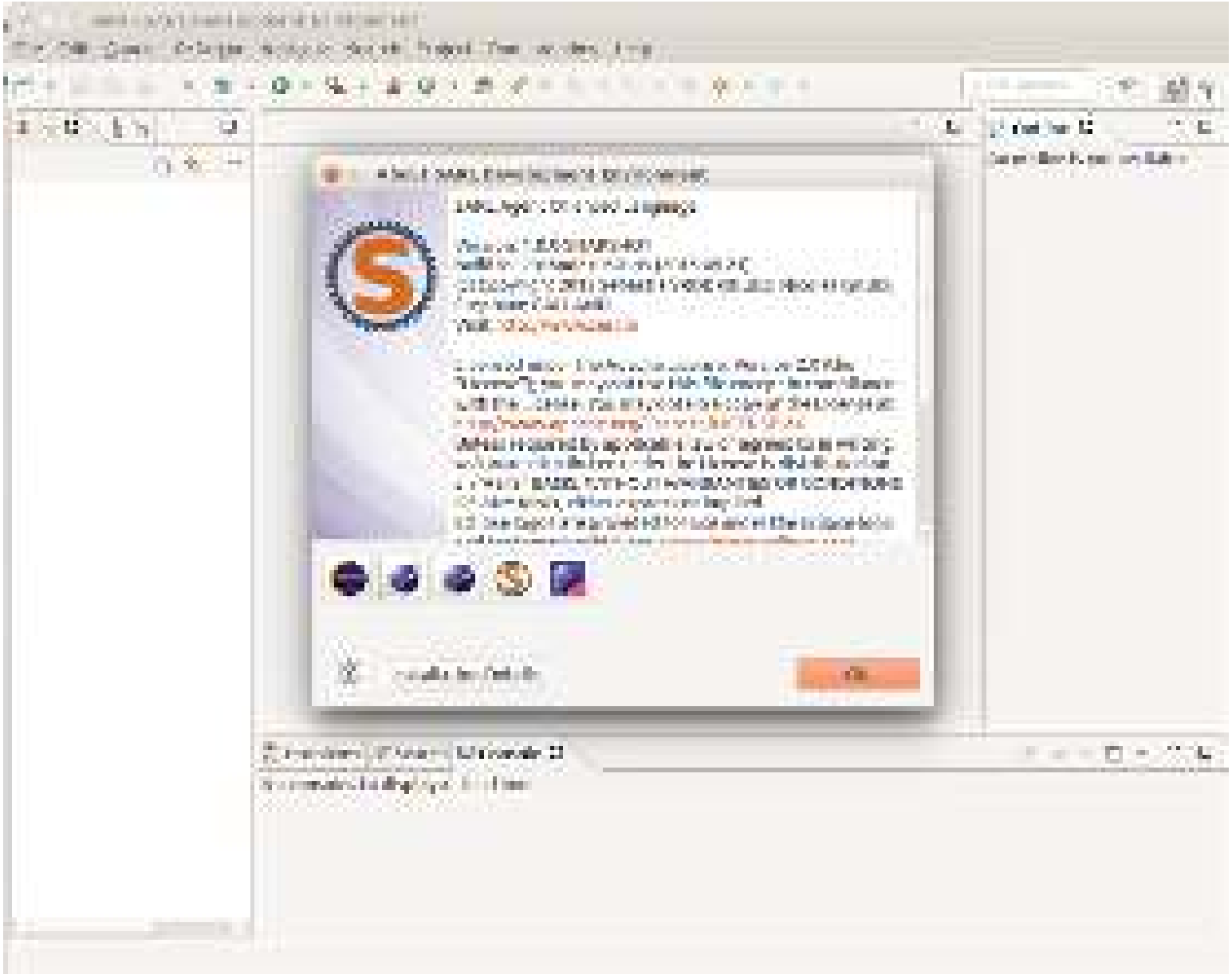
- Any Java feature or library could be included and called from SARL.
- A Java application could call any public feature from the SARL API.

DOCUMENTATION OF SARL

The SARL syntax is explained into the "General Syntax Reference" on the SARL website.

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

SARL IN THE ECLIPSE IDE



RUNTIME ENVIRONMENT FOR SARL

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 created. See Chapter #5.

OUTLINE

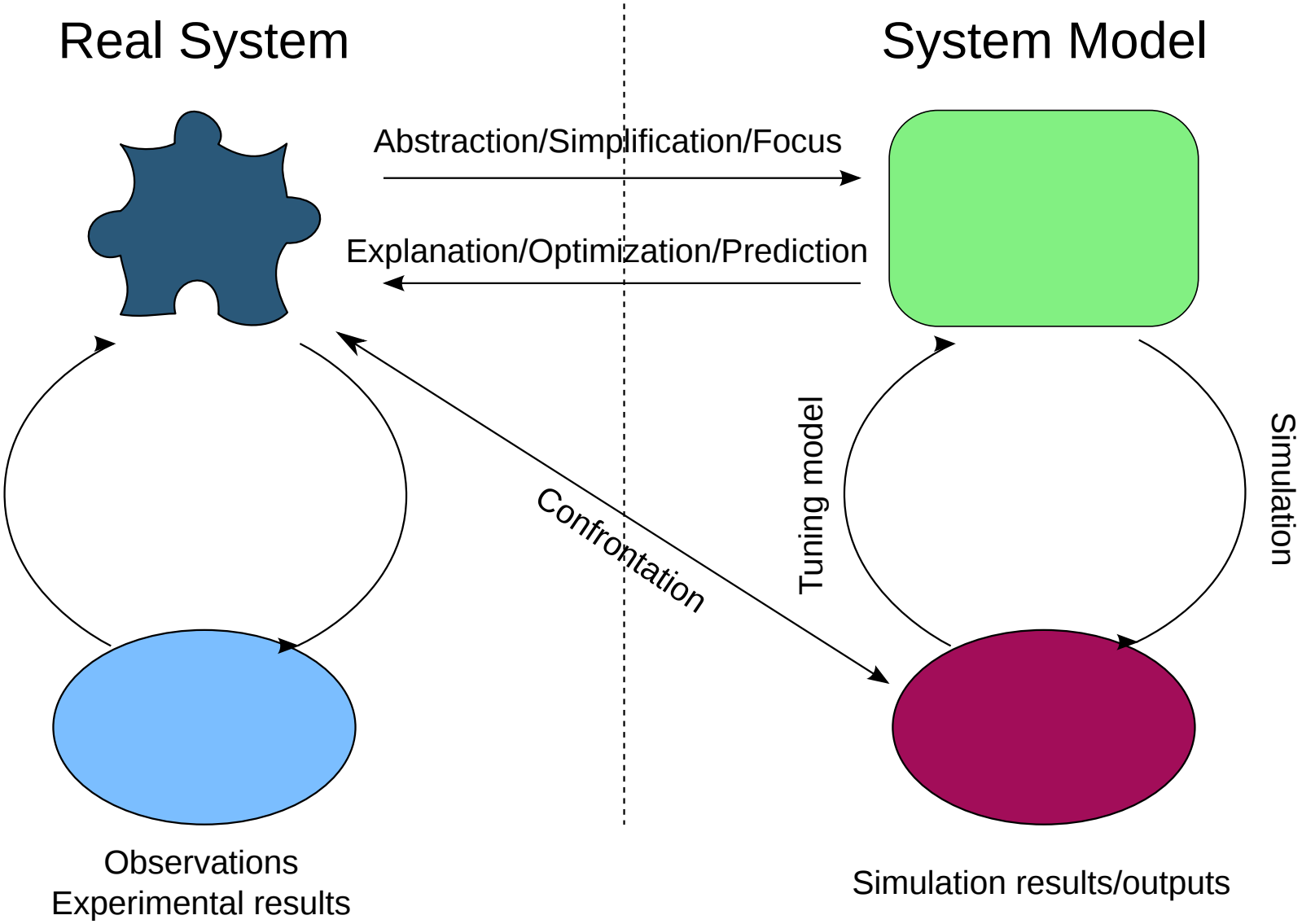
DEFINITION OF SIMULATION

(Shannon, 1977)

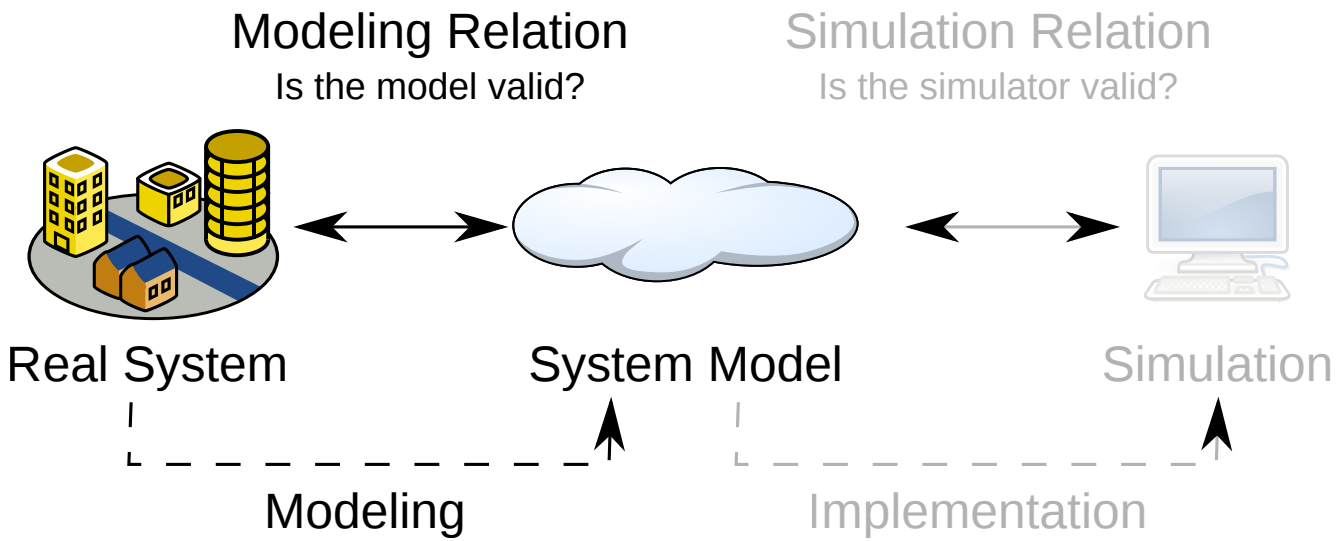
The process of designing a model of a real system and conducting experiments with this model for the purpose either of understanding the behavior of the system or of evaluating various strategies (within the limits imposed by a criterion or a set of criteria) for the operation of the system.

Why simulate?

- Understand / optimize a system.
- Scenarii/strategies evaluation, testing hypotheses to explain a phenomenom (decision-helping tool).
- Predicting the evolution of a system, e.g. metrology.


Real System
System Model
Abstraction/Simplification/Focus
Explanation/Optimization/Prediction
Confrontation
Tuning model
Simulation
Observations
Experimental results
Simulation results/outputs

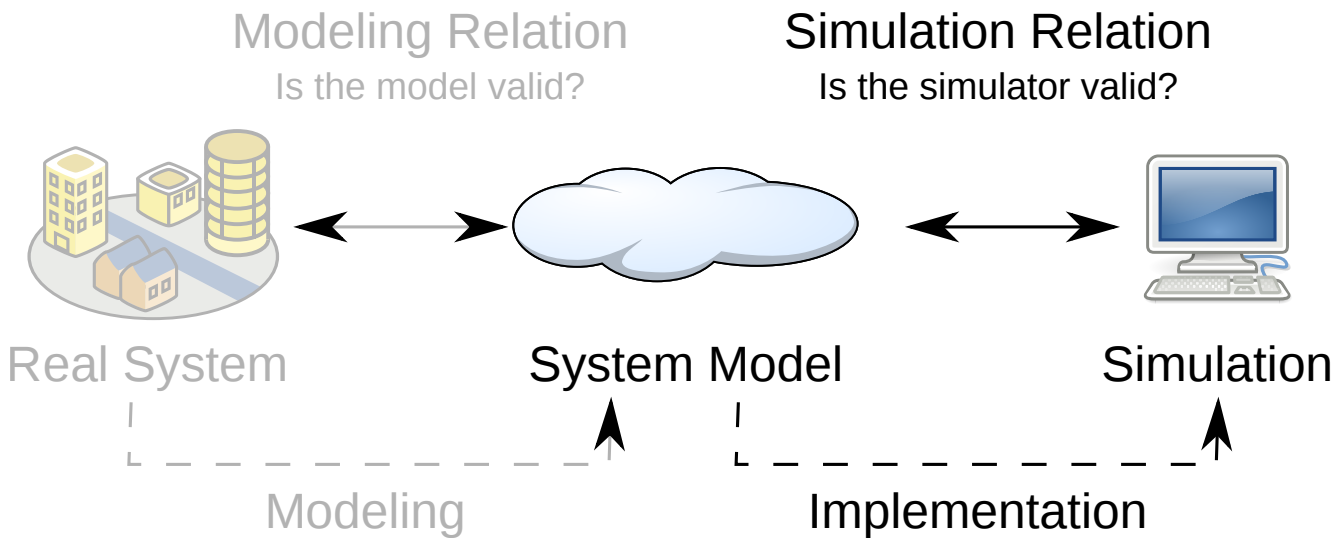
MODELING RELATION: SYSTEM ↔ MODEL



- To determine if the system model is an acceptable simplificiation in terms of quality criteria and experimentation objectives.
- Directly related to the consistency of the model simulation.

Zeigler (2000)

SIMULATION RELATION: MODEL ↔ SIMULATOR



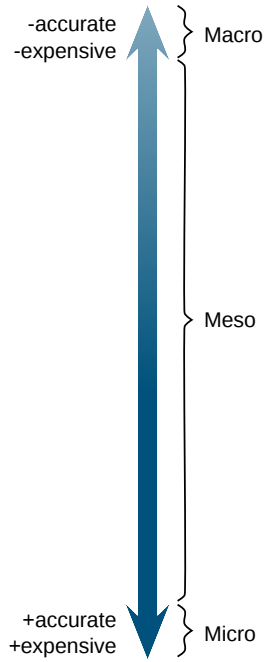
- To guarantee that the simulator, used to implement the model, correctly generates the behavior of the model.
- To be sure that the simulator reproduces clearly the mechanisms of change of state are formalized in the model.

Zeigler (2000)

CLASSICAL TYPOLOGY (1/4)

Microscopic Simulation

- Explicitly attempts to model the behaviors of each individual.
- The system structure is viewed as emergent from the interactions between the individuals.

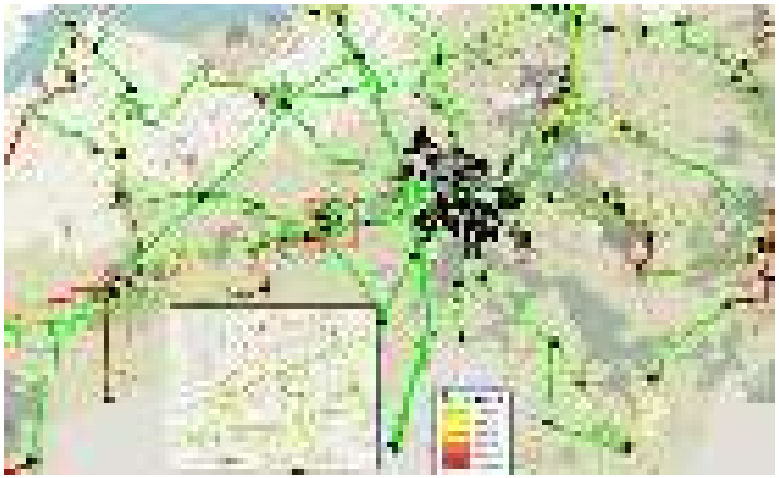


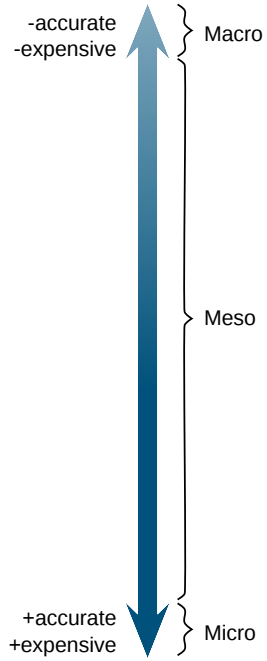
Hoogendoorn (2001); Davidsson (2000)

CLASSICAL TYPOLOGY (2/4)

Mesoscopic Simulation

- Based on small groups, within which elements are considered homogeneous.
- Examples: vehicle platoon dynamics and household-level travel behavior.



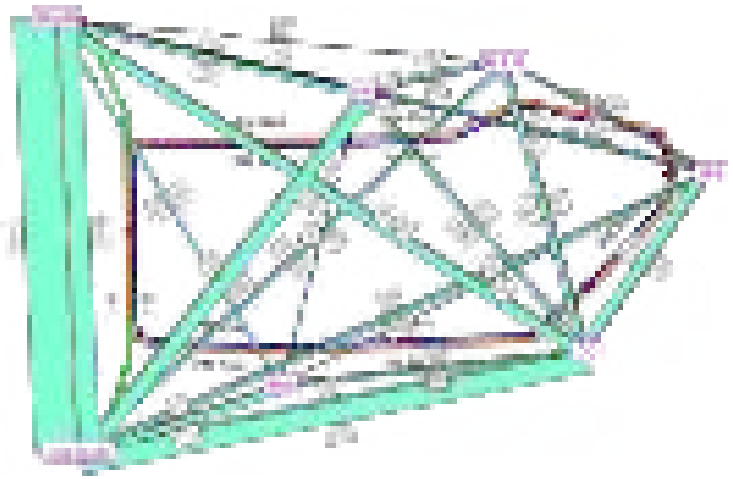


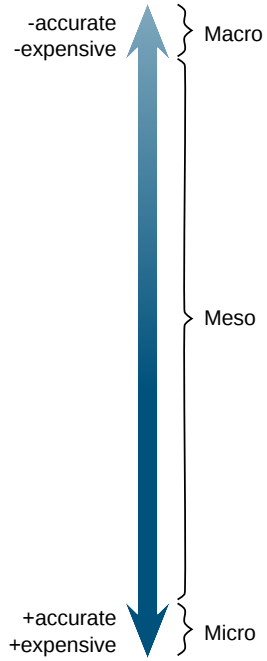
Hoogendoorn (2001); Davidsson (2000)

CLASSICAL TYPOLOGY (3/4)

Macroscopic Simulation

- Based on mathematical models, where the characteristics of a population are averaged together.
- Simulate changes in these averaged characteristics for the whole population.
- The set of individuals is viewed as a structure that can be characterized by a number of variables.



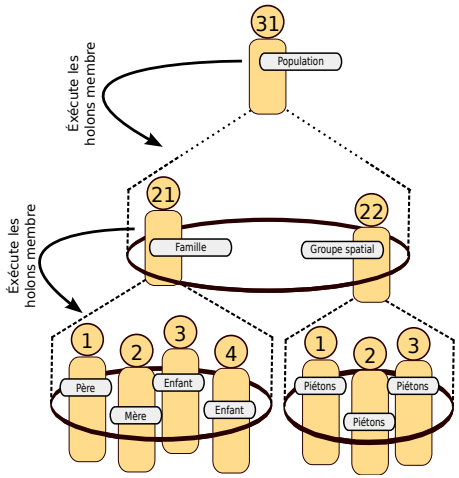


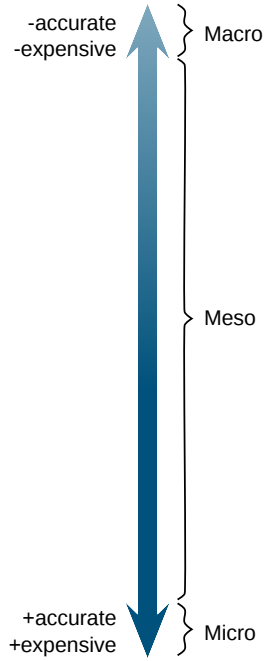
Hoogendoorn (2001); Davidsson (2000)

CLASSICAL TYPOLOGY (4/4)

Multilevel Simulation

- Combines various levels.
- Specify how the different levels are interacting together: one is input of the other, dynamic selection of the level.



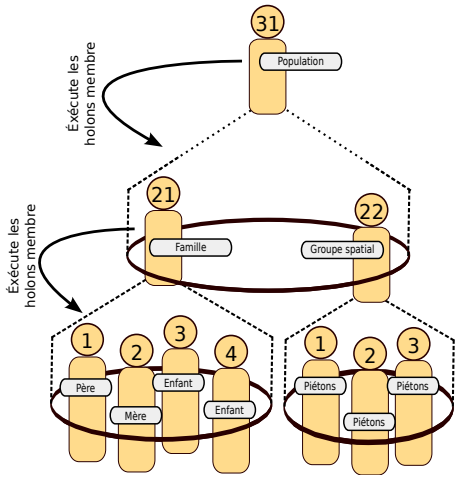


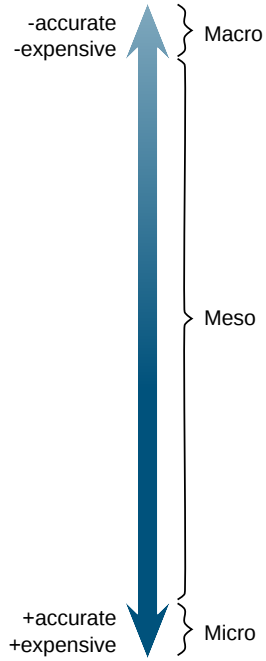
Hoogendoorn (2001); Davidsson (2000); Galland (2014)

CLASSICAL TYPOLOGY (4/4)

Multilevel Simulation

- Combines various levels.
- Specify how the different levels are interacting together: one is input of the other, dynamic selection of the level.





Multiagent-based Simulation (MABS), aka. ABS, is traditionnally considered as a special form of microscopic simulation, but not restricted to.

MABS: GENERAL IDEA

- Create an artificial world composed of interacting agents.
- The behavior of an agent results from:
  - its perceptions/observations;
  - its internal motivations/goals/beliefs/desires;
  - its eventual representations;
  - its interaction with the environment (indirect interactions, ressources) and the other agents (communications, direct interactions, stimuli).
- Agents act and modify the state of the environment through their actions.
- We observe the results of the interactions like in a Virtual Lab $\Rightarrow$ Emergence.

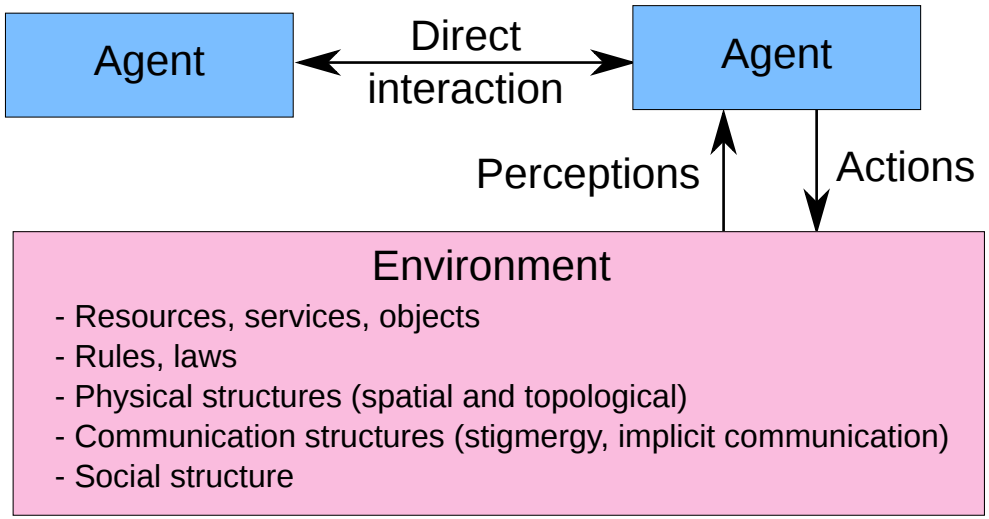
MABS: MAIN CHARACTERISTICS AND ADVANTAGES

- More flexible than macroscopic models to simulate spatial and evolutionary phenomena.
- Dealing with real multiagent systems directly: real Agent = simulated Agent.
- Allows modelling of adaptation and evolution.
- Heterogeneous space and population.
- Multilevel modeling: integrate different levels of observation, and of agent's behaviors.

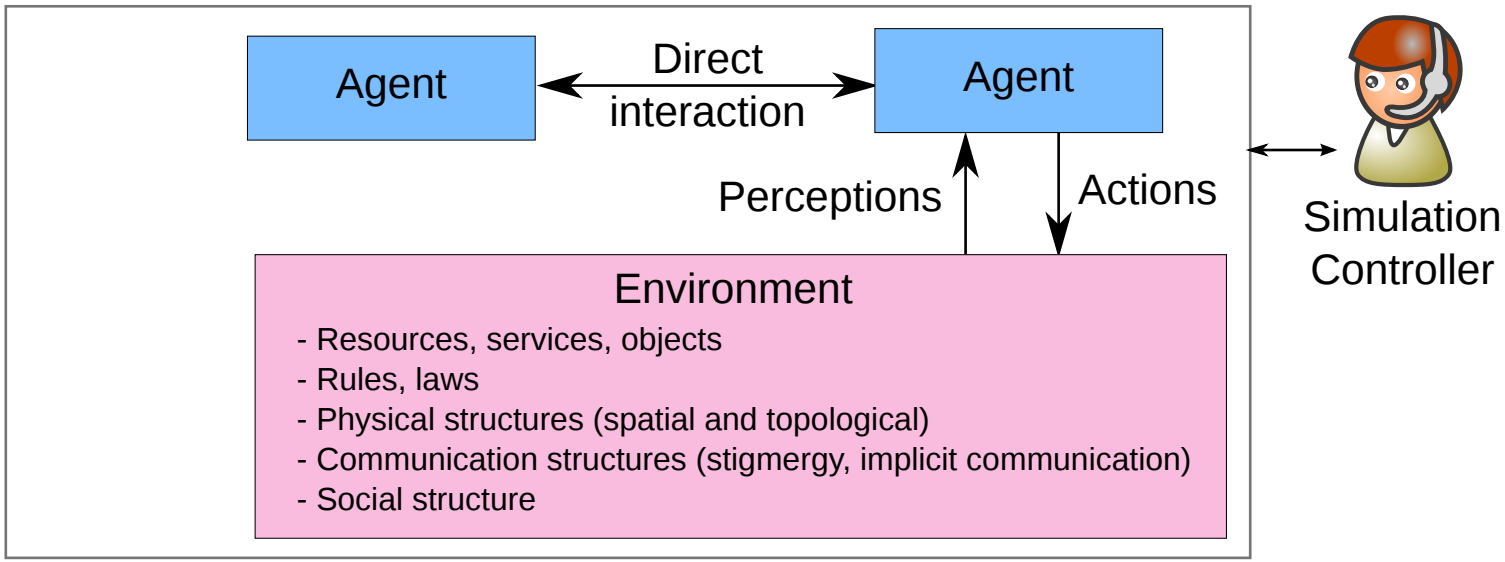
MABS: LIMITATIONS AND DRAWBACKS

- Offer a significant level of accuracy at the expense of a larger computational cost.
- Require many and accurate data for their initialization.
- It is difficult to apply to large scale systems.
- Actual simulation models are costly in time and effort.

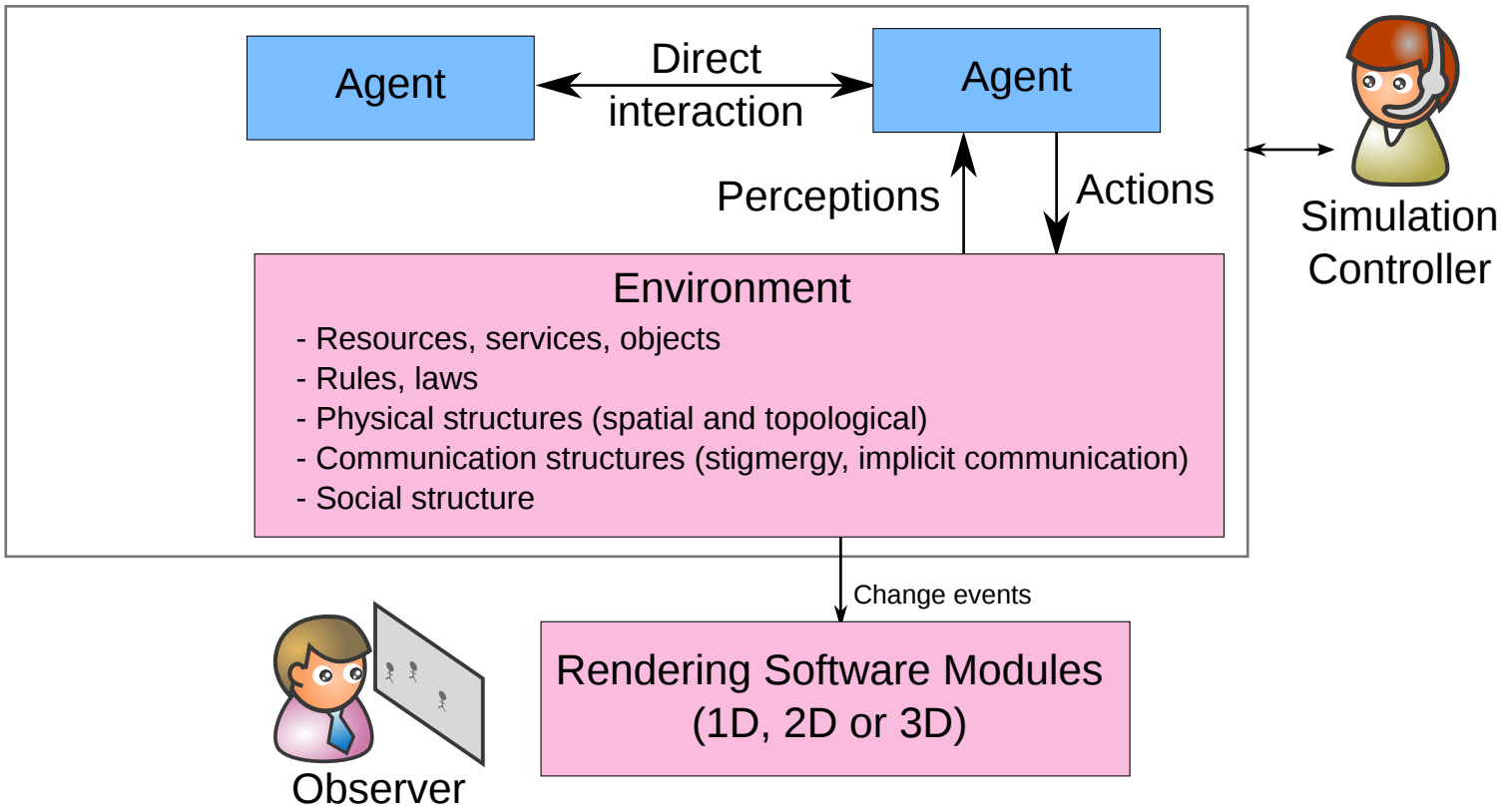
GENERAL ARCHITECTURE



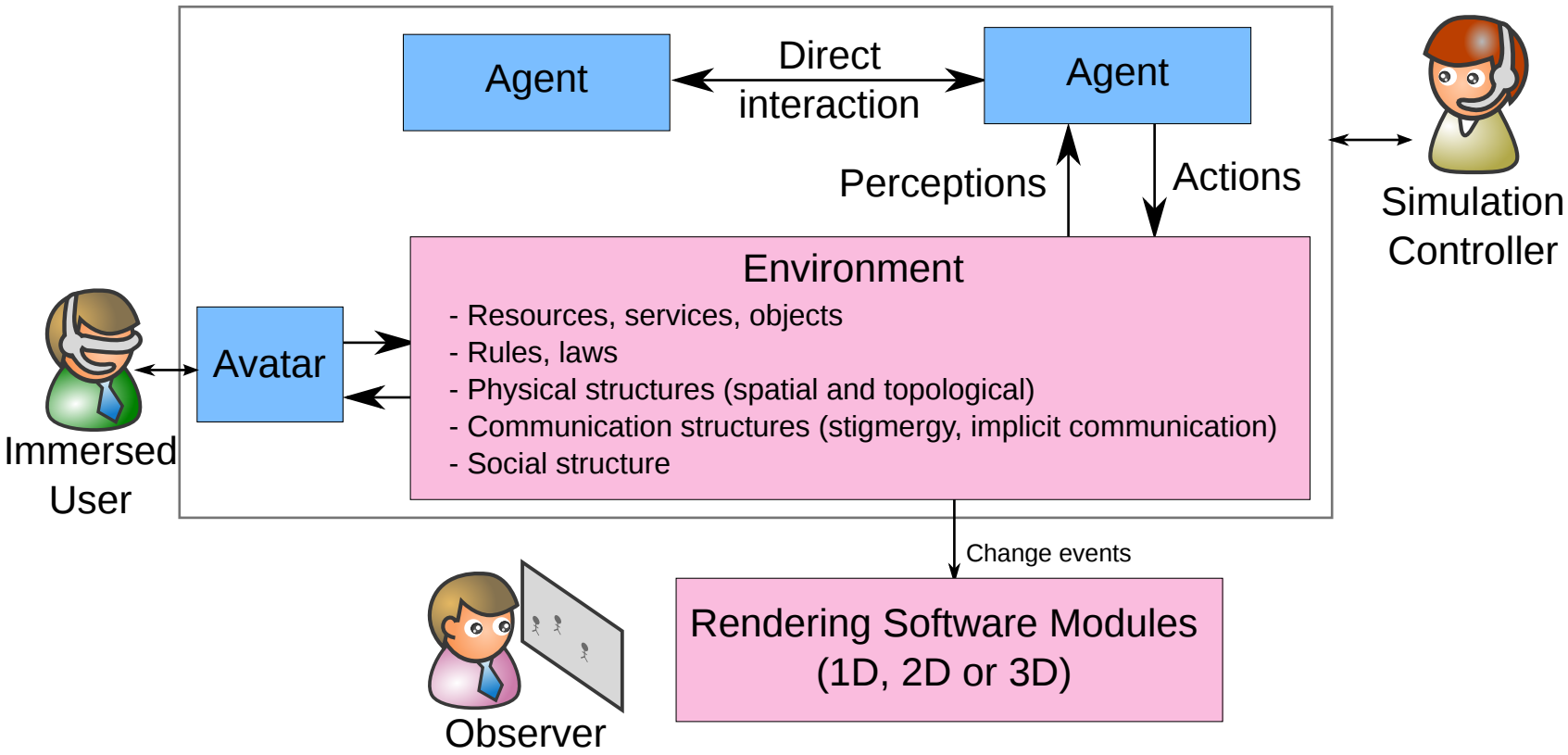
GENERAL ARCHITECTURE



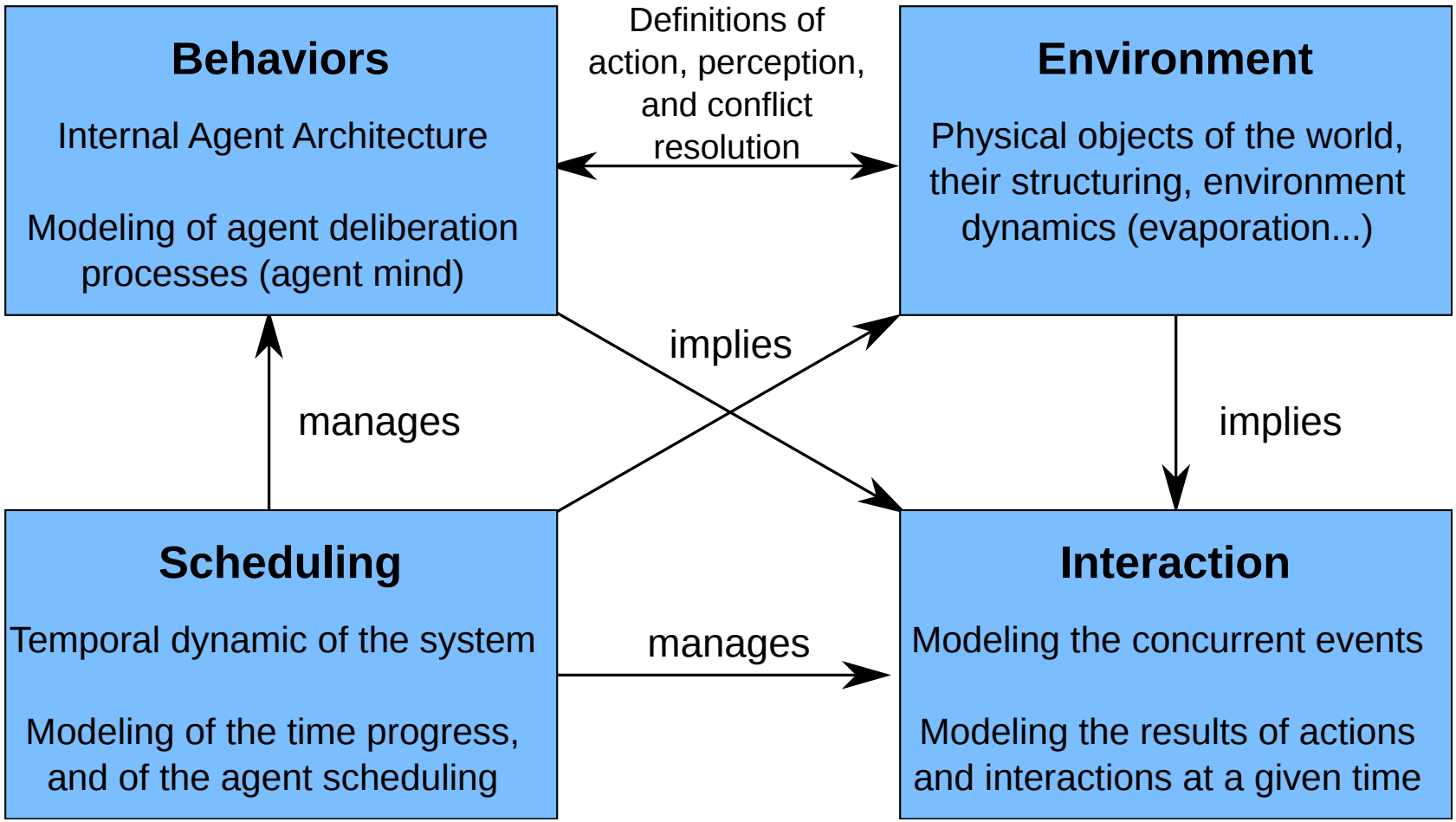
GENERAL ARCHITECTURE



GENERAL ARCHITECTURE



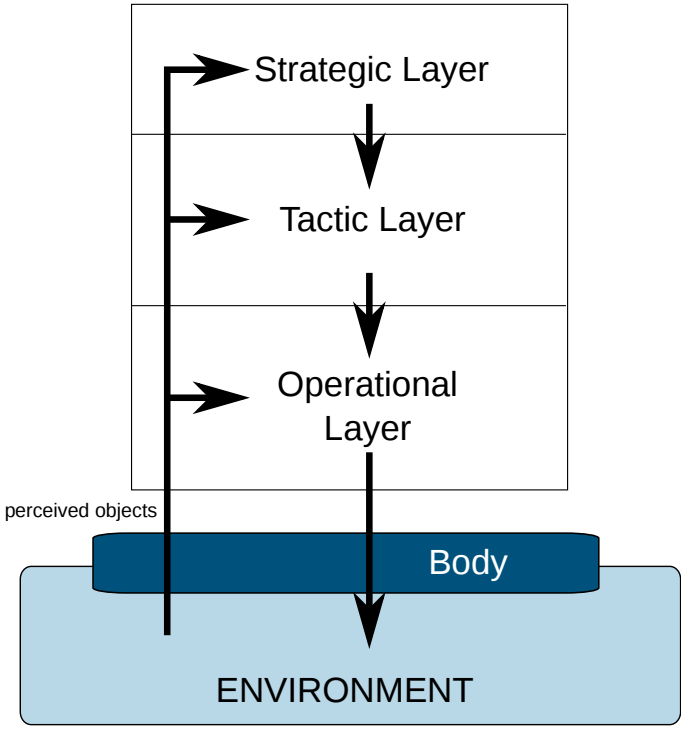
DESIGNING A MULTIAGENT SIMULATION MODEL



Michel (2004)

THREE-LAYER ARCHITECTURE

- Strategic Layer: general planning stage that includes the determination of goals, the route and the modal choice as well as a cost-risk evaluation.
- Tactic Layer: Maneuvering level behavior. Examples: obstacle avoidance, gap acceptance, turning, and overtaking.
- Operational Layer: Fundamental body controlling processes such as controlling speed, following the path, etc.

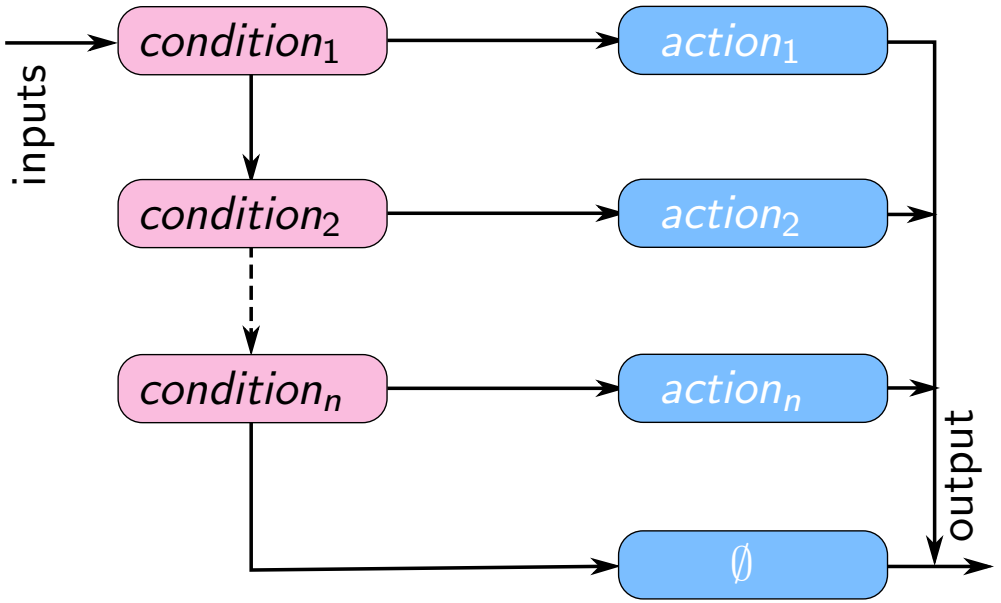


Michon (1985); Hoogendoorn (2001, 2004); Montello (2005)

SUBSOMPTION AGENT ARCHITECTURE

- Priority-ordering sequence of condition-action pairs.
- Generalization of the three-layer architecture: each pair is a layer.

if $condition_1$ **then**
    $action_1$
else
    if $condition_2$ **then**
        $action_2$
    else
        ...
    end if
end if

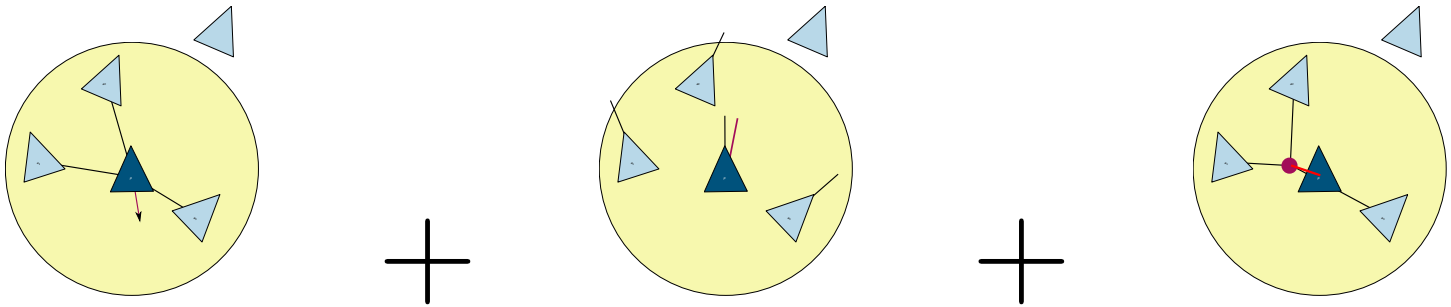


Brooks (1990)

(SOCIAL) FORCE MODELS

- Repulsive forces are computed and summed for obtaining the safer direction.
- May contribute to the too lower layers of the three-layer architecture.

$$\vec{f} = \left(\sum_{i=1}^{n} \frac{\alpha_i . \widehat{p - a_i}}{|\overrightarrow{p - a_i}|} \right) + \beta . \left(\frac{\overrightarrow{\sum_{i=1}^{n} a_i}}{n} - p \right)$$

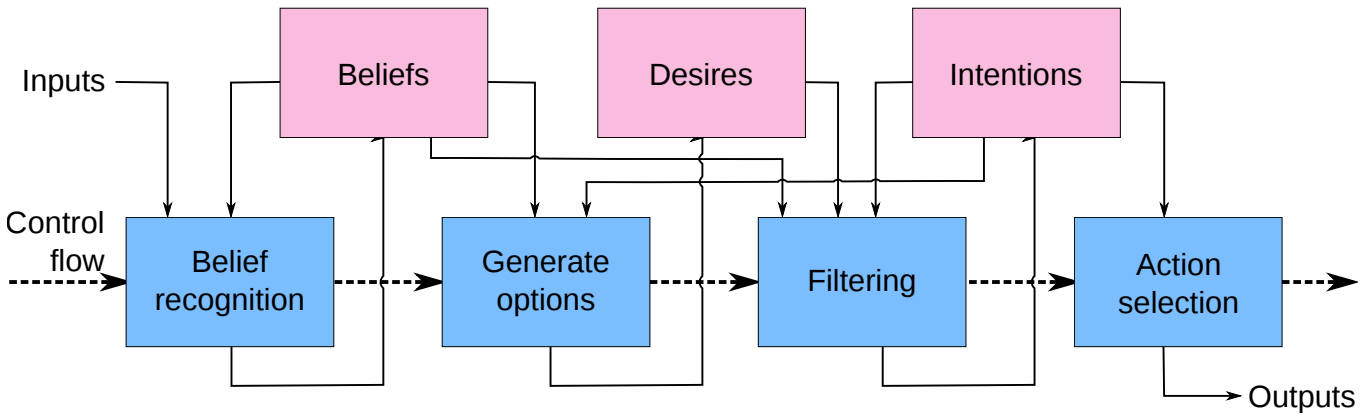


separation + alignment + cohesion

Helbing (1997); Reynolds (1999); Buisson (2013)

BDI ARCHITECTURE

- BDI: Belief Desire Intention
- Beliefs: the informational state of the agent, in other words its beliefs about the world (including itself and other agents).
- Goals: a desire that has been adopted for active pursuit by the agent.
- Intentions: the deliberative state of the agent – what the agent has chosen to do.
- Plans: sequences of actions for achieving one or more of its intentions.
- Events: triggers for reactive activity by the agent.
- BDI may contribute to the too upper layers of the three-layer architecture.



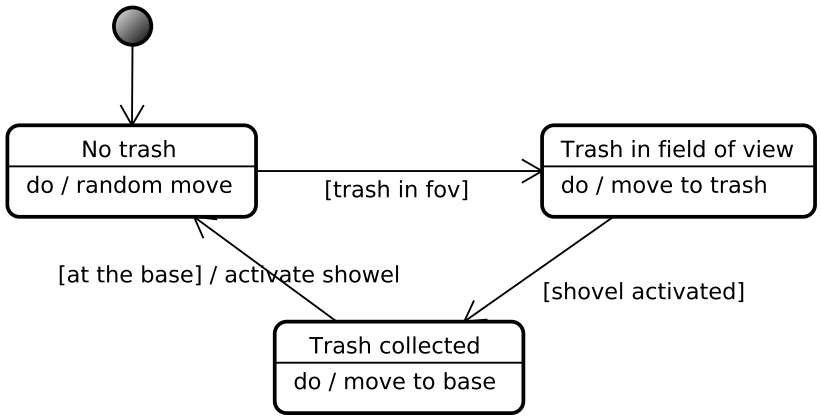
Rao (1995)

STATE-TRANSITION DIAGRAMS

- Define the behavior in terms of states (of the agent knowledge) and transitions.
- Actions may be trigerred on transitions or states.
- Markov models (Baum, 1966) or activity diagrams (Rumbaugh, 1999) may be used in place of state-transition diagrams.

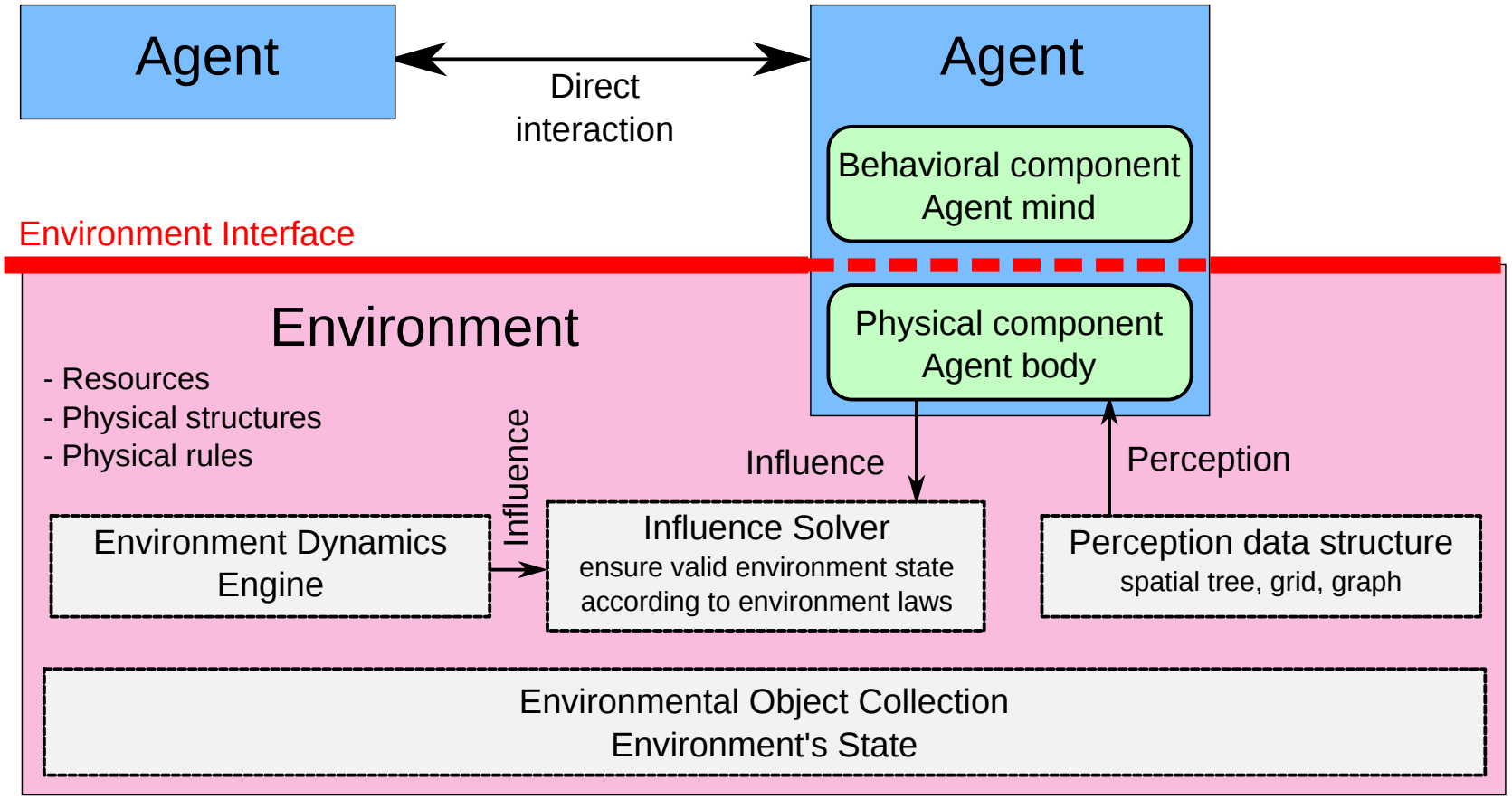
```
enum State{NO_TRASH, TRASH_IN_FOV,
     TRASH_COLLECTED }

agent A {
  var state = NO_TRASH
  on Perception
     [state==NO_TRASH &&
      occurrence.containsTrash] {
   state = TRASH_IN_FOV
  }
  on Perception
     [state==NO_TRASH &&
      !occurrence.containsTrash] {
   randomMove
  }
  //...
}
```



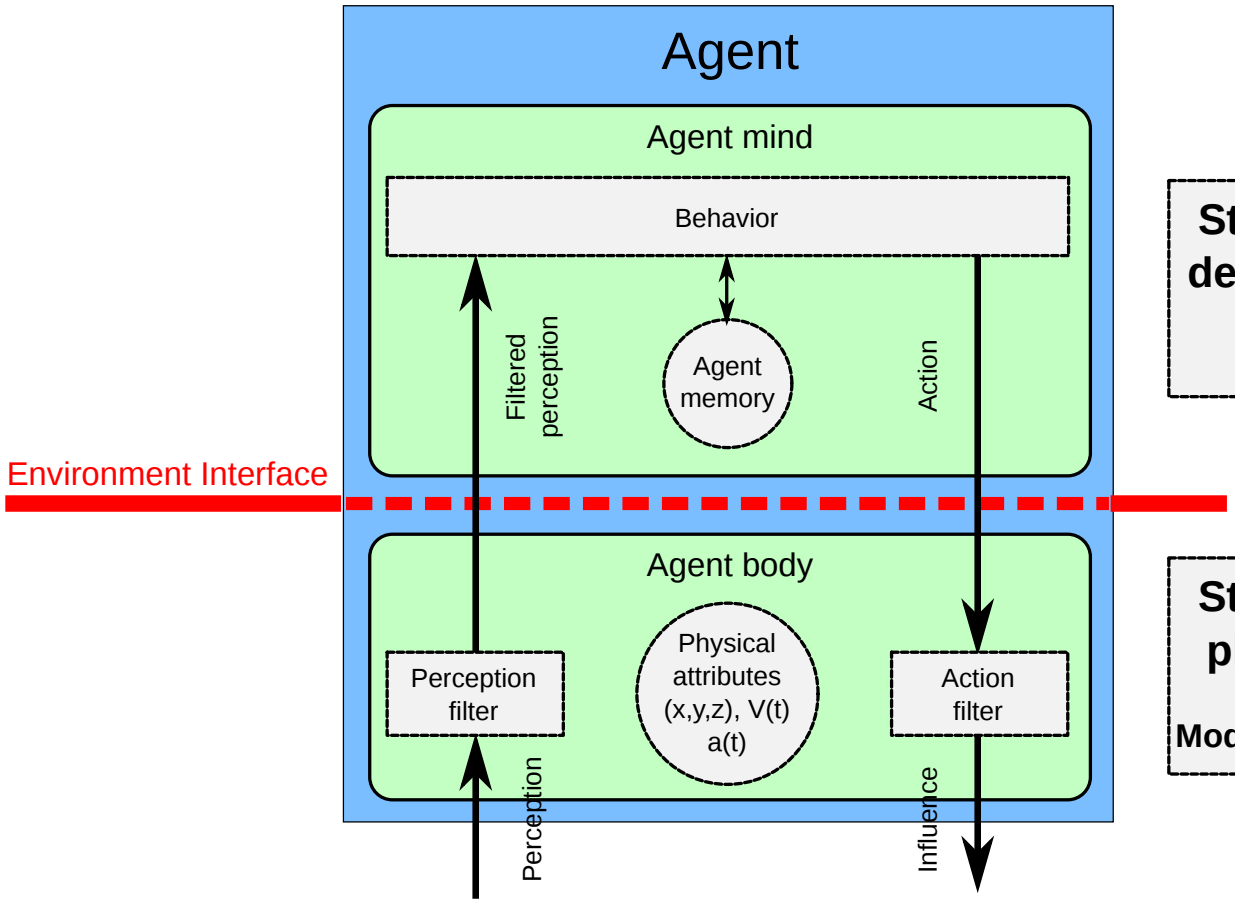
Harel (1987)

OUTLINE



Galland (2009)

BODY-MIND DISTINCTION



State variables of the decisional component
Readable/modifiable only by the agent

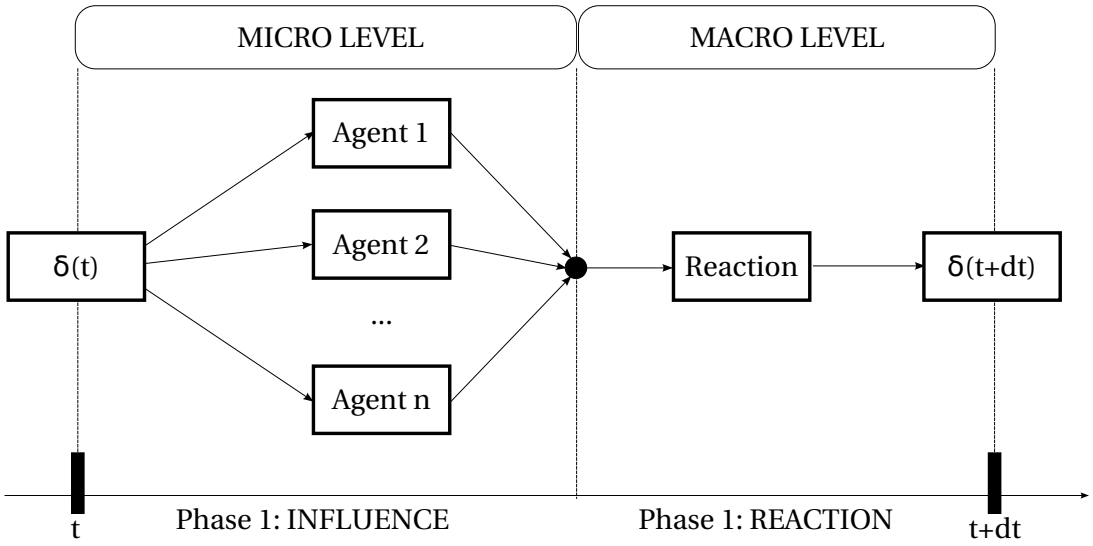
State variables of the physical component
Readable by the agent
Modifiable by the environment

Galland (2009)

SIMULTANEOUS ACTIONS: INFLUENCE-REACTION

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.

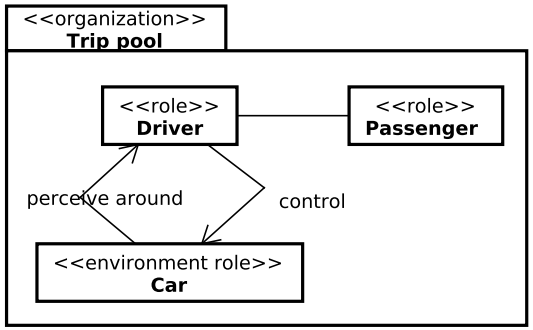


Michel (2007)

DRIVING ACTIVITY

- 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.



MODEL OF THE ENVIRONMENT

Road Network

- Road polylines: $S = \left\{ \langle path, objects \rangle \middle| path = \langle (x_0, y_0) \cdots \rangle \right\}$
- Graph: $G = \{S, S \mapsto S, S \mapsto S\} = \{\text{segments}, \text{entering}, \text{exiting}\}$

Operations

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

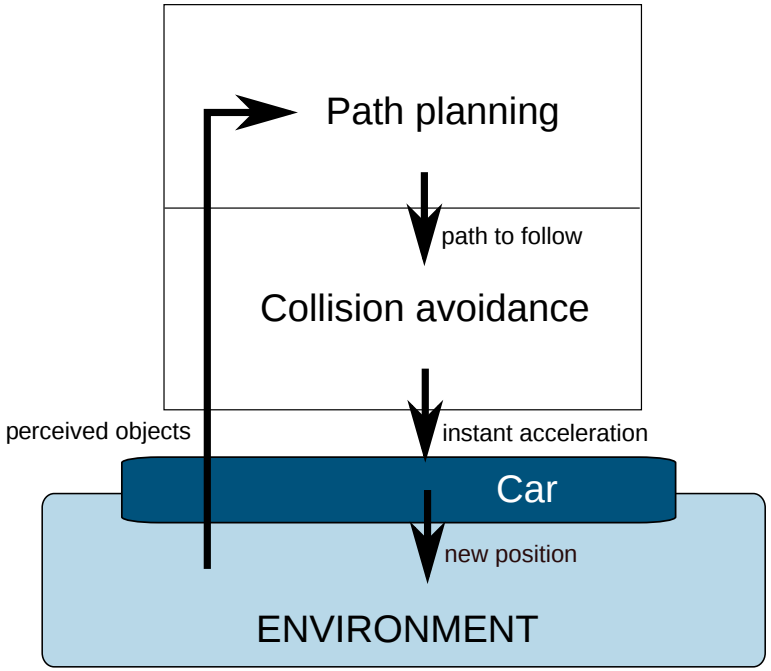
$$P = \left\{ o \middle| \begin{array}{c} distance(d, o) \leq \Delta \wedge \\ o \in O \wedge \\ \forall (s_1, s_2), path = s_1.\langle p, O \rangle.s_2 \end{array} \right\}$$

  where *path* is the roads followed by a driver d.

- Move the vehicles, and avoid physical collisions.

Galland (2009)

ARCHITECTURE OF THE DRIVER AGENT



Jasim model (Galland, 2009)

PATH PLANNING

- 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 permits 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)

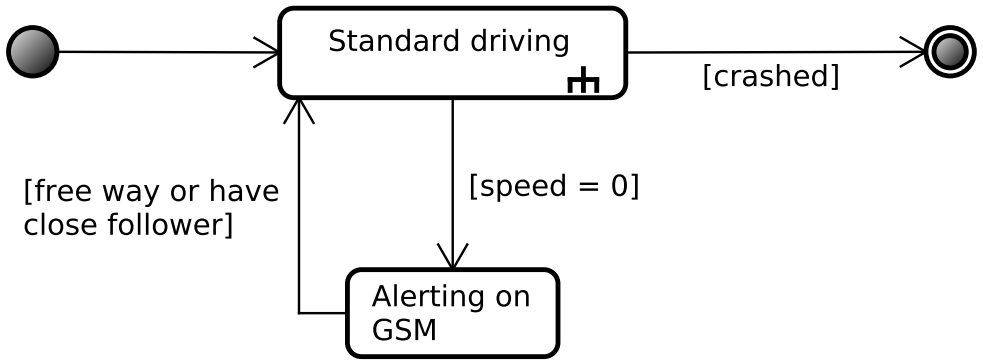
$$\texttt{followerDriving} = \begin{cases} -\dfrac{(v\Delta v)^2}{4b\Delta p^2} & \text{if the ahead object is far} \\ -a\dfrac{(s+vw)^2}{\Delta p^2} & \text{if the ahead object is near} \end{cases}$$

- Free driving:

$$\texttt{freeDriving} = a\left(1 - \left(\frac{v}{v_c}\right)^4\right)$$

What is simulated?

1. Vehicles on a French highway.
2. Danger event → “an animal is crossing the highway and causes a crash”.
3. Alert events by GSM.
4. Arrival of the security and rescue services.



voxelia

http://www.voxelia.com

VIDEO

Video done with the SIMULATE® tool — 2012 © Voxelia S.A.S

PEDESTRIAN SIMULATION

What is simulated?

1. Movements of pedestrians at a microscopic level.
2. Force-based model for avoiding collisions.

(Buisson, 2013)

http://www.voxelia.com

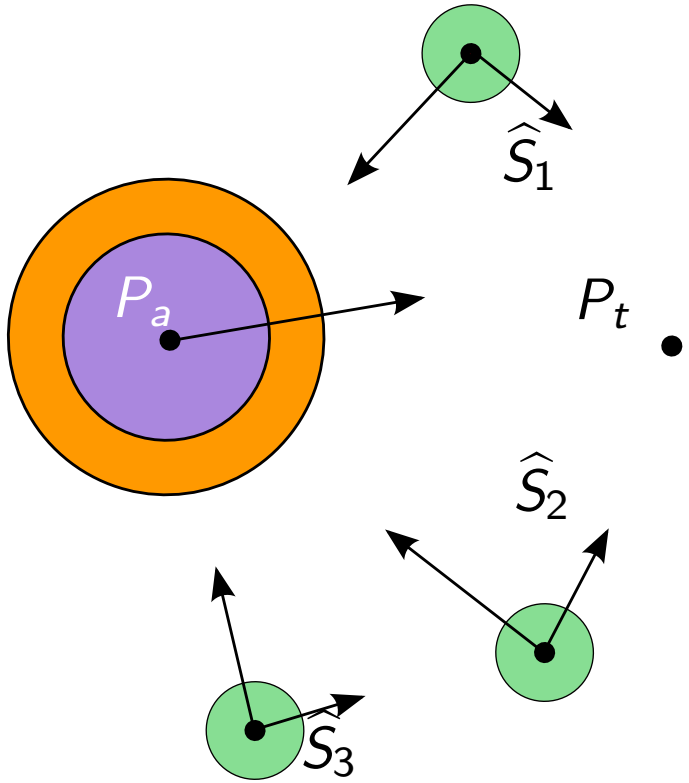
FORCE TO APPLY TO EACH AGENT

The force to apply to each agent is:

$$\vec{F_a} = \vec{F} + w_a.\delta_{\|\vec{F}\|} \frac{\vec{p_t} - p_a}{\|\vec{p_t} - p_a\|}$$

$$\vec{F} = \sum_{i \in M} U(t_c^i) \cdot \hat{S}_i$$

- $\vec{F}$: collision-avoidance force.
- $\hat{S}_i$: a sliding force.
- t_c^i: time to collision to object i.
- $U(t)$: scaling function of the time to collision.
- M: set objects around (including the other agents).
- w_a: weight of the attractive force.
- $\delta_x g$: is g if $x \leq 0$, 0 otherwise.



Buisson (2013)

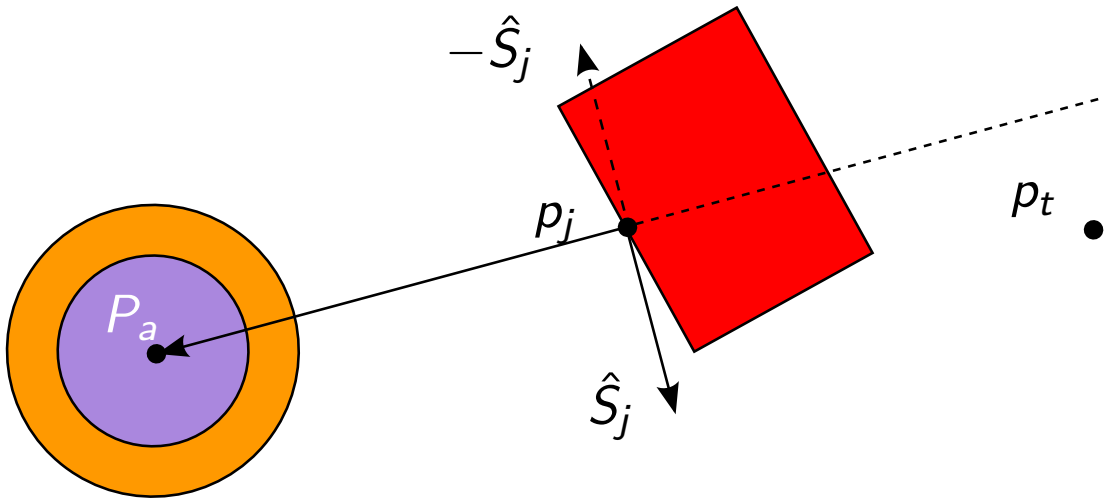
SLIDING FORCE

- The sliding force $\vec{S_i}$ is:

$$\vec{s_j} = (p_j - p_a) \times \hat{y}$$

$$\hat{S}_j = \text{sign}(\vec{s_j} \cdot (\vec{p_t} - p_a)) \frac{\vec{s_j}}{\|\vec{s_j}\|}$$

- $\hat{y}$: vertical unit vector.



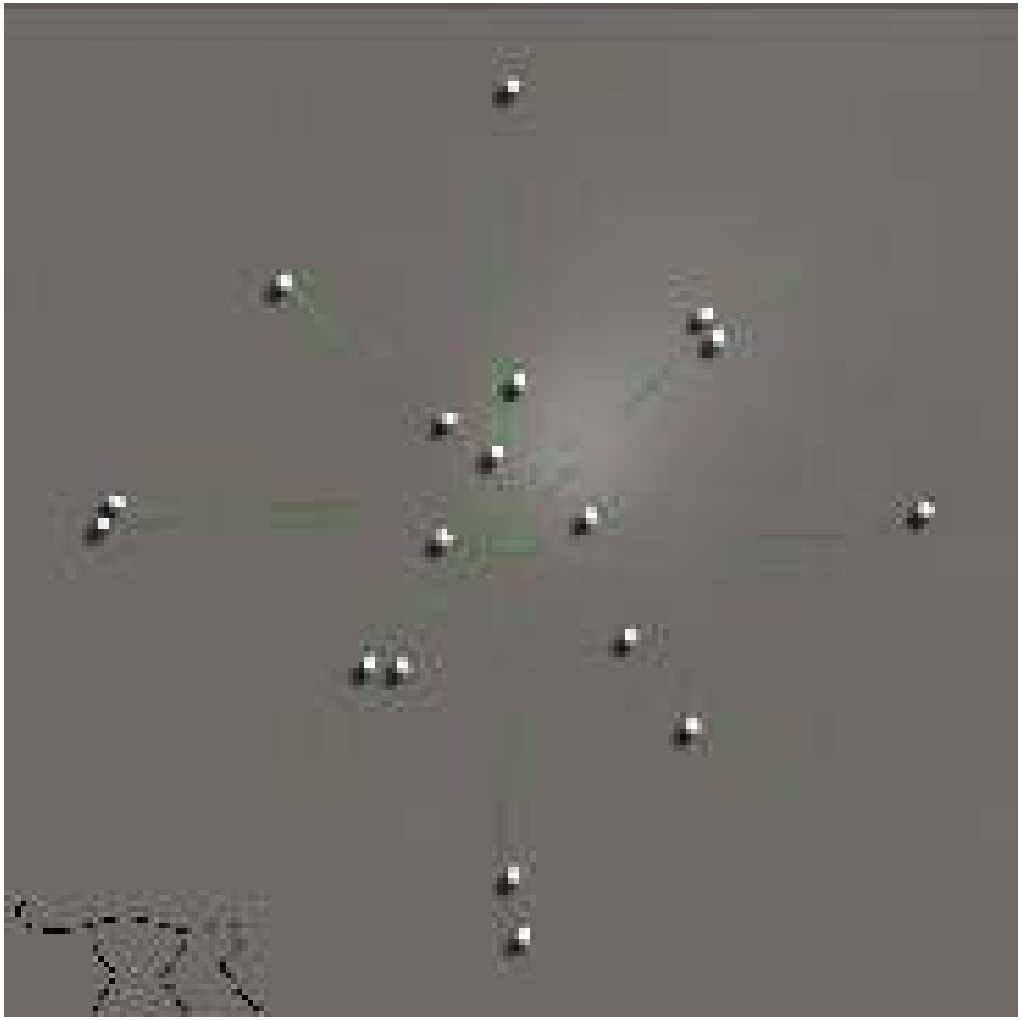
SCALING THE SLIDING FORCE

- How to scale $\hat{S}_j$ to obtain the repulsive force?
- Many force-based models use a monotonic decreasing function of the distance to an obstacle.
- But it does not support the velocity of the agent.
- Solution: Use time-based force scaling function.

$$U(t) = \begin{cases} \frac{\sigma}{t^{\phi}} - \frac{\sigma}{t_{max}{}^{\phi}} & \text{if } 0 \leq t \leq t_{max} \\ 0 & \text{if } t > t_{max} \end{cases}$$

- t: estimated time to collision.
- t_{max}: the maximum anticipation time.
- σ and ϕ are constants, such that $U(t_{max}) = 0$.

VIDEO (1/2)



Video done with the SIMULATE® tool — 2014 © Voxelia S.A.S

http://www.voxelia.com

VIDEO (2/2)

Video done with the SIMULATE® tool — 2014 © Voxelia S.A.S

http://www.voxelia.com

OUTLINE

KEY ELEMENTS FOR THIS CHAPTER

- Key Concepts: Immersion, Interaction, Avatar, Animat, Shadow Avatar, Persistence, Environment Dynamics.
- Key Modeling Approach: Multiagent Systems
- Agent Concepts: Agent, Multiagent System, Environment.
- Simulation: Simulator Architecture.
- Design Principles: Simple behavior model, Complex behavior's perception.

Thank you for your attention...